Chapter 6

Advanced Memory Management and Garbage Collection in OpenJDK

Introduction

In this chapter, we will dive into the world of JVM performance engineering, examining advanced memory management and garbage collection techniques in the OpenJDK Hotspot VM. Our focus will be on the evolution of optimizations and algorithms from JDK 11 to JDK 17.

Automatic memory management in modern OpenJDK is a crucial aspect of the JVM. It empowers developers to leverage optimized algorithms that provide efficient allocation paths and adapt to the applications' demands. Operating within a managed runtime environment minimizes the chances of memory leaks, dangling pointers, and other memory-related bugs that can be challenging to identify and resolve.

My journey with GCs started during my tenure at Advanced Micro Devices (AMD) and then gained wings at Sun Microsystems and Oracle, where I was deeply involved in the evolution of garbage collection technologies. These experiences provided me with a unique perspective on the development and optimization of garbage collectors, which I'll share with you in this chapter.

Among the many enhancements to the JVM, JDK 11 introduced significant improvements to garbage collection, such as abortable mixed collections, which improved GC responsiveness, and the Z Garbage Collector, which aimed to achieve sub-millisecond pauses.

This chapter explores these developments, including optimizations like Thread-Local Allocation Buffers and Non-Uniform Memory Architecture–aware GC. We will also delve into various transactional workloads, examining their interplay with in-memory databases and their impact on the Java heap.

By the end of this chapter, you'll have a deeper understanding of advanced memory management in OpenJDK and be equipped with the knowledge to effectively optimize GC for your Java applications. So, let's dive in and explore the fascinating world of advanced memory management and garbage collection!

Overview of Garbage Collection in Java

Java's GC is an automatic, adaptive memory management system that recycles heap memory used by objects no longer needed by the application. For most GCs in OpenJDK, the JVM segments heap memory into different generations, with most objects allocated in the young generation, which is further divided into the eden and survivor spaces. Objects move from the eden space to the survivor space, and potentially to the old generation, as they survive subsequent minor GC cycles.

This generational approach leverages the "weak generational hypothesis," which states that most objects become unreachable quickly, and there are fewer references from old to young objects. By focusing on the young generation during minor GC cycles, the GC process becomes more efficient.

In the GC process, the collector first traces the object graph starting from root nodes—objects directly accessible from an application. This phase, also known as marking, involves explicitly identifying all live objects. Concurrent collectors might also implicitly consider certain objects as live, contributing to the GC footprint and leading to a phenomenon known as "floating garbage." This occurs when the GC retains references that may no longer be live. After completing the tracing, any objects not reachable during this traversal are considered "garbage" and can be safely deallocated.

OpenJDK HotSpot VM offers several built-in garbage collectors, each designed with specific goals, from increasing throughput and reducing latency to efficiently handling larger heap sizes. In our exploration, we'll focus on two of these collectors—the G1 Garbage Collector and Z Garbage Collector (ZGC). My direct involvement in algorithmic refinement and performance optimization of the G1 collector has given me in-depth insights into its advanced memory management techniques. Along with the G1 GC, we'll also explore ZGC, particularly focusing on their recent enhancements in OpenJDK.

The G1 GC, a revolutionary feature fully supported since JDK 7 update 4, is designed to maximize the capabilities of modern multi-core processors and large memory banks. More than just a garbage collector, G1 is a comprehensive solution that balances responsiveness with throughput. It consistently meets GC pause-time goals, ensuring smooth operation of applications while delivering impressive processing capacity. G1 operates in the background, concurrently with the application, traversing the live object graph and building the collection set for different regions of the heap. It minimizes pauses by splitting the heap into smaller, more manageable pieces, known as regions, and processing each as an independent entity, thereby enhancing the overall responsiveness of the application.

The concept of regions is central to the operation of both G1 and ZGC. The heap is divided into multiple, independently collectible regions. This approach allows the garbage collectors

to focus on collecting those regions that will yield substantial free space, thereby improving the efficiency of the GC process.

ZGC, first introduced as an experimental feature in JDK 11, is a low-latency GC designed for scalability. This state-of-the-art GC ensures that GC pause times are virtually independent of the size of the heap. It achieves this by employing a concurrent copying technique that relocates objects from one region of the memory to another while the application is still running. This concurrent relocation of objects significantly reduces the impact of GC pauses on application performance.

ZGC's design allows Java applications to scale more predictably in terms of memory and latency, making it an ideal choice for applications with large heap sizes and stringent latency requirements. ZGC works alongside the application, operating concurrently to keep pauses to a minimum and maintain a high level of application responsiveness.

In this chapter, we'll delve into the details of these GCs, their characteristics, and their impact on various application types. We'll explore their intricate mechanisms for memory management, the role of thread-local and Non-Uniform Memory Architecture (NUMA)–aware allocations, and ways to tune these collectors for optimal performance. We'll also explore practical strategies for assessing GC performance, from initial measurement to fine-tuning. Understanding this iterative process and recognizing workload patterns will enable developers to select the most suitable GC strategy, ensuring efficient memory management in a wide range of applications.

Thread-Local Allocation Buffers and Promotion-Local Allocation Buffers

Thread-Local Allocation Buffers (TLABs) are an optimization technique employed in the OpenJDK HotSpot VM to improve allocation performance. TLABs reduce synchronization overhead by providing separate memory regions in the heap for individual threads, allowing each thread to allocate memory without the need for locks. This lock-free allocation mechanism is also known as the "fast-path."

In a TLAB-enabled environment, each thread is assigned its buffer within the heap. When a thread needs to allocate a new object, it can simply bump a pointer within its TLAB. This approach makes the allocation process go much faster than the process of acquiring locks and synchronizing with other threads. It is particularly beneficial for multithreaded applications, where contention for a shared memory pool can cause significant performance degradation.

TLABs are resizable and can adapt to the allocation rates of their respective threads. The JVM monitors the allocation behavior of each thread and can adjust the size of the TLABs accordingly. This adaptive resizing helps strike a balance between reducing contention and making efficient use of the heap space.

To fine-tune TLAB performance, the JVM provides several options for configuring TLAB behavior. These options can be used to optimize TLAB usage for specific application workloads and requirements. Some of the key tuning options include

- **`UseTLAB`:** Enables or disables TLABs. By default, TLABs are enabled in the HotSpot VM. To disable TLABs, use `-XX:-UseTLAB`.
- **`-XX:TLABSize=<value>`:** Sets the initial size of TLABs in bytes. Adjusting this value can help balance TLAB size with the allocation rate of individual threads. For example, `-XX:TLABSize=16384` sets the initial TLAB size to 16 KB.
- **`ResizeTLAB`:** Controls whether TLABs are resizable. By default, TLAB resizing is enabled. To disable TLAB resizing, use `-XX:-ResizeTLAB`.
- **`-XX:TLABRefillWasteFraction=<value>`:** Specifies the maximum allowed waste for TLABs, as a fraction of the TLAB size. This value influences the resizing of TLABs, with smaller values leading to more frequent resizing. For example, `-XX:TLABRefillWasteFraction=64` sets the maximum allowed waste to 1/64th of the TLAB size.

By experimenting with these tuning options and monitoring the effects on your application's performance, you can optimize TLAB usage to better suit your application's allocation patterns and demands, ultimately improving its overall performance. To monitor TLAB usage, you can use tools such as Java Flight Recorder (JFR) and JVM logging.

For Java Flight Recorder in JDK 11 and later, you can start recording using this command-line option:

```
-XX:StartFlightRecording=duration=300s,jdk.ObjectAllocationInNewTLAB#enabled=true,
jdk.ObjectAllocationOutsideTLAB#enabled=true,filename=myrecording.jfr
```

This command will record a 300-second profile of your application and save it as `myrecording.jfr`. You can then analyze the recording with tools like JDK Mission Control[1] to visualize TLAB-related statistics.

For JVM logging, you can enable TLAB-related information with the Unified Logging system using the following option:

```
-Xlog:gc*,gc+tlab=debug:file=gc.log:time,tags
```

This configuration logs all GC-related information (`gc*`) along with TLAB information (`gc+tlab=debug`) to a file named `gc.log`. The time and tags decorators are used to include timestamps and tags in the log output. You can then analyze the log file to study the effects of TLAB tuning on your application's performance.

Similar to TLABs, the OpenJDK HotSpot VM employs Promotion-Local Allocation Buffers (PLABs) for GC threads to promote objects. Many GC algorithms perform depth-first copying into these PLABs to enhance co-locality. However, not all promoted objects are copied into PLABs; some objects might be directly promoted to the old generation. Like TLABs, PLABs are automatically resized based on the application's promotion rate.

[1] https://jdk.java.net/jmc/8/

For generational GCs in HotSpot, it is crucial to ensure proper sizing of the generations and allow for appropriate aging of the objects based on the application's allocation rate. This approach ensures that mostly (and possibly only) long-lived static and transient objects are tenured ("promoted"). Occasionally, spikes in the allocation rate may unnecessarily increase the promotion rate, which could trigger an old-generation collection. A properly tuned GC will have enough headroom that such spikes can be easily accommodated without triggering a full stop-the-world (STW) GC event, which is a fail-safe or fallback collection for some of the newer GCs.

To tune PLABs, you can use the following options, although these options are less frequently used because PLABs are usually self-tuned by the JVM:

- **`-XX:GCTimeRatio=<value>`:** Adjusts the ratio of GC time to application time. Lower values result in larger PLABs, potentially reducing GC time at the cost of increased heap usage.
- **`ResizePLAB`:** Enables or disables the adaptive resizing of PLABs. By default, it is enabled.
- **`-XX:PLABWeight=<value>`:** Sets the percentage of the current PLAB size added to the average PLAB size when computing the new PLAB size. The default value is 75.

PLABs can help improve GC performance by reducing contention and providing a faster promotion path for objects. Properly sized PLABs can minimize the amount of memory fragmentation and allow for more efficient memory utilization. However, just as with TLABs, it is essential to strike a balance between reducing contention and optimizing memory usage. If you suspect that PLAB sizing might be an issue in your application, you might consider adjusting the `-XX:PLABWeight` and `-XX:ResizePLAB` options, but be careful and always test their impact on application performance before applying changes in a production environment.

Optimizing Memory Access with NUMA-Aware Garbage Collection

In OpenJDK HotSpot, there is an architecture-specific allocator designed for Non-Uniform Memory Access (NUMA) systems called "NUMA-aware." As discussed in Chapter 5, "End-to-End Java Performance Optimization: Engineering Techniques and Micro-benchmarking with JMH," NUMA is a type of computer memory design that is utilized in multiprocessing. The speed of memory access is dependent on the location of the memory in relation to the processor. On these systems, the operating system allocates physical memory based on the "first-touch" principle, which states that memory is allocated closest to the processor that first accesses it. Hence, to ensure allocations occur in the memory area nearest to the allocating thread, the eden space is divided into segments for each node (where a node is a combination of a CPU, memory controller, and memory bank). Using a NUMA-aware allocator can help improve performance and reduce memory access latency in multiprocessor systems.

To better understand how this works, let's take a look at an example. In Figure 6.1, you can see four nodes labeled Node 0 to Node 3. Each node has its own processing unit and a memory controller plus memory bank combination. A process/thread that is running on Node 0 but

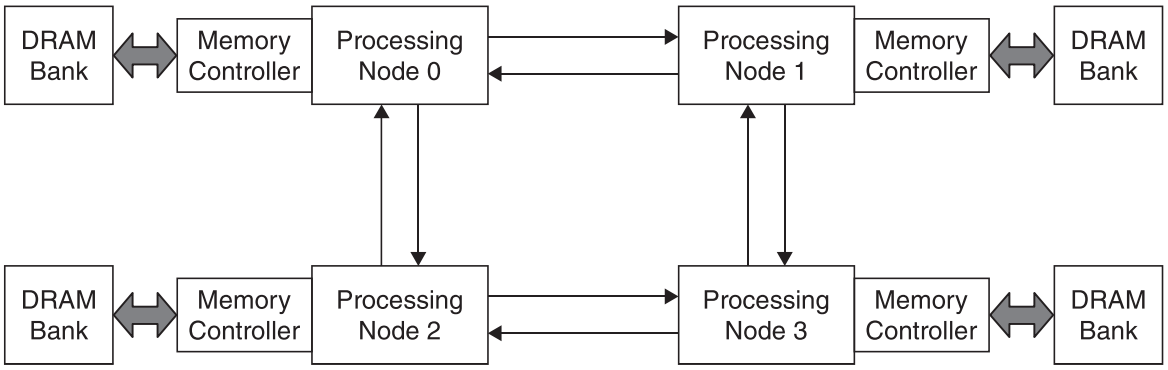

Figure 6.1 NUMA Nodes with Processors, Memory Controllers, and DRAM Banks

wants to access memory on Node 3 will have to reach Node 3 either via Node 2 or via Node 1. This is considered two hops. If a process/thread on Node 0 wants to access Node 3 or Node 2's memory bank, then it's just one hop away. A local memory access needs no hops. An architecture like that depicted in Figure 6.1 is called a NUMA.

If we were to divide the eden space of the Java heap so that we have areas for each node and also have threads running on a particular node allocate out of those node-allocated areas, then we would be able to have hop-free access to node-local memory (provided the scheduler doesn't move the thread to a different node). This is the principle behind the NUMA-aware allocator—basically, the TLABs are allocated from the closest memory areas, which allows for faster allocations.

In the initial stages of an object's life cycle, it resides in the eden space allocated specifically from its NUMA node. Once a few of the objects have survived or when the objects need to be promoted to the old generation, they might be shared between threads across different nodes. At this point, allocation occurs in a node-interleaved manner, meaning memory chunks are sequentially and evenly distributed across nodes, from Node 0 to the highest-numbered node, until all of the memory space has been allocated.

Figure 6.2 illustrates a NUMA-aware eden. Thread 0 and Thread 1 allocate memory out of the eden area for Node 0 because they are both running on Node 0. Meanwhile, Thread 2 allocates out of the eden area for Node 1 because it's running on Node 1.

To enable NUMA-aware GC in JDK 8 and later, use the following command-line option:

```
-XX:+UseNUMA
```

This option enables NUMA-aware memory allocations, allowing the JVM to optimize memory access for NUMA architectures.

To check if your JVM is running with NUMA-aware GC, you can use the following command-line option:

```
-XX:+PrintFlagsFinal
```

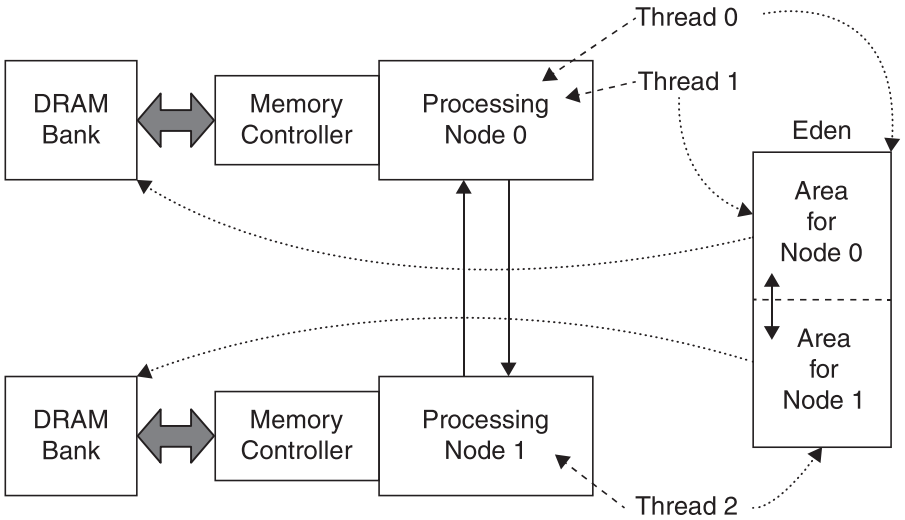

Figure 6.2 The Inner Workings of a NUMA-Aware GC

This option outputs the final values of the JVM flags, including the `UseNUMA` flag, which indicates if NUMA-aware GC is enabled.

In summary, the NUMA-aware allocator is designed to optimize memory access for NUMA architectures by dividing the eden space of the Java heap into areas for each node and creating TLABs from the closest memory areas. This can result in faster allocations and improved performance. However, when enabling NUMA-aware GC, it's crucial to monitor your application's performance, as the impacts might differ based on your system's hardware and software configurations.

Some garbage collectors, such as G1 and the Parallel GC, benefit from NUMA optimizations. G1 was made NUMA-aware in JDK 14,[2] enhancing its efficiency in managing large heaps on NUMA architectures. ZGC is another modern garbage collector designed to address the challenges of large heap sizes, low-latency requirements, and better overall performance. ZGC became NUMA-aware in JDK 15.[3] In the following sections, we will delve into the key features and enhancements of G1 and ZGC, as well as their performance implications and tuning options in the context of modern Java applications.

Exploring Garbage Collection Improvements

As Java applications continue to evolve in complexity, the need for efficient memory management and GC becomes increasingly important. Two significant garbage collectors in the OpenJDK HotSpot VM, G1 and ZGC, aim to improve garbage collection performance, scalability, and predictability, particularly for applications with substantial heaps sizes and stringent low-latency demands.

[2] https://openjdk.org/jeps/345

[3] https://wiki.openjdk.org/display/zgc/Main#Main-EnablingNUMASupport

This section highlights the salient features of each garbage collector, listing the key advancements and optimizations made from JDK 11 to JDK 17. We will explore G1's enhancements in mixed collection performance, pause-time predictability, and its ability to better adapt to different workloads. As ZGC is a newer garbage collector, we will also dive deeper into its unique capabilities, such as concurrent garbage collection, reduced pause times, and its suitability for large heap sizes. By understanding these advances in GC technology, you will gain valuable insights into their impact on application performance and memory management, and the trade-offs to consider when choosing a garbage collector for your specific use case.

G1 Garbage Collector: A Deep Dive into Advanced Heap Management

Over the years, there has been a significant shift in the design of GCs. Initially, the focus was on throughput-oriented GCs, which aimed to maximize the overall efficiency of memory management processes. However, with the increasing complexity of applications and heightened user expectations, the emphasis has shifted towards latency-oriented GCs. These GCs aim to minimize the pause times that can disrupt application performance, even if it means a slight reduction in overall throughput. G1, which is the default GC for JDK 11 LTS and beyond, is a prime example of this shift toward latency-oriented GCs. In the subsequent sections, we'll examine the intricacies of the G1 GC, its heap management, and its advantages.

Regionalized Heap

G1 divides the heap into multiple equal-size regions, allowing each region to play the role of eden, survivor, or old space dynamically. This flexibility aids in efficient memory management.

Adaptive Sizing

- **Generational sizing:** G1's generational framework is complemented by its adaptive sizing capability, allowing it to adjust the sizes of the young and old generations based on runtime behavior. This ensures optimal heap utilization.
- **Collection set (CSet) sizing:** G1 dynamically selects a set of regions for collection to meet the pause-time target. The CSet is chosen based on the amount of garbage those regions contain, ensuring efficient space reclamation.
- **Remembered set (RSet) coarsening:** G1 maintains remembered sets to track cross-region references. Over time, if these RSets grow too large, G1 can coarsen them, reducing their granularity and overhead.

Pause-Time Predictability

- **Incremental collection:** G1 breaks down garbage collection work into smaller chunks, processing these chunks incrementally to ensure that pause time remains predictable.
- **Concurrent marking enhancement:** Although CMS introduced concurrent marking, G1 enhances this phase with advanced techniques like snapshot-at-the-beginning (SATB)

marking.[4] G1 employs a multiphase approach that is intricately linked with its region-based heap management, ensuring minimal disruptions and efficient space reclamation.

- **Predictions in G1:** G1 uses a sophisticated set of predictive mechanisms to optimize its performance. These predictions, grounded in G1's history-based analysis, allow it to adapt dynamically to the application's behavior and optimize performance:
  - **Single region copy time:** G1 forecasts the time it will take to copy/evacuate the contents of a single region by drawing on historical data for past evacuations. This prediction is pivotal in estimating the total pause time of a GC cycle and in dividing the number of regions to include in a mixed collection to adhere to the desired pause time.
  - **Concurrent marking time:** G1 anticipates the time required to complete the concurrent marking phase. This foresight is critical in timing the initiation of the marking phase by dynamically adjusting the IHOP—the initiating heap occupancy percent (`-XX:InitiatingHeapOccupancyPercent`). The adjustment of IHOP is based on real-time analysis of the old generation's occupancy and the application's memory allocation behavior.
  - **Mixed garbage collection time:** G1 predicts the time for mixed garbage collections to strategically select a set of old regions to collect along with the young generation. This estimation helps G1 in maintaining a balance between reclaiming sufficient memory space and adhering to pause-time targets.
  - **Young garbage collection time:** G1 calculates the time required for young-only collections to guide the resizing of the young generation. G1 adjusts the size of the young generation based on these predictions to optimize pause times based on the desired target (`-XX:MaxGCPauseTimeMillis`) and adapt to the application's changing allocation patterns.
  - **Amount of reclaimable space:** G1 estimates the space that will be reclaimed by collecting specific old regions. This estimate informs the decision-making process for including old regions in mixed collections, optimizing memory utilization, and minimizing the risk of heap fragmentation.

Core Scaling

G1's design inherently supports multithreading, allowing it to scale with the number of available cores. This has the following implications:

- **Parallelized young-generation collections:** Ensuring quick reclamation of short-lived objects.
- **Parallelized old-generation collections:** Enhancing the efficiency of space reclamation in the old generation.
- **Concurrent threads during the marking phase:** Speeding up the identification of live objects.

[4] Charlie Hunt, Monica Beckwith, Poonam Parhar, and Bengt Rutisson. *Java Performance Companion.* Boston, MA: Addison-Wesley Professional, 2016.

Pioneering Features

Beyond the regionalized heap and adaptive sizing, G1 introduced several other features:

- **Humongous objects handling:** G1 has specialized handling for large objects that span multiple regions, ensuring they don't cause fragmentation or extended pause times.
- **Adaptive threshold tuning (IHOP):** As mentioned earlier, G1 dynamically adjusts the initiation threshold of the concurrent marking phase based on runtime metrics. This tuning is particularly focused on IHOP's dynamic adjustment to determine the right time to start the concurrent marking phase.

In essence, G1 is a culmination of innovative techniques and adaptive strategies, ensuring efficient memory management while providing predictable responsiveness.

Advantages of the Regionalized Heap

In G1, the division of the heap into regions creates a more flexible unit of collection compared to traditional generational approaches. These regions serve as G1's unit of work (UoW), enabling its predictive logic to efficiently tailor the collection set for each GC cycle by adding or removing regions as needed. By adjusting which regions are included in the collection set, G1 can respond dynamically to the application's current memory usage patterns.

A regionalized heap facilitates incremental compaction, enabling the collection set to include all young regions and at least one old region. This is a significant improvement over the monolithic old-generation collection, such as full GC in Parallel GC or the fallback full GC for CMS, which are less adaptable and efficient.

Traditional Versus Regionalized Java Heap Layouts

To better understand the benefits of a regionalized heap, let's compare it with a traditional Java heap layout. In a traditional Java heap (shown in Figure 6.3), there are separate areas designated for young and old generations. In contrast, a regionalized Java heap layout (shown in Figure 6.4) divides the heap into regions, which can be classified as either free or occupied. When a young region is released back to the free list, it can be reclaimed as an old region based on the promotion needs. This noncontiguous generation structure, coupled with the flexibility to resize the generations as needed, allows G1's predictive logic to achieve its service level objective (SLO) for responsiveness.

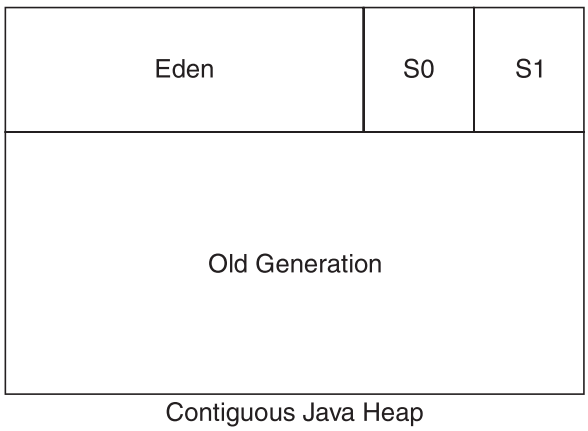

Figure 6.3 A Traditional Java Heap Layout

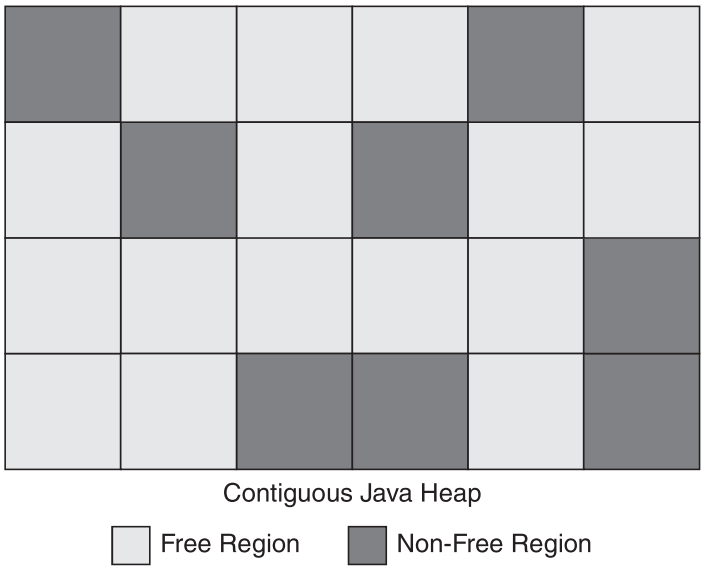

Figure 6.4 A Regionalized Java Heap Layout

Key Aspects of the Regionalized Java Heap Layout

One important aspect of G1 is the introduction of humongous objects. An object is considered humongous if it spans 50% or more of a single G1 region (Figure 6.5). G1 handles the allocation of humongous objects differently than the allocation of regular objects. Instead of following the fast-path (TLABs) allocation, humongous objects are allocated directly out of the old generation into designated humongous regions. If a humongous object spans more than one region size, it will require contiguous regions. (For more about humongous regions and objects, please refer to the *Java Performance Companion* book.[5])

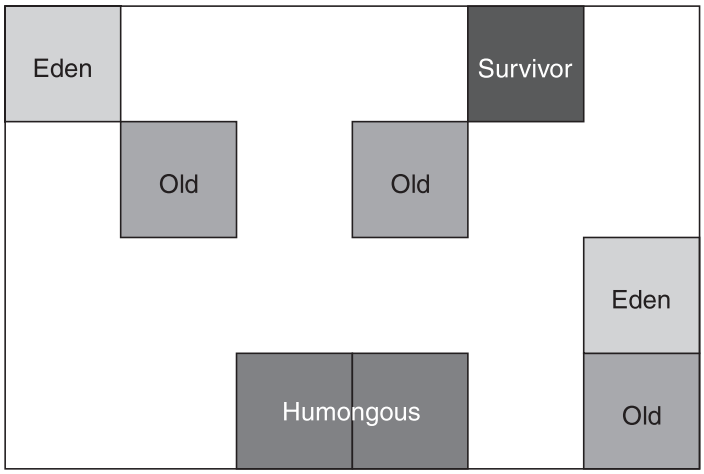

Figure 6.5 A Regionalized Heap Showing Old, Humongous, and Young Regions

[5] Charlie Hunt, Monica Beckwith, Poonam Parhar, and Bengt Rutisson. *Java Performance Companion.* Boston, MA: Addison-Wesley Professional, 2016: chapters 2 and 3.

Balancing Scalability and Responsiveness in a Regionalized Heap

A regionalized heap aims to enhance scalability while maintaining the target responsiveness and pause time. However, the command-line option `-XX:MaxGCPauseMillis` provides only soft-real time goals for G1. Although the prediction logic strives to meet the required goal, the collection pause doesn't have a hard-stop when the pause-time goal is reached. If your application experiences high GC tail latencies due to allocation (rate) spikes or increased mutation rates, and the prediction logic struggles to keep up, you may not consistently meet your responsiveness SLOs. In such cases, manual GC tuning (covered in the next section) may be necessary.

NOTE The mutation rate is the rate at which your application is updating references.

NOTE Tail latency refers to the slowest response times experienced by the end users. These latencies can significantly impact the user experience in latency-sensitive applications and are usually represented by higher percentiles (for example, 4-9s, 5-9s percentiles) of the latency distribution.

Optimizing G1 Parameters for Peak Performance

Under most circumstances, G1's automatic tuning and prediction logic work effectively when applications follow a predictable pattern, encompassing a warm-up (or ramp-up) phase, a steady-state, and a cool-down (or ramp-down) phase. However, challenges may arise when multiple spikes in the allocation rate occur, especially when these spikes result from bursty, transient, humongous objects allocations. Such situations can lead to premature promotions in the old generation and necessitate accommodating transient humongous objects/regions in the old generation. These factors can disrupt the prediction logic and place undue pressure on the marking and mixed collection cycles. Before recovery can occur, the application may encounter evacuation failures due to insufficient space for promotions or a lack of contiguous regions for humongous objects as a result of fragmentation. In these cases, intervention whether manual or via automated/AI systems becomes necessary.

G1's regionalized heap, collection set, and incremental collections offer numerous tuning knobs for adjusting internal thresholds, and they support targeted tuning when needed. To make well-informed decisions about modifying these parameters, it's essential to understand these tuning knobs and their potential impact on G1's ergonomics.

Optimizing GC Pause Responsiveness in JDK 11 and Beyond

To effectively tune G1, analyzing the pause histogram obtained from a GC log file can provide valuable insights into your application's performance. Consider the pause-time series plot from a JDK 11 GC log file, as shown in Figure 6.6. This plot reveals the following GC pauses:

- 4 young GC pauses followed by ...
- 1 initial mark pause when the IHOP threshold is crossed
- Next, 1 young GC pause during the concurrent mark phase ...
- And then 1 remark pause and 1 cleanup pause
- Finally, another young GC pause before 7 mixed GCs to incrementally reclaim the old-generation space

Assuming the system has an SLO requiring 100% of GC pauses to be less than or equal to 100 ms, it is evident that the last mixed collection did not meet this requirement. Such one-off pauses can potentially be mitigated by the abortable mixed collection feature introduced in JDK 12—which we will discuss in a bit. However, in cases where automated optimizations don't suffice or aren't applicable, manual tuning becomes necessary (Figure 6.7).

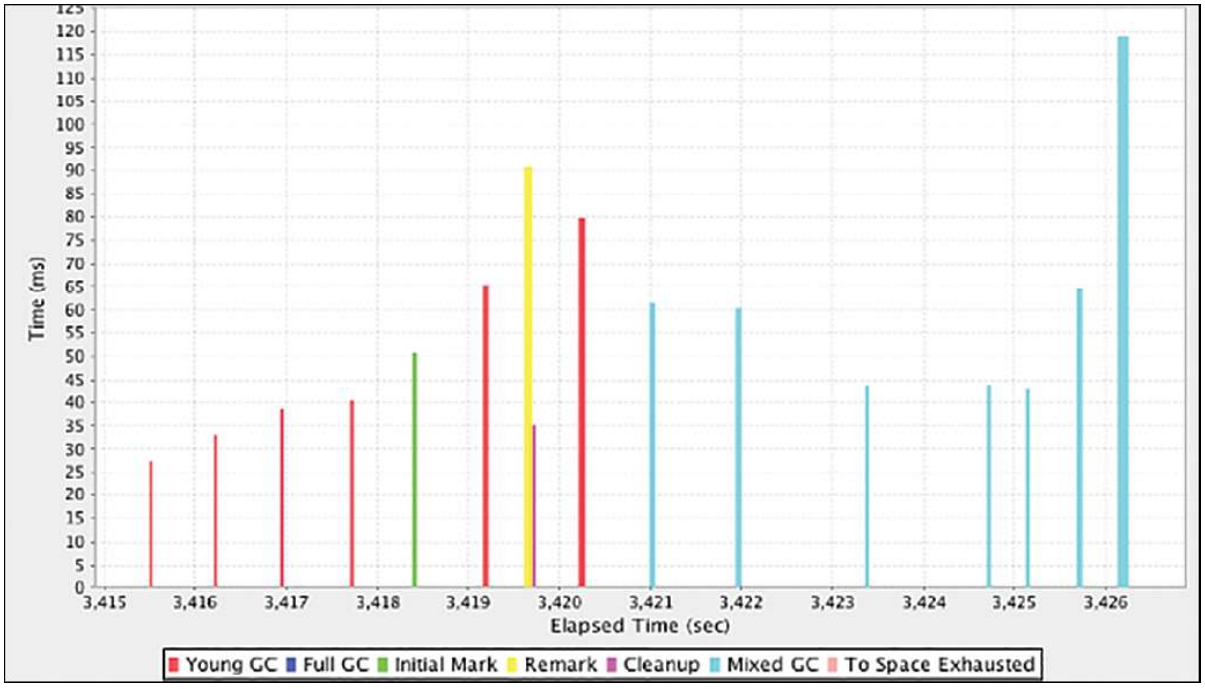

Figure 6.6 G1 Garage Collector's Pause-Time Histogram

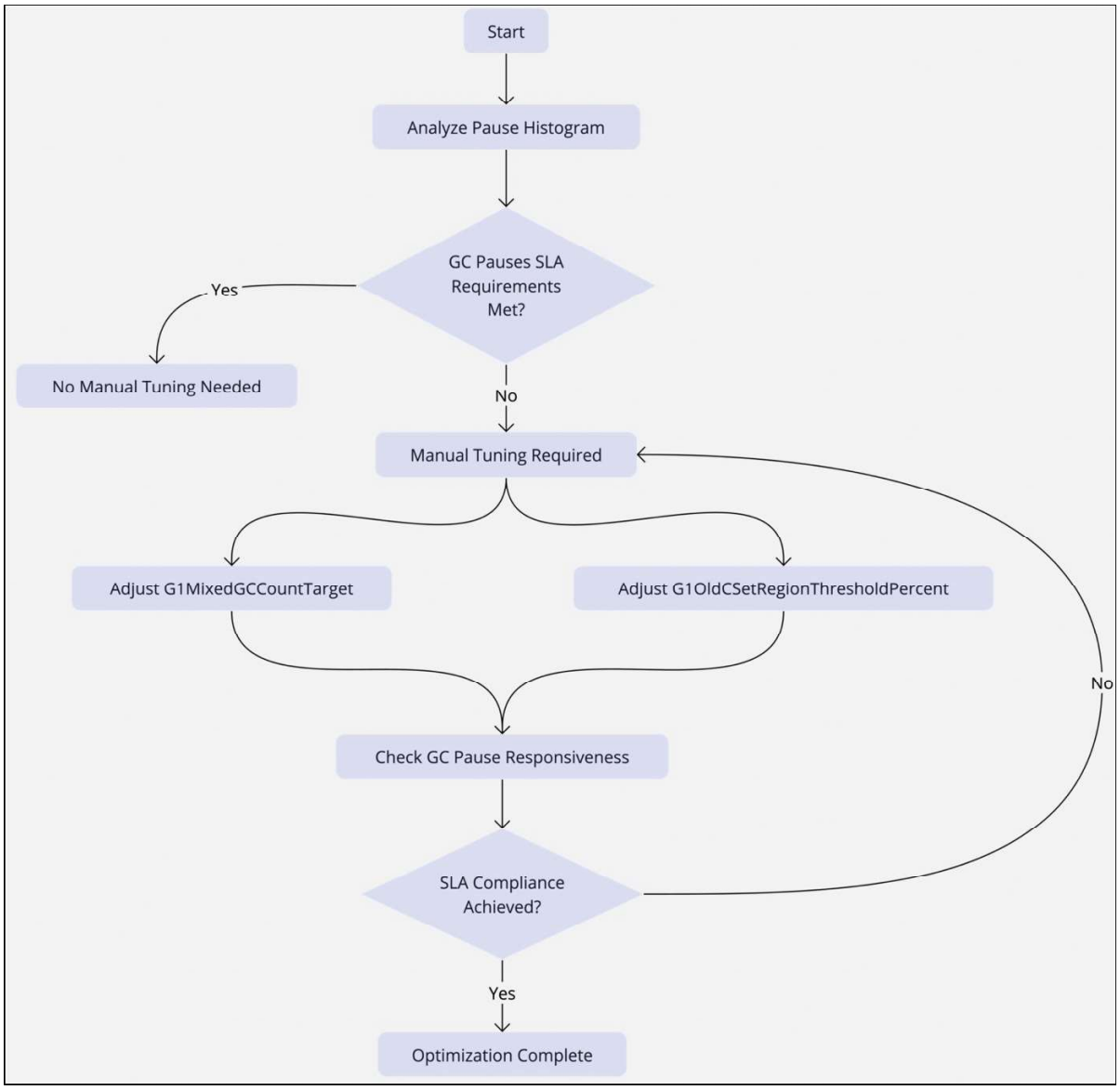

Figure 6.7 Optimizing G1 Responsiveness

Two thresholds can help control the number of old regions included in mixed collections:

- Maximum number of old regions to be included in the mixed collection set:`-XX:G1OldCSetRegionThresholdPercent=<p>`
- Minimum number of old regions to be included in the mixed collection set: `-XX:G1MixedGCCountTarget=<n>`

Both tunables help determine how many old regions are added to the young regions, forming the mixed collection set. Adjusting these values can improve pause times, ensuring SLO compliance.

In our example, as shown in the timeline of pause events (Figure 6.6), most young collections and mixed collections are below the SLO requirements except for the last mixed collection. A simple solution is to reduce the maximum number of old regions included in the mixed collection set by using the following command-line option:

```
-XX:G1OldCSetRegionThresholdPercent=<p>
```

where `p` is a percentage of the total Java heap. For example, with a heap size of 1 GB and a default value of 10% (i.e., `–Xmx1G – Xms1G -XX:G1OldCSetRegionThresholdPercent=10`), the count of old regions added to the collection set would be 10% of 1 GB (which is rounded up), so we get a count of 103. By reducing the percentage to 7%, the maximum count drops to 72, which would be enough to bring the last mixed-collection pause within the acceptable GC response time SLO. However, this change alone may increase the total number of mixed collections, which might be a concern.

G1 Responsiveness: Enhancements in Action

With advancements in JDK versions, several new features and enhancements have been introduced to G1. These features, when utilized correctly, can significantly improve the responsiveness and performance of applications without the need for extensive tuning. Although the official documentation provides a comprehensive overview of these features, there's immense value in understanding their practical applications. Here's how field experiences translate these enhancements into real-world performance gains:

- **Parallelizing reference processing (JDK 11):** Consider an e-commerce platform using `SoftReference` objects to cache product images. During high-traffic sales events, the number of references skyrockets. The parallel reference processing capability keeps the system agile under heavy loads, maintaining a smooth user experience even as traffic spikes.
- **Abortable mixed collections (JDK 12):** Picture a real-time multiplayer gaming server during intense gameplay, serving thousands of concurrent online players. This server can have medium-lived (transient) data (player, game objects). Here, real-time constraints with low tail latencies are the dominant SLO requirement. The introduction of abortable mixed collections ensures that gameplay stays fluid, even as in-game actions generate transient data at a frenzied pace.
- **Promptly returning unused committed memory (JDK 12):** A cloud-based video rendering service, which performs postproduction of 3D scenes, often sits on idle resources. G1 can return unused committed memory to the operating system more quickly based on the system load. With this ability, the service can optimize resource utilization, thereby ensuring that other applications on the same server have access to memory when needed, and in turn leading to cost savings in cloud hosting fees. To further fine-tune this behavior, two G1 tuning options can be utilized:
  - **`G1PeriodicGCInterval`:** This option specifies the interval (in milliseconds) at which periodic GC cycles are triggered. By setting this option, you can control how frequently G1 checks and potentially returns unused memory to the OS. The default is 0, which means that periodic GC cycles are disabled by default.

- **`G1PeriodicGCSystemLoadThreshold`:** This option sets a system load threshold. If the system load is below this threshold, a periodic GC cycle is triggered even if the `G1PeriodicGCInterval` has not been reached. This option can be particularly useful in scenarios where the system is not under heavy load, and it's beneficial to return memory more aggressively. The default is 0, meaning that the system load threshold check is disabled.

CAUTION Although these tuning options provide greater control over memory reclamation, it's essential to exercise caution. Overly aggressive settings might lead to frequent and unnecessary GC cycles, which could negatively impact the application's performance. Always monitor the system's behavior after making changes and adjust it accordingly to strike a balance between memory reclamation and application performance.

- **Improved concurrent marking termination (JDK 13):** Imagine a global financial analytics platform that processes billions of market signals for real-time trading insights. The improved concurrent marking termination in JDK 13 can significantly shrink GC pause times, ensuring that high-volume data analyses and reporting are not interrupted, and thereby providing traders with consistent, timely market intelligence.
- **G1 NUMA-awareness (JDK 14):** Envision a high-performance computing (HPC) environment tasked with running complex simulations. With G1's NUMA-awareness, the JVM is optimized for multi-socket systems, leading to notable improvements in GC pause times and overall application performance. For HPC workloads, where every millisecond counts, this means faster computation and more efficient use of hardware resources.
- **Improved concurrent refinement (JDK 15):** Consider a scenario in which an enterprise search engine must handle millions of queries across vast indexes. The improved concurrent refinement in JDK 15 reduces the number of log buffer entries that will be made, which translates into shorter GC pauses. This refinement allows for rapid query processing, which is critical for maintaining the responsiveness users expect from a modern search engine.
- **Improved heap management (JDK 17):** Consider the case of a large-scale Internet of Things (IoT) platform that aggregates data from millions of devices. JDK 17's enhanced heap management within G1 can lead to more efficient distribution of heap regions, which minimizes GC pause times and boosts application throughput. For an IoT ecosystem, this translates into faster data ingestion and processing, enabling real-time responses to critical sensor data.

Building on these enhancements, G1 also includes other optimizations that further enhance responsiveness and efficiency:

- **Optimizing evacuation:** This optimization streamlines the evacuation process, potentially reducing GC pause times. In transactional applications where quick response

times are critical, such as financial processing or real-time data analytics, optimizing evacuation can ensure faster transaction processing. This results in a system that remains responsive even under high load conditions.

- **Building remembered sets on the fly:** By constructing remembered sets during the marking cycle, G1 minimizes the time spent in the remark phase. This can be particularly beneficial in systems with massive data sets, such as a big data analytics platform, where quicker remark phases can lead to reduced processing times.
- **Improvements in remembered sets scanning:** These improvements facilitate shorter GC pause times—an essential consideration for applications such as live-streaming services, where uninterrupted data flow is critical.
- **Reduce barrier overhead:** Lowering the overhead associated with barriers can lead to shorter GC pause times. This enhancement is valuable in cloud-based software as a service (SaaS) applications, where efficiency directly translates into reduced operational costs.
- **Adaptive IHOP:** This feature allows G1 to adaptively adjust the IHOP value based on the current application behavior. For example, in a dynamic content delivery network, adaptive IHOP can improve responsiveness by efficiently managing memory under varying traffic loads.

By understanding these enhancements and the available tuning options, you can better optimize G1 to meet the responsiveness requirements for your specific application. Implementing these improvements thoughtfully can lead to substantial performance gains, ensuring that your Java applications remain efficient and responsive under a wide range of conditions.

Optimizing GC Pause Frequency and Overhead with JDK 11 and Beyond

Meeting stringent SLOs that require specific throughput goals and minimal GC overhead is critical for many applications. For example, an SLO might dictate that GC overhead must not exceed 5% of the total execution time. Analysis of pause-time histograms may reveal that an application is struggling to meet the SLO's throughput requirement due to frequent mixed GC pauses, resulting in excessive GC overhead. Addressing this challenge involves strategically tuning of G1 to improve its efficiency. Here's how:

- **Strategically eliminating expensive regions:** Tailor the mixed collection set by omitting costly old regions. This action, while accepting some uncollected garbage, or "heap waste," ensures the heap accommodates the live data, transient objects, humongous objects, and additional space for waste. Increasing the total Java heap size might be necessary to achieve this goal.
- **Setting live data thresholds:** Utilize `-XX:G1MixedGCLiveThresholdPercent=<p>` to determine a cutoff for each old region based on the percentage of live data in that region. Regions exceeding this threshold are not included in the mixed collection set to avoid costly collection cycles.
- **Heap waste management:** Apply `-XX:G1HeapWastePercent=<p>` to permit a certain percentage of the total heap to remain uncollected (that is, "wasted"). This tactic targets the costly regions at the end of the sorted array established during the marking phase.

Figure 6.8 presents an overview of this process.

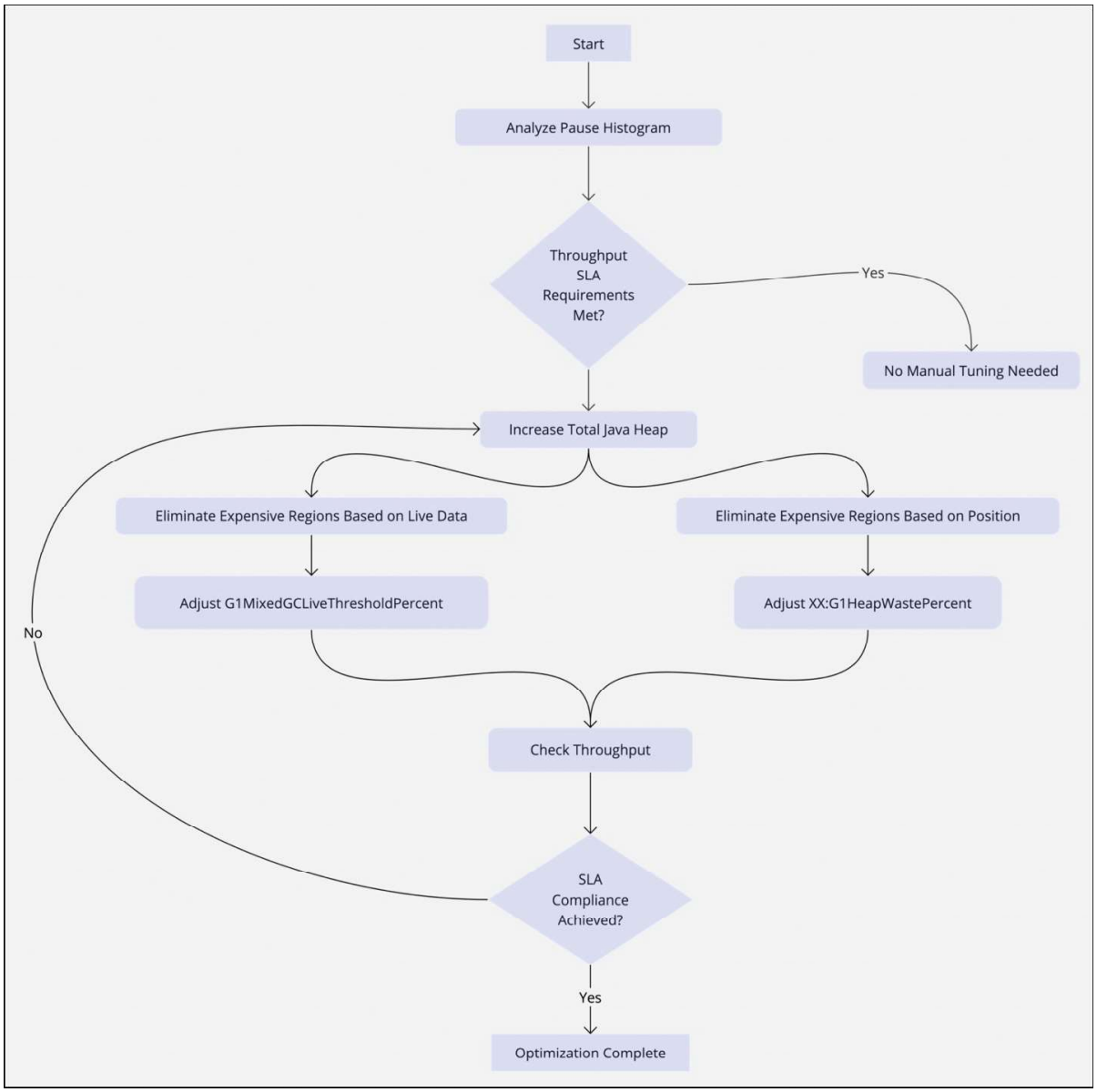

Figure 6.8 Optimizing G1's Throughput

G1 Throughput Optimizations: Real-World Applications

G1 has undergone a series of improvements and optimizations since Java 11, including the introduction of adaptive sizing and thresholds, that can help optimize its use to meet your specific application's throughput needs. These enhancements may reduce the need for fine-tuning for overhead reduction because they can lead to more efficient memory usage, thereby improving application performance. Let's look at these updates:

- **Eager reclaim of humongous objects (JDK 11):** In a social-media analytics application, large data sets representing user interactions are processed quite frequently. The capability to eagerly reclaim humongous objects prevents heap exhaustion and keeps analysis workflows running smoothly, especially during high-traffic events such as viral marketing campaigns.

- **Improved `G1MixedGCCountTarget` and abortable mixed collections (JDK 12):** Consider a high-volume transaction system in a stock trading application. By fine-tuning the number of mixed GCs and allowing abortable collections, G1 helps maintain consistent throughput even during market surges, ensuring traders experience minimal latency.
- **Adaptive heap sizing (JDK 15):** Cloud-based services, such as a document storage and collaboration platform, experience variable loads throughout the day. Adaptive heap sizing ensures that during periods of low activity, the system minimizes resource usage, reducing operational costs while still delivering high throughput during peak usage.
- **Concurrent humongous object allocation and young generation sizing improvements (JDK 17):** In a large-scale IoT data-processing service, concurrent allocation of large objects allows for more efficient handling of incoming sensor data streams. Improved young-generation sizing dynamically adjusts to the workload, preventing potential bottlenecks and ensuring steady throughput for real-time analytics.

In addition to these enhancements, the following optimizations contribute to improving the G1 GC's throughput:

- **Remembered sets space reductions:** In database management systems, reducing the space for remembered sets means more memory is available for caching query results, leading to faster response times and higher throughput for database operations.
- **Lazy threads and remembered sets initialization:** For an on-demand video streaming service, lazy initialization allows for quick scaling of resources in response to user demand, minimizing start-up delays and improving viewer satisfaction.
- **Uncommit at remark:** Cloud platforms can benefit from this feature by dynamically scaling down resource allocations during off-peak hours, optimizing cost-effectiveness without sacrificing throughput.
- **Old generation on NVDIMM:** By placing the old generation on nonvolatile DIMM (NVDIMM), G1 can achieve faster access times and improved performance. This can lead to more efficient use of the available memory and thus increasing throughput.
- **Dynamic GC threads with a smaller heap size per thread:** Microservices running in a containerized environment can utilize dynamic GC threads to optimize resource usage. With a smaller heap size per thread, the GC can adapt to the specific needs of each microservice, leading to more efficient memory management and improved service response times.
- **Periodic GC:** For IoT platforms that process continuous data streams, periodic garbage collection can help maintain a consistent application state by periodically cleaning up unused objects. This strategy can reduce the latency associated with garbage collection cycles, ensuring timely data processing and decision making for time-sensitive IoT operations.

By understanding these enhancements and the available tuning options, you can better optimize G1 to meet your specific application's throughput requirements.

Tuning the Marking Threshold

As mentioned earlier, Java 9 introduced an adaptive marking threshold, known as the IHOP. This feature should benefit most applications. However, certain pathological cases, such as bursty allocations of short-lived (possibly humongous) objects, often seen when building a short-lived transactional cache, could lead to suboptimal behavior, In such scenarios, a low adaptive IHOP might result in a premature concurrent marking phase, which completes without triggering the needed mixed collections—especially if the bursty allocations occur afterward. Alternatively, a high adaptive IHOP that doesn't decrease quickly enough could delay the concurrent marking phase initiation, leading to regular evacuation failures.

To address these issues, you can disable the adaptive marking threshold and set it to a steady value by adding the following option to your command line:

```
-XX:-G1UseAdaptiveIHOP -XX:InitiatingHeapOccupancyPercent=<p>
```

where p is the occupancy percentage of the total heap at which you would like the marking cycle to start.

In Figure 6.9, after the mixed GC pause at approximately 486.4 seconds, there is a young GC pause followed by the start of a marking cycle (the line after the young GC pause is the initiating mark pause). After the initial mark pause, we observe 23 young GC pauses, a "to"-space exhausted pause, and finally a full GC pause. This pattern, in which a mixed collection cycle has just been completed and a new concurrent marking cycle begins but is unable to finish while the size of the young generation is being reduced (hence the young collections), indicates issues with the adaptive marking threshold. The threshold may be too high, delaying the marking cycle to a point where back-to-back concurrent marking and mixed collection cycles are needed, or the concurrent cycle may be taking too long to complete due to a low number of concurrent marking threads that can't keep up with the workload.

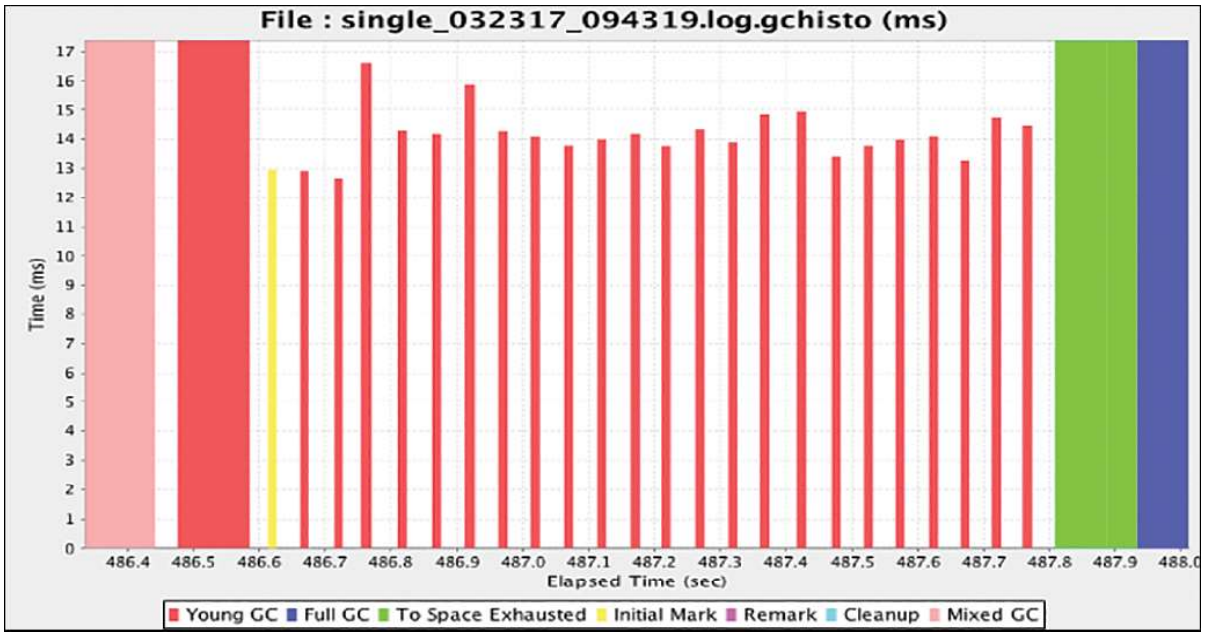

Figure 6.9 A G1 Garbage Collector Pause Timeline Showing Incomplete Marking Cycle

In such cases, you can stabilize the IHOP value to a lower level to accommodate the static and transient live data set size (LDS) (and humongous objects) and increase your concurrent threads count by setting `n` to a higher value: `-XX:ConcGCThreads=<n>`. The default value for `n` is one-fourth of your GC threads available for parallel (STW) GC activity.

G1 represents an important milestone in the evolution of GCs in the Java ecosystem. It introduced a regionalized heap and enabled users to specify realistic pause-time goals and maximize desired heap sizes. G1 predicts the total number of regions per collection (young and mixed) to meet the target pause time. However, the actual pause time might still vary due to factors such as the density of data structures, the popularity of regions or objects in the collected regions, and the amount of live data per region. This variability can make it challenging for applications to meet their already defined SLOs and creates a demand for a near-real-time, predictable, scalable, and low-latency garbage collector.

Z Garbage Collector: A Scalable, Low-Latency GC for Multi-terabyte Heaps

To meet the need for real-time responsiveness in applications, the Z Garbage Collector (ZGC) was introduced, advancing OpenJDK's garbage collection capabilities. ZGC incorporates innovative concepts into the OpenJDK universe—some of which have precedents in other collectors like Azul's C4 collector.[6] Among ZGC's pioneering features are concurrent compaction, colored pointers, off-heap forwarding tables, and load barriers, which collectively enhance performance and predictability.

ZGC was first introduced as an experimental collector in JDK 11 and reached production readiness with JDK 15. In JDK 16, concurrent thread stack processing was enabled, allowing ZGC to guarantee that pause times would remain below the millisecond threshold, independent of LDS and heap size. This advancement enables ZGC to support a wide spectrum of heap sizes, ranging from as little as 8 MB to as much as 16 TB, allowing it to cater to a diverse array of applications from lightweight microservices to memory-intensive domains like big data analytics, machine learning, and graphics rendering.

In this section, we'll take a look at some of the core concepts of ZGC that have enabled this collector to achieve ultra-low pauses and high predictability.

ZGC Core Concepts: Colored Pointers

One of the key innovations of ZGC is its use of colored pointers, a form of metadata tagging that allows the garbage collector to store additional information directly within the pointers. This technique, known as virtual address mapping or tagging, is achieved through multi-mapping on the x86-64 and aarch64 architectures. The colored pointers can indicate the state of an object, such as whether it is known to be marked, whether it is pointing into the relocation set, or whether it is reachable only through a *Finalizer* (Figure 6.10).

[6] www.azul.com/c4-white-paper/

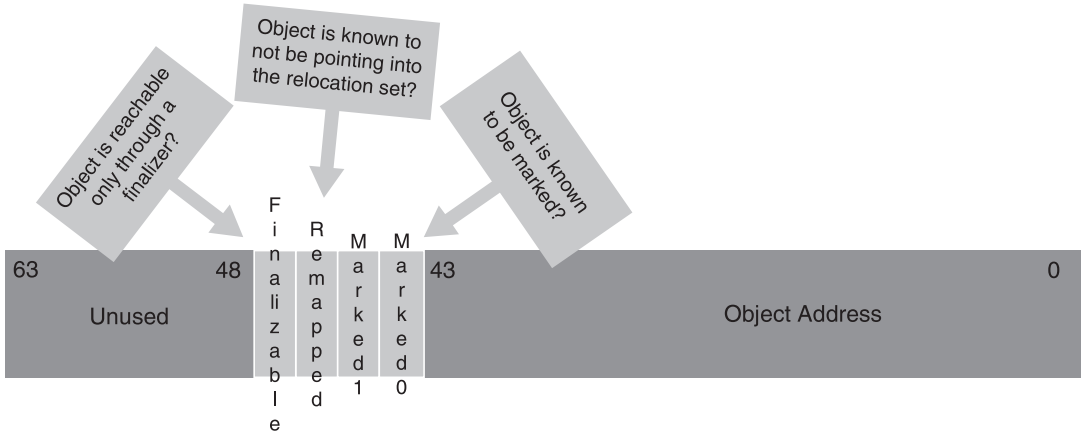

Figure 6.10 Colored Pointers

These tagged pointers allow ZGC to efficiently handle complex tasks such as concurrent marking and relocation without needing to pause the application. This leads to overall shorter GC pause times and improved application performance, making ZGC an effective solution for applications that require low latency and high throughput.

ZGC Core Concepts: Utilizing Thread-Local Handshakes for Improved Efficiency

ZGC is notable for its innovative approach to garbage collection, especially in how it utilizes thread-local handshakes.[7] Thread-local handshakes enable the JVM to execute actions on individual threads without requiring a global STW pause. This approach reduces the impact of STW events and contributes to ZGC's ability to deliver low pause times—a crucial consideration for latency-sensitive applications.

ZGC's adoption of thread-local handshakes is a significant departure from the traditional garbage collection STW techniques (the two are compared in Figure 6.11). In essence, they enable ZGC to adapt and scale the collections, thereby making it suitable for applications with larger heap sizes and stringent latency requirements. However, while ZGC's concurrent processing approach optimizes for lower latency, it could have implications for an application's maximum throughput. ZGC is designed to balance these aspects, ensuring that the throughput degradation remains within acceptable limits.

ZGC Core Concepts: Phases and Concurrent Activities

Like other OpenJDK garbage collectors, ZGC is a tracing collector, but it stands out because of its ability to perform both concurrent marking and compaction. It addresses the challenge of moving live objects from the "from" space to the "to" space within a running application through a set of specialized phases, innovative optimization techniques, and a unique approach to garbage collection.

ZGC refines the concept of a regionalized heap, pioneered by G1, with the introduction of *ZPages*—dynamically sized regions that are allocated and managed according to their memory usage patterns. This flexibility in region sizing is part of ZGC's strategy to optimize memory management and accommodate various allocation rates efficiently.

[7] https://openjdk.org/jeps/312

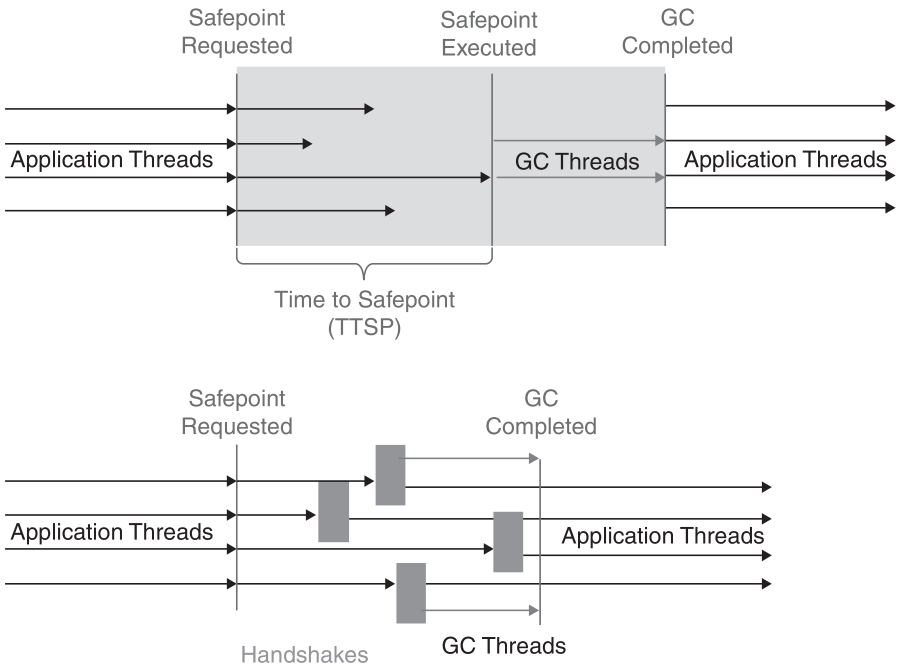

Figure 6.11 Thread-Local Handshakes Versus Global Stop-the-World Events

ZGC also improves on the concurrent marking phase by employing the unique approach of dividing the heap into logical stripes. Each GC thread works on its own stripe, thus reducing contention and synchronization overhead. This design allows ZGC to perform concurrent, parallel processing while marking, reducing phase times and improving overall application performance.

Figure 6.12 provides a visual representation of the ZGC's heap organization into logical stripes. The stripes are shown as separate segments within the heap space, numbered from 0 to *n*. Corresponding to each stripe is a GC thread—"GC Thread 0" through "GC Thread *n*"—tasked with garbage collection duties for its assigned segment.

Let's now look at the phases and concurrent activities of ZGC:

1. **STW Initial Mark (Pause Mark Start):** ZGC initiates the concurrent mark cycle with a brief pause to mark root objects. Recent optimizations, such as moving thread-stack scanning to the concurrent phase, have reduced the duration of this STW phase.

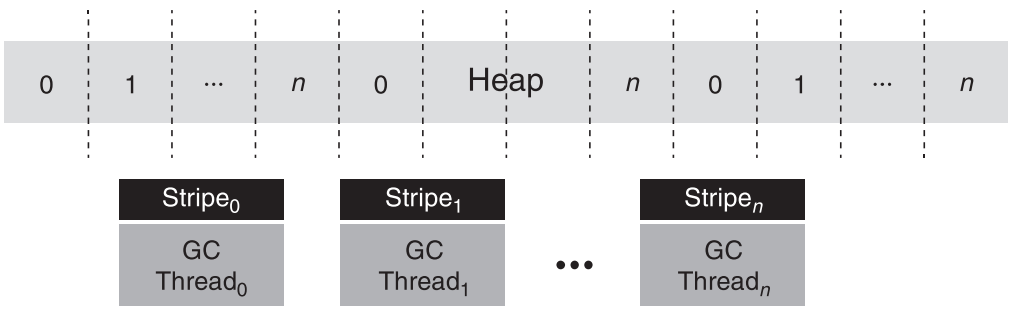

Figure 6.12 ZGC Logical Stripes

2. **Concurrent Mark/Remap:** During this phase, ZGC traces the application's live object graph and marks non-root objects. It employs a load barrier to assist in loading unmarked object pointers. Concurrent reference processing is also carried out, and thread-local handshakes are used.
3. **STW Remark (Pause Mark End):** ZGC executes a quick pause to complete the marking of objects that were in-flight during the concurrent phase. The duration of this phase is tightly controlled to minimize pause times. As of JDK 16, the amount of mark work is capped at 200 microseconds. If marking work exceeds this time limit, ZGC will continue marking concurrently, and another "pause mark end" activity will be attempted until all work is complete.
4. **Concurrent Prepare for Relocation:** During this phase, ZGC identifies the regions that will form the collection/relocation set. It also processes weak roots and non-strong references and performs class unloading.
5. **STW Relocate (Pause Relocate Start):** In this phase, ZGC finishes unloading any leftover metadata and *nmethods*.[8] It then prepares for object relocation by setting up the necessary data structures, including the use of a remap mask. This mask is part of ZGC's pointer forwarding mechanism, which ensures that any references to relocated objects are updated to point to their new locations. Application threads are briefly paused to capture roots into the collection/relocation set.
6. **Concurrent Relocate:** ZGC concurrently performs the actual work of relocating live objects from the "from" space to the "to" space. This phase can employ a technique called *self-healing*, in which the application threads themselves assist in the relocation process when they encounter an object that needs to be relocated. This approach reduces the amount of work that needs to be done during the STW pause, leading to shorter GC pause times and improved application performance.

Figure 6.13 depicts the interplay between Java application threads and ZGC background threads during the garbage collection process. The broad horizontal arrow represents the continuum of application execution over time, with Java application threads running continuously. The vertical bars symbolize STW pauses, where application threads are briefly halted to allow certain GC activities to take place.

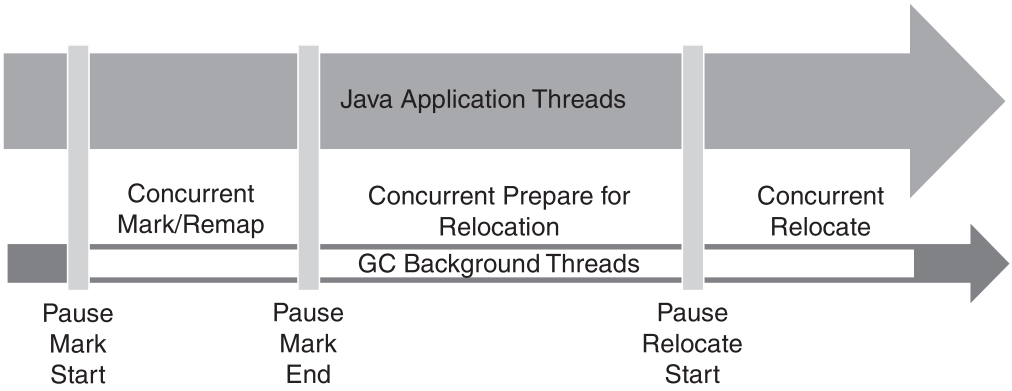

Figure 6.13 ZGC Phases and Concurrent Activities

[8] https://openjdk.org/groups/hotspot/docs/HotSpotGlossary.html