For instance, to set up a benchmark with 5 warmup iterations, 10 measurement iterations, 2 forks, and with thread-local state scope, you would use the following code:

```
@State(Scope.Thread)
@BenchmarkMode(Mode.Throughput)
@OutputTimeUnit(TimeUnit.SECONDS)
@Warmup(iterations = 5)
@Measurement(iterations = 10)
@Fork(2)
public class MyBenchmark {
    // ...
}
```

JVM Benchmarking with JMH

To demonstrate how JMH can be effectively employed for benchmarking, let's consider a practical use case, where we benchmark different processor options. For instance, Arm v8.1+ can use Large System Extensions (LSE)—a set of atomic instructions explicitly formulated to enhance the performance of multithreaded applications by providing more efficient atomic operations. LSE instructions can be used to construct a variety of synchronization primitives, including locks and atomic variables.

The focal point of our benchmarking exercise is the atomic operation known as *compare-and-swap* (*CAS*), where we aim to identify both the maximum and the minimum overhead. *CAS* operations can dramatically influence the performance of multithreaded applications, so a thorough understanding of their performance characteristics is essential for developers of such applications. *CAS*, an essential element in concurrent programming, compares the contents of a memory location to a given value; it then modifies the contents of that memory location to a new given value only if the comparison is true. This is done in a single, atomic step that prevents other threads from changing the memory location in the middle of the operation.

The expected results of the benchmarking process would be to understand the performance characteristics of the *CAS* operation on different processor architectures. Specifically, the benchmark aims to find the maximum and minimum overhead for the *CAS* operation on Arm v8.1+ with and without the use of LSE. The outcomes of this benchmarking process would be valuable in several ways:

- It would provide insights into the performance benefits of using LSE instructions on Arm v8.1+ for multithreaded applications.
- It would help developers make informed decisions when designing and optimizing multithreaded applications especially when coupled with scalability studies.
- It could potentially influence future hardware design decisions by highlighting the performance impacts of LSE instructions.

Remember, the goal of benchmarking is not just to gather performance data but to use that data to inform decisions and drive improvements in both software and hardware design.

In our benchmarking scenario, we will use the `AtomicInteger` class from the `java.util.concurrent` package. The benchmark features two methods: `testAtomicIntegerAlways` and `testAtomicIntegerNever`. The former consistently swaps in a value; the latter never commits a change and simply retrieves the existing value. This benchmarking exercise aims to illustrate how processors and locking primitives grappling with scalability issues can potentially reap substantial benefits from the new LSE instructions.

In the context of Arm processors, LSE provides more efficient atomic operations. When LSE is enabled, Arm processors demonstrate more efficient scaling with minimal overhead—an outcome substantiated by extreme use cases. In the absence of LSE, Arm processors revert to *load-link/store-conditional* (*LL/SC*) operations, specifically using *exclusive load-acquire* (`LDAXR`) and *store-release* (`STLXR`) instructions for atomic *read-modify-write* operations. Atomic *read-modify-write* operations, such as *CAS*, form the backbone of many synchronization primitives. They involve reading a value from memory, comparing it with an expected value, possibly modifying it, and then writing it back to memory—all done atomically.

In the *LL/SC* model, the `LDAXR` instruction reads a value from memory and marks the location as reserved. The subsequent `STLXR` instruction attempts to write a new value back into memory only if no other operation has written to that memory location since the `LDAXR` operation.

However, *LL/SC* operations can experience performance bottlenecks due to potential contention and retries when multiple threads concurrently access the same memory location. This challenge is effectively addressed by the LSE instructions, which are designed to improve the performance of atomic operations and hence enhance the overall scalability of multithreaded applications.

Here's the relevant code snippet from the OpenJDK JMH micro-benchmarks:

```
package org.openjdk.bench.java.util.concurrent;

import org.openjdk.jmh.annotations.*;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@State(Scope.Benchmark)
public class Atomic {
    public AtomicInteger aInteger;

    @Setup(Level.Iteration)
    public void setupIteration() {
        aInteger = new AtomicInteger(0);
    }

    /** Always swap in value. This test should be compiled into a CAS */
    @Benchmark
    @OperationsPerInvocation(2)
```

```
    public void testAtomicIntegerAlways(Blackhole bh) {
        bh.consume(aInteger.compareAndSet(0, 2));
        bh.consume(aInteger.compareAndSet(2, 0));
    }

    /** Never write a value just return the old one. This test should be compiled into a CAS */
    @Benchmark
    public void testAtomicIntegerNever(Blackhole bh) {
        bh.consume(aInteger.compareAndSet(1, 3));
    }
}
```

In this benchmark, the `compareAndSet` method of the `AtomicInteger` class is a *CAS* operation when LSE is enabled. It compares the current value of the `AtomicInteger` to an expected value, and if they are the same, it sets the `AtomicInteger` to a new value. The `testAtomicIntegerAlways` method performs two operations per invocation: It swaps the value of `aInteger` from 0 to 2, and then from 2 back to 0. The `testAtomicIntegerNever` method attempts to swap the value of `aInteger` from 1 to 3, but since the initial value is 0, the swap never occurs.

Profiling JMH Benchmarks with perfasm

The `perfasm` profiler provides a detailed view of your benchmark at the assembly level. This becomes imperative when you're trying to understand the low-level performance characteristics of your code.

As an example, let's consider the `testAtomicIntegerAlways` and `testAtomicIntegerNever` benchmarks we discussed earlier. If we run these benchmarks with the `perfasm` profiler, we can see the assembly instructions that implement these *CAS* operations. On an Arm v8.1+ processor, these will be LSE instructions. By comparing the performance of these instructions on different processors, we can gain insights into how different hardware architectures can impact the performance of atomic operations.

To use the `perfasm` profiler, you can include the `-prof perfasm` option when running your benchmark. For example:

```
$ java -jar benchmarks.jar -prof perfasm
```

The output from the `perfasm` profiler will include a list of the most frequently executed assembly instructions during your benchmark. This sheds light on the way your code is being executed at the hardware level and can spotlight potential bottlenecks or areas for optimization.

By better comprehending the low-level performance characteristics of your code, you will be able to make more-informed decisions about your optimization focal points. Consequently, you are more likely to achieve efficient code, which in turn leads to enhanced overall performance of your applications.

Conclusion

JVM performance engineering is a complex yet indispensable discipline for crafting efficient Java applications. Through the use of robust tools like JMH alongwith its integrated profilers, and the application of rigorous methodologies—monitoring, profiling, analysis, and tuning—we can significantly improve the performance of our applications. These resources allow us to delve into the intricacies of our code, providing valuable insights into its performance characteristics and helping us identify areas that are ripe for optimization. Even seemingly small changes, like the use of `volatile` variables or the choice of atomic operations, can have significant impacts on performance. Moreover, understanding the interactions between the JVM and the underlying hardware, including the role of memory barriers, can further enhance our optimization efforts.

As we move forward, I encourage you to apply the principles, tools, and techniques discussed in this chapter to your own projects. Performance optimization is an ongoing journey and there is always room for refinement and improvement. Continue exploring this fascinating area of software development and see the difference that JVM performance engineering can make in your applications.

This page intentionally left blank

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

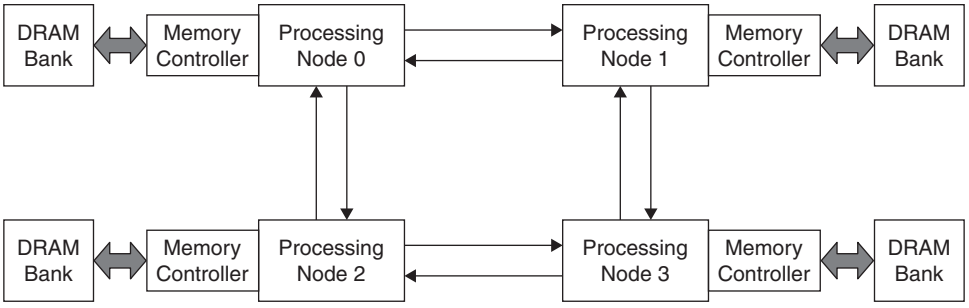

Figure 6.1 NUMA Nodes with Processors, Memory Controllers, and DRAM Banks

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
```

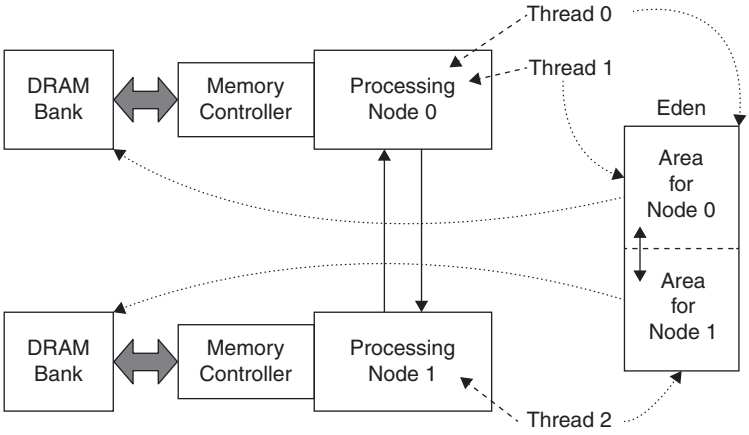

Figure 6.2 The Inner Workings of a NUMA-Aware GC

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.

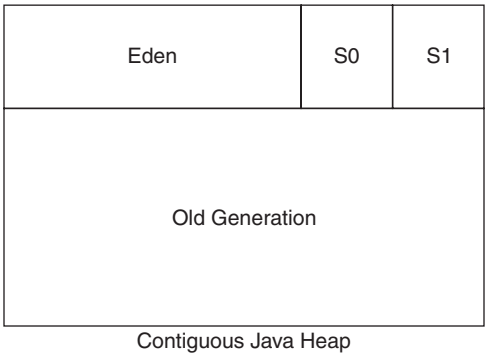

Figure 6.3 A Traditional Java Heap Layout

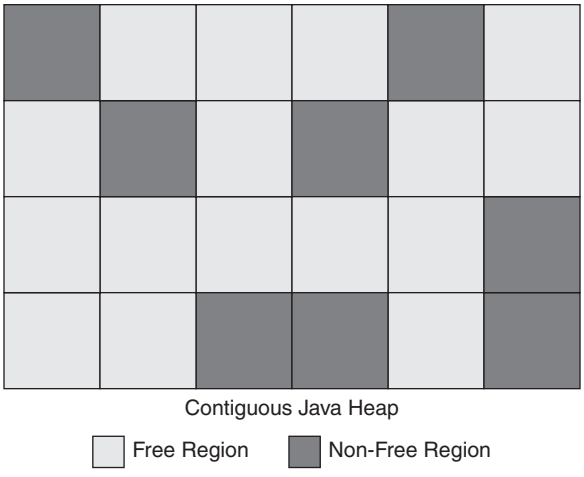

Figure 6.4 A Regionalized Java Heap Layout

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])

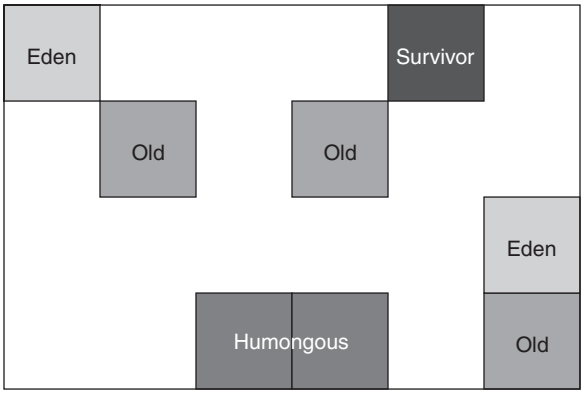

Figure 6.5 A Regionalized Heap Showing Old, Humongous, and Young Regions

[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).

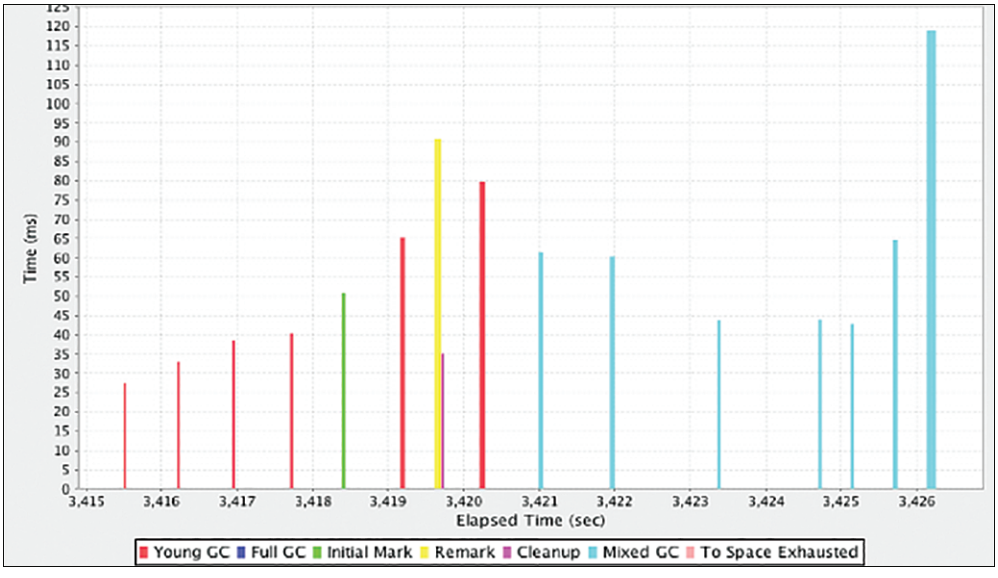

Figure 6.6 G1 Garage Collector's Pause-Time Histogram

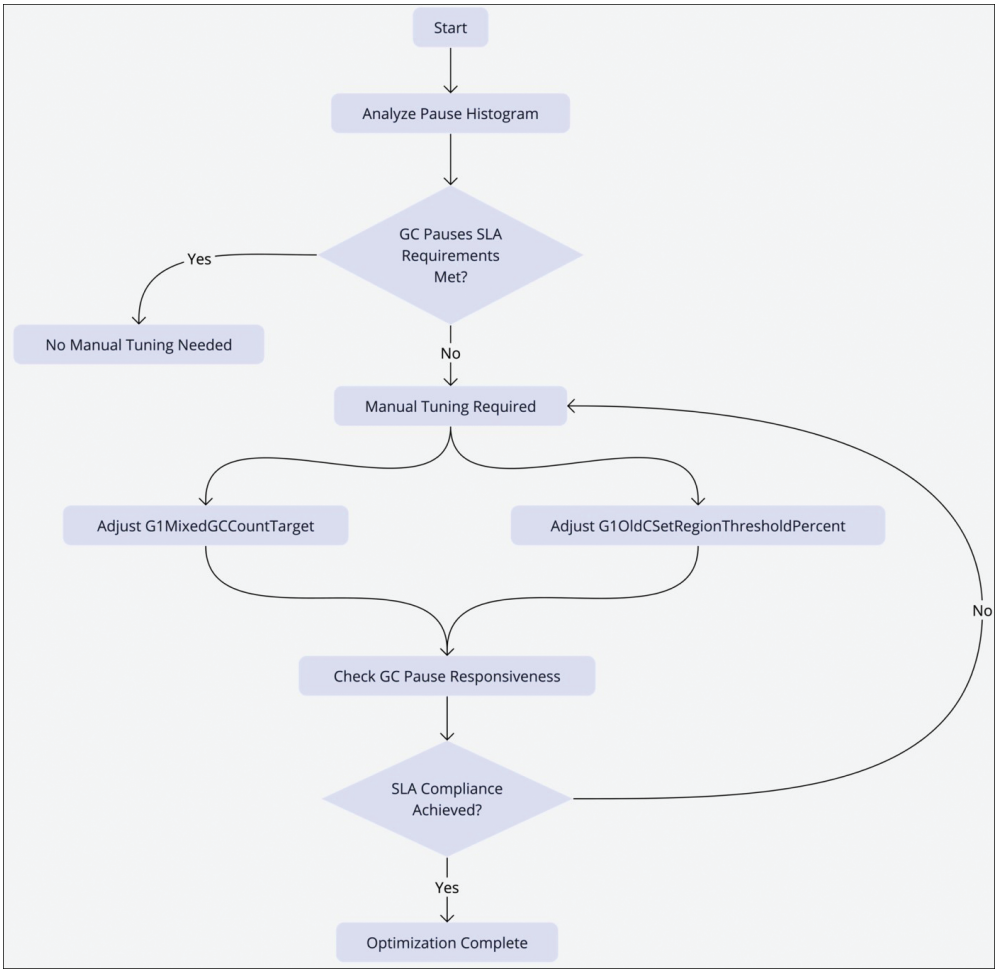

Figure 6.7 Optimizing G1 Responsiveness

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.

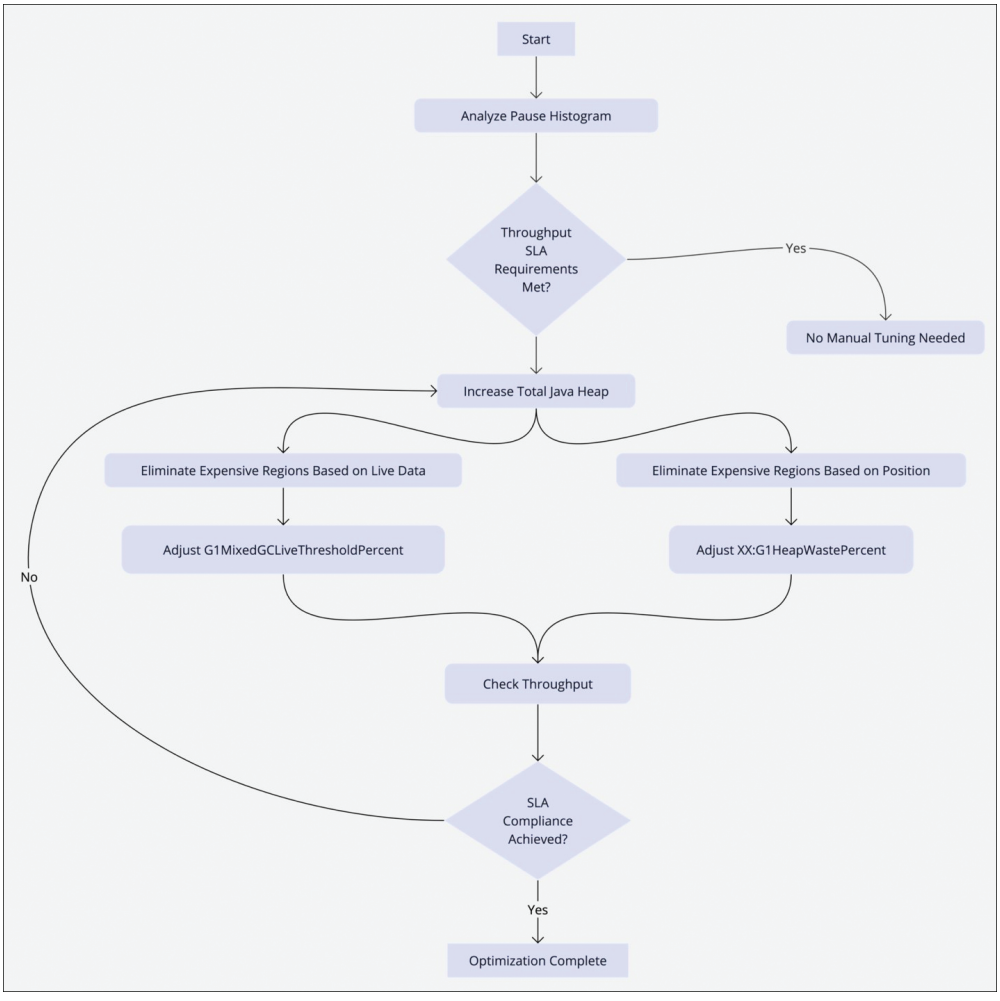

Figure 6.8 Optimizing G1's Throughput

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.

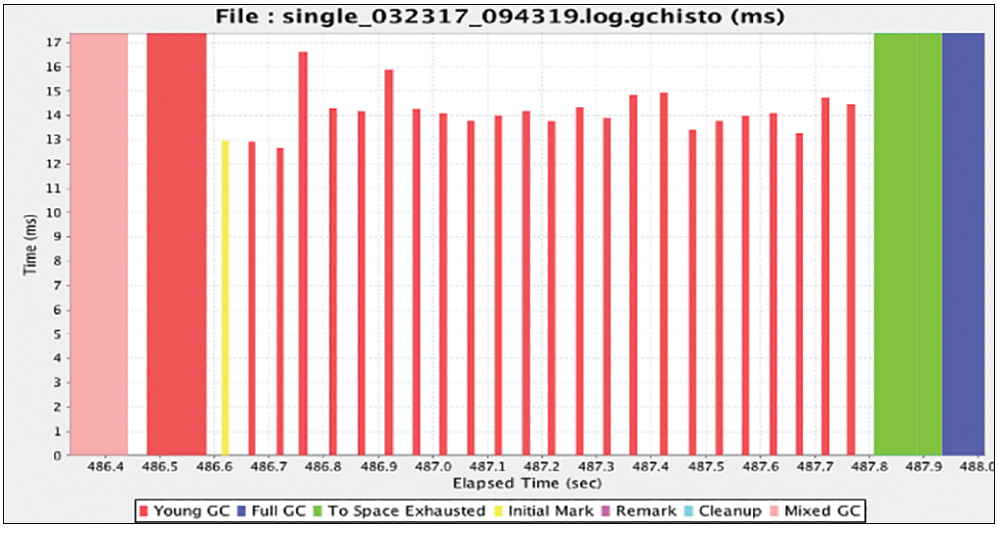

Figure 6.9 A G1 Garbage Collector Pause Timeline Showing Incomplete Marking Cycle

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/

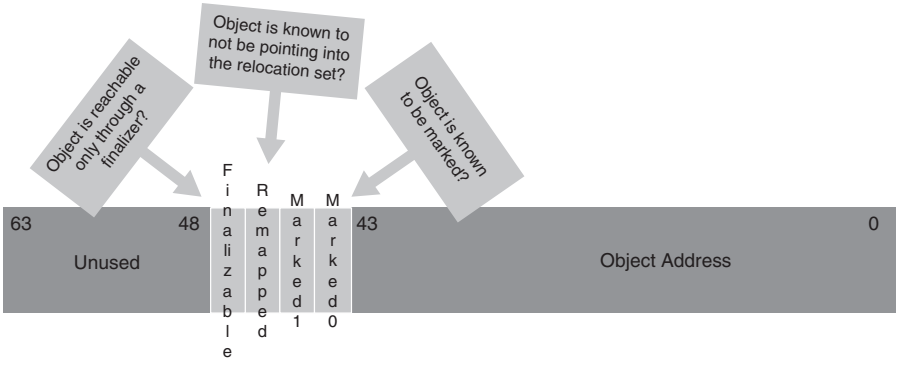

Figure 6.10 Colored Pointers

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

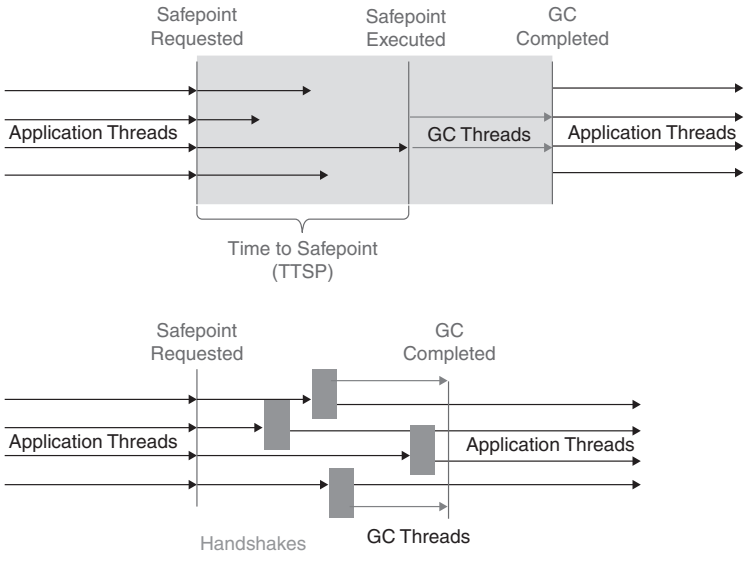

Figure 6.11 Thread-Local Handshakes Versus Global Stop-the-World Events

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.

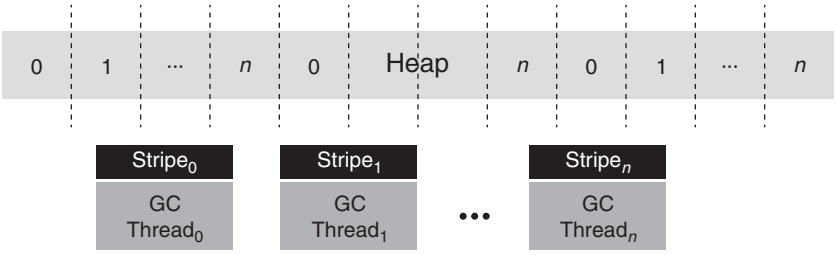

Figure 6.12 ZGC Logical Stripes

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.

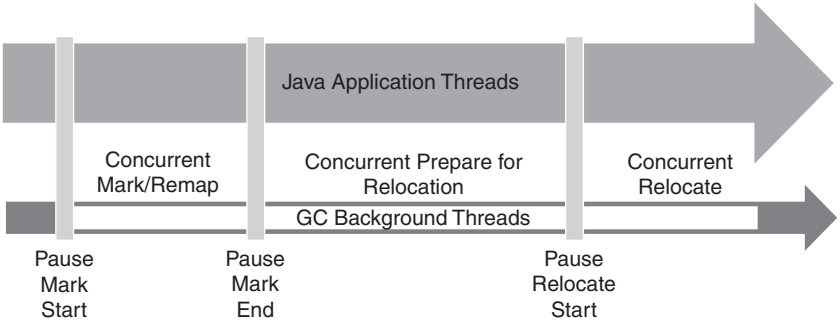

Figure 6.13 ZGC Phases and Concurrent Activities

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

Beneath the application thread timeline is a dotted line representing the ongoing activity of GC background threads. These threads run concurrently with the application threads, performing garbage collection tasks such as marking, remapping, and relocating objects.

In Figure 6.13, the STW pauses are short and infrequent, reflecting ZGC's design goal to minimize their impact on application latency. The periods between these pauses represent phases of concurrent garbage collection activity where ZGC operates alongside the running application without interrupting it.

ZGC Core Concepts: Off-Heap Forwarding Tables

ZGC introduces off-heap forwarding tables—that is, data structures that store the new locations of relocated objects. By allocating these tables outside the heap space, ZGC ensures that the memory used by the forwarding tables does not impact the available heap space for the application. This allows for the immediate return and reuse of both virtual and physical memory, leading to more efficient memory usage. Furthermore, off-heap forwarding tables enable ZGC to handle larger heap sizes, as the size of the forwarding tables is not constrained by the size of the heap.

Figure 6.14 is a diagrammatic representation of the off-heap forwarding tables. In this figure,

- Live objects reside in heap memory.
- These live objects are relocated, and their new locations are recorded in forwarding tables.
- The forwarding tables are allocated in off-heap memory.
- The relocated objects, now in their new locations, can be accessed by application threads.

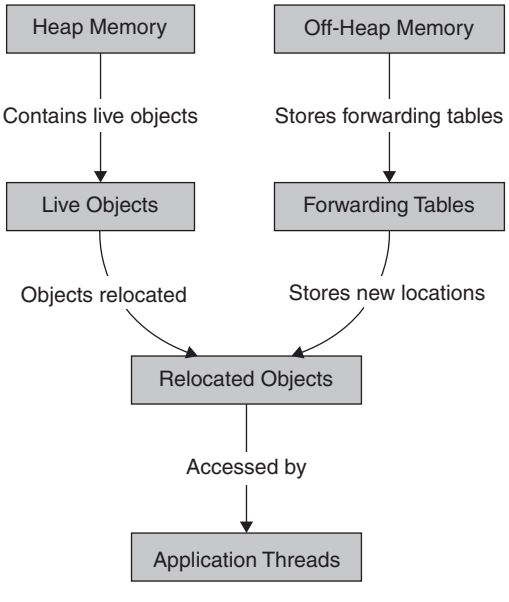

Figure 6.14 ZGC's Off-Heap Forwarding Tables

ZGC Core Concepts: *ZPages*

A *ZPage* is a contiguous chunk of memory within the heap, designed to optimize memory management and the GC process. It can have three different dimensions—small, medium, and large.[9] The exact sizes of these pages can vary based on the JVM configuration and the platform on which it's running. Here's a brief overview:

- **Small pages:** These are typically used for small object allocations. Multiple small pages can exist, and each page can be managed independently.
- **Medium pages:** These are larger than small pages and are used for medium-sized object allocations.
- **Large pages:** These are reserved for large object allocations. Large pages are typically used for objects that are too big to fit into small or medium pages. An object allocated in a large page occupies the entire page.

Figures 6.15 and 6.16 show the usage of *ZPages* in a benchmark, highlighting the frequency and volume of each page type. Figure 6.15 shows a predominance of small pages and Figure 6.16 is a zoomed-in version of the same graph to focus on the medium and large page types.

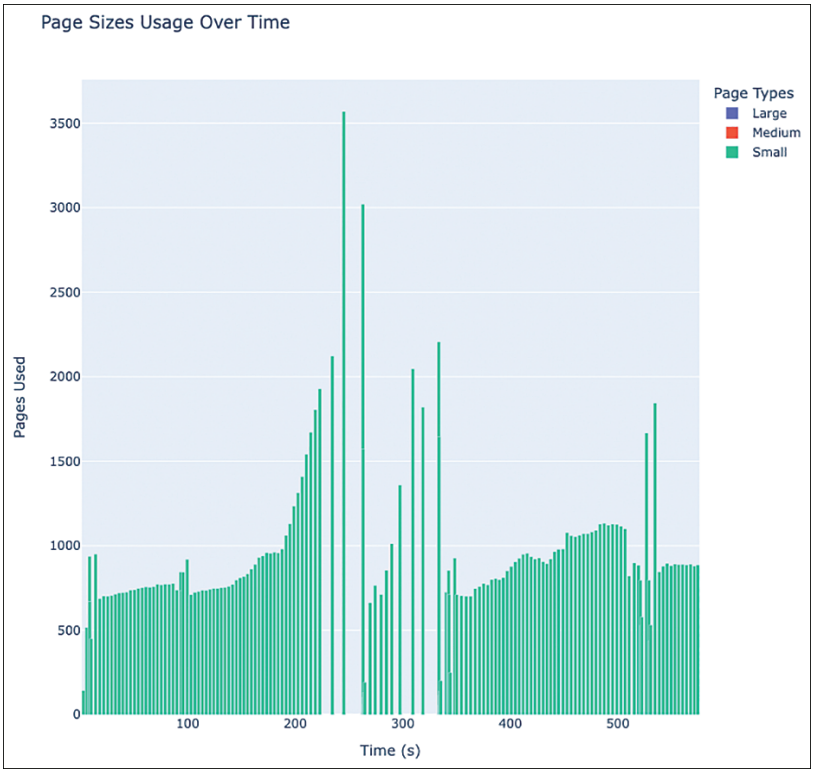

Figure 6.15 Page Sizes Usage Over Time

[9] https://malloc.se/blog/zgc-jdk14

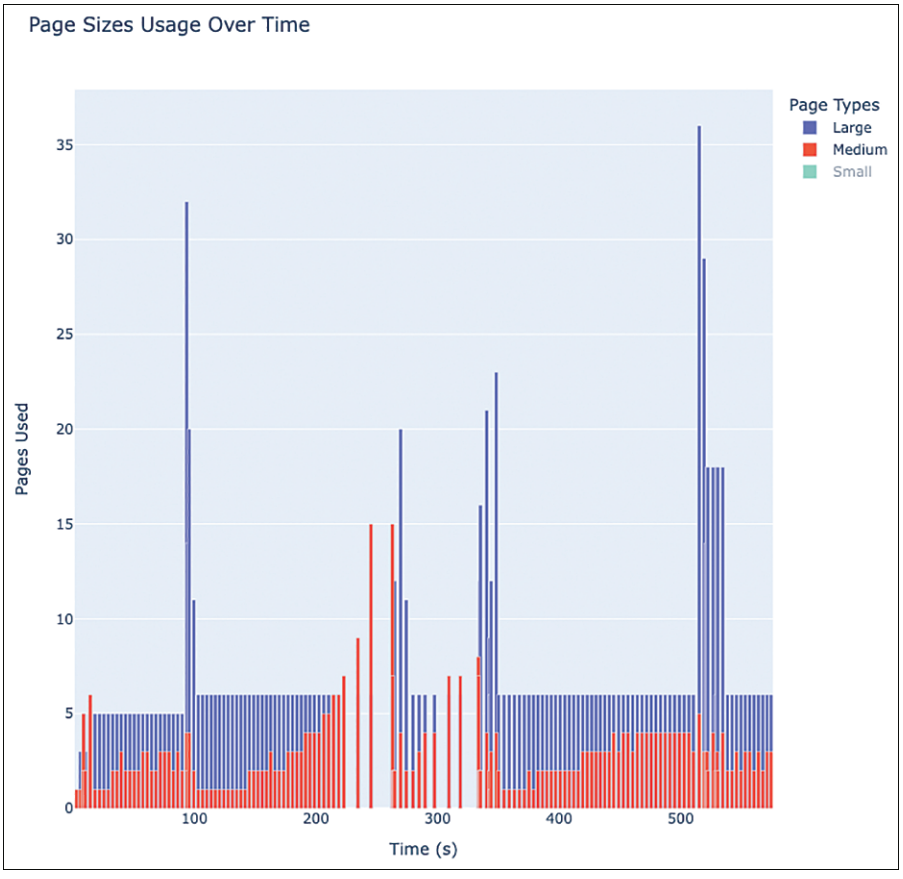

Figure 6.16 Medium and Large ZPages Over Time

Purpose and Functionality

ZGC uses pages to facilitate its multithreaded, concurrent, and region-based approach to garbage collection.

- **Region spanning:** A single *ZPage* (for example, a large *ZPage*) can cover numerous regions. When ZGC allocates a *ZPage*, it marks the corresponding regions that the *ZPage* spans as being used by that *ZPage*.
- **Relocation:** During the relocation phase, live objects are moved from one *ZPage* to another. The regions that these objects previously occupied in the source *ZPage* are then made available for future allocations.
- **Concurrency:** *ZPages* enable ZGC to concurrently operate on distinct pages, enhancing scalability and minimizing pause durations.

- **Dynamic nature:** While the size of regions is fixed, *ZPages* are dynamic. They can span multiple regions based on the size of the allocation request.

Allocation and Relocation

The *ZPage* size is chosen based on the allocation need and can be a small, medium, or large page. The chosen *ZPage* will then cover the necessary number of fixed-size regions to accommodate its size. During the GC cycle, live objects might be relocated from one page to another. The concept of pages allows ZGC to move objects around efficiently and reclaim entire pages when they become empty.

Advantages of ZPages

Using pages provides several benefits:

- **Concurrent relocation:** Since objects are grouped into pages, ZGC can relocate the contents of entire pages concurrently.
- **Efficient memory management:** Empty pages can be quickly reclaimed, and memory fragmentation issues are reduced.
- **Scalability:** The multithreaded nature of ZGC can efficiently handle multiple pages, making it scalable for applications with large heaps.
- **Page metadata:** Each *ZPage* will have associated metadata that keeps track of information such as its size, occupancy, state (whether it's used for object allocation or relocation), and more.

In essence, *ZPages* and regions are fundamental concepts in ZGC that work hand in hand. While *ZPages* provide the actual memory areas for object storage, regions offer a mechanism for ZGC to efficiently track, allocate, and reclaim memory.

ZGC Adaptive Optimization: Triggering a ZGC Cycle

In modern applications, memory management has become more dynamic, which in turn makes it crucial to have a GC that can adapt in real time. ZGC stands as a testament to this need, offering a range of adaptive triggers to initiate its garbage collection cycles. These triggers, which may include a timer, warm-up, high allocation rates, allocation stall, proactive, and high utilization, are designed to cater to the dynamic memory demands of contemporary applications. For application developers and system administrators, understanding these triggers is not just beneficial—it's essential. Grasping their nuances can lead to better management and tuning of ZGC, ensuring that applications consistently meet their performance and latency benchmarks.

Timer-Based Garbage Collection

ZGC can be configured to trigger a GC cycle based on a timer. This timer-based approach ensures that garbage collection occurs at predictable intervals, offering a level of consistency in memory management.

- **Mechanism:** When the specified timer duration elapses and if no other garbage collection has been initiated in the meantime, a ZGC cycle is automatically triggered.

- **Usage:**
  - **Setting the timer:** The interval for this timer can be specified using the JVM option `-XX:ZCollectionInterval=<value-in-seconds>`.
    - **`-XX:ZCollectionInterval=0.01:`** Configures ZGC to check for garbage collection every 10 milliseconds.
    - **`-XX:ZCollectionInterval=3:`** Sets the interval to 3 seconds.
  - **Default behavior:** By default, the value of `ZCollectionInterval` is set to 0. This means that the timer-based triggering mechanism is turned off, and ZGC will not initiate garbage collection based on time alone.
- **Example logs:** When ZGC operates based on this timer, the logs would resemble the following output:

```
[21.374s][info][gc,start    ] GC(7) Garbage Collection (Timer)
[21.374s][info][gc,task     ] GC(7) Using 4 workers
…
[22.031s][info][gc          ] GC(7) Garbage Collection (Timer) 1480M(14%)->1434M(14%)
```

  These logs indicate that a GC cycle was triggered due to the timer, and they provide details about the memory reclaimed during this cycle.
- **Considerations:** The timer-based approach, while offering predictability, should be used judiciously. It's essential to understand ZGC's adaptive triggers. Using a timer-based collection should ideally be a supplementary strategy—a periodic "cleanse" to bring ZGC to a known state. This approach can be especially beneficial for tasks like refreshing classes or processing references, as ZGC cycles can handle concurrent unloading and reference processing.

Warm-up–Based GC

A ZGC cycle can be triggered during the warm-up phase of an application. This is especially beneficial during the ramp-up phase of the application when it hasn't reached its full operational load. The decision to trigger a ZGC cycle during this phase is influenced by the heap occupancy.

- **Mechanism:** During the application's ramp-up phase, ZGC monitors heap occupancy. When it reaches the thresholds of 10%, 20%, or 30%, ZGC triggers a *warm-up* cycle. This can be seen as priming the GC, preparing it for the application's typical memory patterns. If heap usage surpasses one of the thresholds and no other garbage collection has been initiated, a ZGC cycle is triggered. This mechanism allows the system to gather early samples of the GC duration, which can be used by other rules.
- **Usage:** ZGC's threshold, `soft_max_capacity`, determines the maximum heap occupancy it aims to maintain.[10] For the warm-up–based GC cycle, ZGC checks heap occupancy at 10%, 20%, and 30% of the `soft_max_capacity`, provided no other garbage

[10] https://malloc.se/blog/zgc-softmaxheapsize

collections have been initiated. To influence these warm-up cycles, end users can adjust the `-XX:SoftMaxHeapSize=<size>` command-line option.

- **Example logs:** When ZGC operates under this rule, the logs would resemble the following output:

```
[7.171s][info][gc,start    ] GC(2) Garbage Collection (Warmup)
[7.171s][info][gc,task     ] GC(2) Using 4 workers
…
[8.842s][info][gc          ] GC(2) Garbage Collection (Warmup) 2078M(20%)->1448M(14%)
```

  From the logs, it's evident that a warm-up cycle was triggered when the heap occupancy reached 20%.

- **Considerations:** The warm-up/ramp-up phases are crucial for applications, especially those that build extensive caches during start-up. These caches are often used heavily in subsequent transactions. By initiating a ZGC cycle during warm-up, the application can ensure that the caches are efficiently built and optimized for future use. This ensures a smoother transition to consistent performance as the application transitions from the warm-up phase to handling real-world transactions.

GC Based on High Allocation Rates

When an application allocates memory at a high rate, ZGC can initiate a GC cycle to free up memory. This prevents the application from exhausting the heap space.

- **Mechanism:** ZGC continuously monitors the rate of memory allocation. Based on the observed allocation rate and the available free memory, ZGC estimates the time until the application will run out of memory (OOM). If this estimated time is less than the expected duration of a GC cycle, ZGC triggers a cycle.

  ZGC can operate in two modes based on the `UseDynamicNumberOfGCThreads` setting:

  - **Dynamic GC workers:** In this mode, ZGC dynamically calculates the number of GC workers needed to avoid OOM. It considers factors such as the current allocation rate, variance in the allocation rate, and the average time taken for serial and parallel phases of the GC.
  - **Static GC workers:** In this mode, ZGC uses a fixed number of GC workers (`ConcGCThreads`). It estimates the time until OOM based on the moving average of the sampled allocation rate, adjusted with a safety margin for unforeseen allocation spikes. The safety margin ("headroom") is calculated as the space required for all worker threads to allocate a small ZPage and for all workers to share a single medium ZPage.

- **Usage:** Adjusting the `UseDynamicNumberOfGCThreads` setting and the `-XX:SoftMaxHeapSize=<size>` command-line option, and possibly adjusting the `-XX:ConGCThreads=<number>`, allows fine-tuning of the garbage collection behavior to suit specific application needs.

- **Example logs:** The following logs provide insights into the triggering of the GC cycle due to high allocation rates:

```
[221.876s][info][gc,start    ] GC(12) Garbage Collection (Allocation Rate)
[221.876s][info][gc,task     ] GC(12) Using 3 workers
```

```
…
[224.073s][info][gc          ] GC(12) Garbage Collection (Allocation Rate)
5778M(56%)->2814M(27%)
```

From the logs, it's evident that a GC was initiated due to a high allocation rate, and it reduced the heap occupancy from 56% to 27%.

GC Based on Allocation Stall

When the application is unable to allocate memory due to insufficient space, ZGC will trigger a cycle to free up memory. This mechanism ensures that the application doesn't stall or crash due to memory exhaustion.

- **Mechanism:** ZGC continuously monitors memory allocation attempts. If an allocation request cannot be satisfied because the heap doesn't have enough free memory and an allocation stall has been observed since the last GC cycle, ZGC initiates a GC cycle. This is a reactive approach, ensuring that memory is made available promptly to prevent application stalls.
- **Usage:** This trigger is inherent to ZGC's design and doesn't require explicit configuration. However, monitoring and understanding the frequency of such triggers can provide insights into the application's memory usage patterns and potential optimizations.
- **Example logs:**

```
…
[265.354s][info][gc          ] Allocation Stall (<thread-info>) 1554.773ms
[265.354s][info][gc          ] Allocation Stall (<thread-info>) 1550.993ms
…
…
[270.476s][info][gc,start    ] GC(19) Garbage Collection (Allocation Stall)
[270.476s][info][gc,ref      ] GC(19) Clearing All SoftReferences
[270.476s][info][gc,task     ] GC(19) Using 4 workers
…
[275.112s][info][gc          ] GC(19) Garbage Collection (Allocation Stall)
7814M(76%)->2580M(25%)
```

From the logs, it's evident that a GC was initiated due to an allocation stall, and it reduced the heap occupancy from 76% to 25%.

- **Considerations:** Frequent allocation stalls might indicate that the application is nearing its memory limits or has sporadic memory usage spikes. It's imperative to monitor such occurrences and consider increasing the heap size or optimizing the application's memory usage.

GC Based on High Usage

If the heap utilization rate is high, a ZGC cycle can be triggered to free up memory. This is the first step before triggering ZGC cycles due to high allocation rates. The GC is triggered as a preventive measure if the heap is at 95% occupancy and based on your application's allocation rates.

- **Mechanism:** ZGC continuously monitors the amount of free memory available. If the free memory drops to 5% or less of the heap's total capacity, a ZGC cycle is triggered.

This is especially useful in scenarios where the application has a very low allocation rate, such that the allocation rate rule doesn't trigger a GC cycle, but the free memory is still decreasing steadily.

- **Usage:** This trigger is inherent to ZGC's design and doesn't require explicit configuration. However, understanding this mechanism can help in tuning the heap size and optimizing memory usage.
- **Example logs:**

```
[388.683s][info][gc,start    ] GC(36) Garbage Collection (High Usage)
[388.683s][info][gc,task     ] GC(36) Using 4 workers
…
[396.071s][info][gc          ] GC(36) Garbage Collection (High Usage) 9698M(95%)->2726M(27%)
```

- **Considerations:** If the high usage–based GC is triggered frequently, it might indicate that the heap size is not adequate for the application's needs. Monitoring the frequency of such triggers and the heap's occupancy can provide insights into potential optimizations or the need to adjust the heap size.

Proactive GC

ZGC can proactively initiate a GC cycle based on heuristics or predictive models. This approach is designed to anticipate the need for a GC cycle, allowing the JVM to maintain smaller heap sizes and ensure smoother application performance.

- **Mechanism:** ZGC uses a set of rules to determine whether a proactive GC should be initiated. These rules consider factors such as the growth of heap usage since the last GC, the time elapsed since the last GC, and the potential impact on application throughput. The goal is to perform a GC cycle when it's deemed beneficial, even if the heap still has a significant amount of free space.
- **Usage:**
  - **Enabling/disabling proactive GC:** Proactive garbage collection can be enabled or disabled using the JVM option `ZProactive`.
    - **`-XX:+ZProactive`:** Enables proactive garbage collection.
    - **`-XX:-ZProactive`:** Disables proactive garbage collection.
  - **Default behavior:** By default, proactive garbage collection in ZGC is *enabled* (`-XX:+ZProactive`). When it is enabled, ZGC will use its heuristics to decide when to initiate a proactive GC cycle, even if there's still a significant amount of free space in the heap.
- **Example logs:**

```
[753.984s][info][gc,start    ] GC(41) Garbage Collection (Proactive)
[753.984s][info][gc,task     ] GC(41) Using 4 workers
…
[755.897s][info][gc          ] GC(41) Garbage Collection (Proactive) 5458M(53%)->1724M(17%)
```

From the logs, it's evident that a proactive GC was initiated, which reduced the heap occupancy from 53% to 17%.

- **Considerations:** The proactive GC mechanism is designed to optimize application performance by anticipating memory needs. However, it's vital to monitor its behavior to ensure it aligns with the application's requirements. Frequent proactive GCs might indicate that the JVM's heuristics are too aggressive, or the application has unpredictable memory usage patterns.

The depth and breadth of ZGC's adaptive triggers underscore its versatility in managing memory for diverse applications. These triggers, while generally applicable to most newer garbage collectors, find a unique implementation in ZGC. The specific details and sensitivity of these triggers can vary based on the GC's design and configuration. Although ZGC's default behavior is quite robust, a wealth of optimization potential lies beneath the surface. By delving into the specifics of each trigger and interpreting the associated logs, practitioners and system specialists can fine-tune ZGC's operations to better align them with their application's unique memory patterns. In the fast-paced world of software development, such insights can be the difference between good performance and exceptional performance.

Advancements in ZGC

Since its experimental debut in JDK 11 LTS, ZGC has undergone numerous enhancements and refinements in subsequent JDK releases. Here is an overview of some key improvements to ZGC from JDK 11 to JDK 17:

- **JDK 11: Experimental introduction of ZGC**. ZGC brought innovative concepts such as load barriers and colored pointers to the table, aiming to provide near-real-time, low-latency garbage collection and scalability for large heap sizes.
- **JDK 12: Concurrent class unloading.** Class unloading is the process of removing classes that are no longer needed from memory, thereby freeing up space and making the memory available for other tasks. In traditional GCs, class unloading requires a STW pause, which can lead to longer GC pause times. However, with the introduction of concurrent class unloading in JDK 12, ZGC became able to unload classes while the application is still running, thereby reducing the need for STW pauses and leading to shorter GC pause times.
- **JDK 13: Uncommitting unused memory.** In JDK 13, ZGC was enhanced so that it uncommitted unused memory more rapidly. This improvement led to more efficient memory usage and potentially shorter GC pause times.
- **JDK 14: Windows and macOS support added.** ZGC extended its platform support in JDK 14, becoming available on Windows and macOS. This broadened the applicability of ZGC to a wider range of systems.

- **JDK 15: Production readiness, compressed class pointers, and NUMA-awareness.** By JDK 15, ZGC had matured to a production-ready state, offering a robust low-latency garbage collection solution for applications with large heap sizes. Additionally, ZGC became NUMA-aware, considering memory placement on multi-socket systems with the NUMA architecture. This led to improved GC pause times and overall performance.
- **JDK 16: Concurrent thread stack processing.** JDK 16 introduced concurrent thread stack processing to ZGC, further reducing the work done during the remark pause and leading to even shorter GC pause times.
- **JDK 17: Dynamic GC threads, faster terminations, and memory efficiency.** ZGC now dynamically adjusts the number of GC threads for better CPU usage and performance. Additionally, ZGC's ability to abort ongoing GC cycles enables faster JVM termination. There's a reduction in mark stack memory usage, making the algorithm more memory efficient.

Future Trends in Garbage Collection

For runtimes, the memory management unit is the largest "disrupter" in efforts to harmonize operations with the underlying hardware. Specifically:

- The GC not only traverses the live object graph, but also relocates the live objects to achieve (partial) compaction. This relocation can disrupt the state of CPU caches because it moves live objects from different regions into a new region.
- While TLAB bump-the-pointer allocations are efficient, the disruption mainly occurs when the GC starts its work, given the contrasting nature of these two activities.
- Concurrent work, which is becoming increasingly popular with modern GC algorithms, introduces overhead. This is due to maintenance barriers and the challenge of keeping pace with the application. The goal is to achieve pause times in milliseconds or less and ensure graceful degradation in scenarios where the GC might be overwhelmed.

In today's cloud computing era, where data centers are under pressure to be both cost-effective and efficient, power and footprint efficiency are the real needs to be addressed. This is especially true when considering the GC's need to work concurrently and meet the increasing demands of modern applications. Incremental marking and the use of regions are strategies employed to make this concurrent work more efficient.

One of the emerging trends in modern GC is the concept of "tunable work units"—that is, the ability to adjust and fine-tune specific tasks within the GC process to optimize performance and efficiency. By breaking the GC process down into smaller, tunable units, it becomes possible to better align the GC's operations with the application's needs and the underlying hardware capabilities. This granularity allows for more precise control over aspects such as partial compaction, maintenance barriers, and concurrent work, leading to more predictable and efficient garbage collection cycles (Figure 6.17).

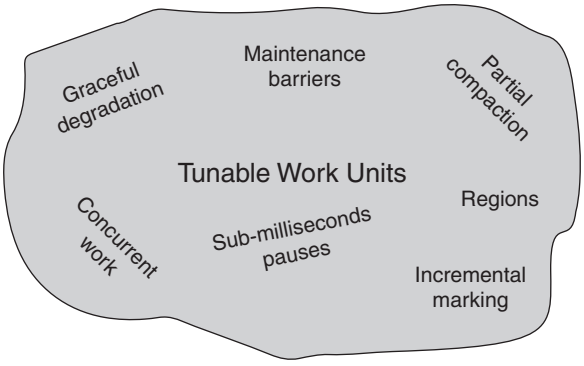

Figure 6.17 A Concurrent Garbage Collector's Tunable Work Units

As systems become more powerful and multi-core processors become the norm, GCs that utilize these cores concurrently will be at an advantage. However, power efficiency, especially in data centers and cloud environments, is crucial. It's not just about scaling with CPU usage, but also about how efficiently that power is used.

GCs need to be in sync with both the hardware on which they operate and the software patterns they manage. Doing so will require addressing the following needs:

- Efficient CPU and system cache utilization
- Minimizing latency for accessing such caches and bringing them into the right states
- Understanding allocation size, patterns, and GC pressures

By being more aware of the hardware and software patterns, and adapting its strategies based on the current state of the system, a GC can aid its intrinsic adaptiveness and can significantly improve both performance and efficiency.

Extrinsic tools such as machine learning can further boost GC performance by analyzing application behavior and memory usage patterns, with machine learning algorithms potentially automating much of the complex and time-consuming process of GC tuning. But it's not just about tuning: It's also about prompting the GC to behave in a particular way by recognizing and matching patterns. This can help avoid potential GC behavior issues before they occur. Such machine learning algorithms could also help in optimizing power usage by learning the most efficient ways to perform garbage collection tasks, thereby reducing the overall CPU usage and power consumption.

The future of garbage collection in the JVM will undoubtedly be dynamic and adaptive, driven by both intrinsic improvements and extrinsic optimizations. As we continue to push the boundaries of what's possible with modern applications, it's clear that our garbage collectors will need to keep pace, evolving and adapting to meet these new challenges head-on.

The field of garbage collection is not static; it evolves alongside the landscape of software development. New GCs are being developed, and the existing ones are being improved to handle the changing needs of applications. For instance,

- Low-pause collectors like ZGC and Shenandoah aim to minimize GC pause times so as to improve application performance.
- Recent JDK releases have featured continuous improvements in existing GCs (for example, G1, ZGC, and Shenandoah) aimed at improving their performance, scalability, and usability. As these GCs continue to evolve, they may become suitable for a wider range of workloads. As of the writing of this book, the enhancements made so far have sown the seeds that will likely turn ZGC and Shenandoah into generational GCs.
- Changes in application development practices, such as the rise of microservices and cloud-native applications, have led to an increased focus on containerization and resource efficiency. This trend will influence improvements to GCs because such applications often have different memory management requirements than the traditional monolithic applications.
- The rise of big data and machine learning applications, which often require handling large data sets in memory, is pushing the boundaries of garbage collection. These applications require GCs that can efficiently manage large heaps and provide good overall throughput.

As these trends continue to unfold, it will be important for architects and performance engineers to stay informed and be ready to adapt their garbage collection strategies as needed. This will lead to further improvements and enhancements to OpenJDK HotSpot VM garbage collectors.

Practical Tips for Evaluating GC Performance

When it comes to GC tuning, my advice is to build upon the methodologies and performance engineering approach detailed in Chapter 5. Start by understanding your application's memory usage patterns. GC logs are your best friends. There are various log parsers and plotters that can help you visualize your application patterns like a lit Christmas tree. Enriching your understanding with GC logs will lay the foundation for effective GC tuning.

And don't underestimate the importance of testing. Testing can help you understand the synergies between your chosen collector and your application's memory demands. Once we establish the synergies and their gaps, we can move to targeted, incremental tuning. Such tuning efforts can have a significant impact on application performance, and their incremental nature will allow you to thoroughly test any changes to ensure they have the desired effect.

Evaluating the performance of different GCs with your workloads is a crucial step in optimizing your application's performance. Here are some practical tips that resonate with the benchmarking strategies and structured approach championed in Chapter 5:

- **Measure GC performance:** Utilize tools like Java's built-in JMX (Java Management Extensions) or VisualVM[11] for intermittent, real-time monitoring of GC performance. These tools can provide valuable metrics such as total GC time, maximum GC pause time, and heap usage. For continuous, low-overhead monitoring, especially in

[11] https://visualvm.github.io/

production, Java Flight Recorder (JFR) is the tool of choice. It provides detailed runtime information with minimal impact on performance. GC logs, by comparison, are ideal for post-analysis, capturing a comprehensive history of GC events. Analyzing these logs can help you identify trends and patterns that might not be apparent from a single snapshot.

- **Understand GC metrics:** It's important to understand what the different GC metrics mean. For instance, a high GC time might indicate that your application is spending a lot of time in garbage collection, which could impact its performance. Similarly, a high heap usage might indicate that your application has a high "GC churn" or it isn't sized appropriately for your live data set. "GC churn" refers to how frequently your application allocates and deallocates memory. High churn rates and under-provisioned heaps can lead to frequent GC cycles. Monitoring these metrics will help you make informed decisions about GC tuning.
- **Interpret metrics in the context of your application:** The impact of GC on your application's performance can vary depending on its specific characteristics. For instance, an application with a high allocation rate might be more affected by GC pause times than an application with a low allocation rate. Therefore, it's important to interpret GC metrics in the context of your application. Consider factors such as your application's workload, transaction patterns, and memory usage patterns when interpreting these metrics. Additionally, correlate these GC metrics with overall application performance indicators such as response time and throughput. This holistic approach ensures that GC tuning efforts are aligned not just with memory management efficiency but also with the application's overall performance objectives.
- **Understand GC triggers and adaptation:** Garbage collection is typically triggered when the eden space is nearing capacity or when a threshold related to the old generation is reached. Modern GC algorithms are adaptive and can adjust their behavior based on the application's memory usage patterns in these different areas of the heap. For instance, if an application has a high allocation rate, the young GC might be triggered more frequently to keep up with the rate of object creation. Similarly, if the old generation is highly utilized, the GC might spend more time compacting the heap to free up space. Understanding these adaptive behaviors can help you tune your GC strategy more effectively.
- **Experiment with different GC algorithms:** Different GC algorithms have different strengths and weaknesses. For instance, the Parallel GC is designed to maximize throughput, while ZGC is designed to minimize pause times. Additionally, the G1 GC offers a balance between throughput and pause times and can be a good choice for applications with a large heap size. Experiment with different GC algorithms to see which one works best with your workload. Remember, the best GC algorithm for your application depends on your specific performance requirements and constraints.
- **Tune GC parameters:** Most GC algorithms offer a range of parameters that you can fine-tune to optimize performance. For instance, you can adjust the heap size or the ratio of young-generation to old-generation space. Be sure to understand what these parameters do before adjusting them, as improper tuning can lead to worse performance. Start with the default settings or follow recommended best practices as a baseline. From there, make incremental adjustments and closely monitor their effects. Consistent with the

experimental design principles outlined in Chapter 5, testing each change in a controlled environment is crucial. Utilizing a staging area that mirrors your production environment allows for a safe assessment of each tuning effort, ensuring no disruptions to your live applications. Keep in mind that GC tuning is inherently an iterative performance engineering process. Finding the optimal configuration often involves a series of careful experiments and observations.

When selecting a GC for your application, it is crucial to assess its performance in the context of your workloads. Remember, the objective is not solely to enhance GC performance, but rather to ensure your application provides the best possible user experience.

Evaluating GC Performance in Various Workloads

Effective memory management is pivotal to the efficient execution of Java applications. As a central aspect of this process, garbage collection must be optimized to deliver both efficiency and adaptability. To achieve this, we will examine the needs and shared characteristics across a variety of open-source frameworks such as Apache Hadoop, Apache Hive, Apache HBase, Apache Cassandra, Apache Spark, and other such projects. In this context, we will delve into different transaction patterns, their relationship with in-memory storage mechanisms, and their effects on heap management, all of which inform our approach to optimizing GC to best suit our application and its transactions, while carefully considering their unique characteristics.

Types of Transactional Workloads

The projects and frameworks can be broadly categorized into three groups based on their transactional interactions with in-memory databases and their influence on the Java heap:

- **Analytics:** Online analytical processing (OLAP)
- **Operational stores:** Online transactional processing (OLTP)
- **Hybrid:** Hybrid transactional and analytical processing (HTAP)

Analytics (OLAP)

This transaction type typically involves complex, long-running queries against large data sets for business intelligence and data mining purposes. The term “stateless” describes these interactions, signifying that each request and response pair is independent, with no information retained between transactions. These workloads are often characterized by high throughput and large heap demands. In the context of garbage collection, the primary concern is the management of transient objects and the high allocation rates, which could lead to a large heap size. Examples of OLAP applications in the open-source world include Apache Hadoop and Spark, which are used for distributed processing of large data sets across clusters of computers.[12] The

[12] www.infoq.com/articles/apache-spark-introduction/

memory management requirements for these kinds of applications often benefit from GCs (for example, G1) that can efficiently handle large heaps and provide good overall throughput.

Operational Stores (OLTP)

This transaction type is usually associated with serving real-time business transactions. OLTP applications are characterized by a large number of short, atomic transactions such as updates to a database. They require fast, low-latency access to small amounts of data in large databases. In these interactive transactions, the state of data is frequently read from or written to; thus, these interactions are "stateful." In the context of Java garbage collection, the focus is on managing the high allocation rates and pressure from medium-lived data or transactions. Examples include NoSQL databases like Apache Cassandra and HBase. For these types of workloads, low-pause collectors such as ZGC and Shenandoah are often a good fit, as they are designed to minimize GC pause times, which can directly impact the latency of transactions.

Hybrid (HTAP)

HTAP applications involve a relatively new type of workload that is a combination of OLAP and OLTP, requiring both analytical processing and transactional processing. Such systems aim to perform real-time analytics on operational data. For instance, this category includes in-memory computing systems such as Apache Ignite, which provide high-performance, integrated, and distributed in-memory computing capabilities for processing OLAP and OLTP workloads. Another example from the open-source world is Apache Flink, which is designed for stream processing but also supports batch processing tasks. Although not a stand-alone HTAP database, Flink complements HTAP architectures with its real-time processing power, especially when integrated with systems that manage transactional data storage. The desired GC traits for such workloads include high throughput for analytical tasks, combined with low latency for real-time transactions, suggesting an enhancement of GCs might be optimal. Ideally, these workloads would use a throughput-optimized GC that minimizes pauses and maintains low latency—optimized generational ZGC, for example.

Synthesis

By understanding the different types of workloads (OLAP, OLTP, HTAP) and their commonalities, we can pinpoint the most efficient GC traits for each. This knowledge aids in optimizing GC performance and adaptiveness, which contributes to the overall performance of Java applications. As the nature of these workloads continues to evolve and new ones emerge, the ongoing study and enhancement of GC strategies will remain crucial to ensure they stay effective and adaptable. In this context, automatic memory management becomes an integral part of optimizing and adapting to diverse workload scenarios in the Java landscape.

The key to optimizing GC performance lies in understanding the nature of the workload and choosing the right GC algorithm or tuning parameters accordingly. For instance, OLAP workloads, which are stateless and characterized by high throughput and large heap demands, may benefit from GCs like optimized G1. OLTP workloads, which are stateful and require fast, low-latency access to data, may be better served by low-pause GCs like generational ZGC or

Shenandoah. HTAP workloads, which combine the characteristics of both OLAP and OLTP, will drive improvements to GC strategies to achieve optimal performance.

Adding to this, the increasing presence of machine learning and other advanced analytical tools in performance tuning suggests that future GC optimization might leverage these technologies to further refine and automate tuning processes. Considering the CPU utilization associated with each GC strategy is also essential because it directly influences not just garbage collection efficiency but the overall application throughput.

By tailoring the garbage collection strategy to the specific needs of the workload, we can ensure that Java applications run efficiently and effectively, delivering the best possible performance for the end user. This approach, combined with the ongoing research and development in the field of GC, will ensure that Java continues to be a robust and reliable platform for a wide range of applications.

Live Data Set Pressure

The "live data set" (LDS) comprises the volume of live data in the heap, which can impact the overall performance of the garbage collection process. The LDS encompasses objects that are still in use by the application and have not yet become eligible for garbage collection.

In traditional GC algorithms, the greater the live data set pressure, the longer the GC pauses can be, as the algorithm has to process more live data. However, this doesn't hold true for all GC algorithms.

Understanding Data Lifespan Patterns

Different types of data contribute to LDS pressure in varying ways, highlighting the importance of understanding your application's data lifespan patterns. Data can be broadly categorized into four types based on their lifespan:

- **Transient data:** These short-lived objects are quickly discarded after use. They usually shouldn't survive past minor GC cycles and are collected in the young generation.
- **Short-lived data:** These objects have a slightly longer lifespan than transient data but are still relatively short-lived. They may survive a few minor GC cycles but are typically collected before being promoted to the old generation.
- **Medium-lived data:** These objects have a lifespan that extends beyond several minor GC cycles and often get promoted to the old generation. They can increase the heap's occupancy and apply pressure on the GC.
- **Long-lived data:** These objects remain in use for a significant portion of the application's lifetime. They reside in the old generation and are collected only during old-generation cycles.

Impact on Different GC Algorithms

G1 is designed to provide more consistent pause times and better performance by incrementally processing the heap in smaller regions and predicting which regions will yield the most garbage. Yet, the volume of live data still impacts its performance. Medium-lived data and short-lived data surviving minor GC cycles can increase the heap's occupancy and apply unnecessary pressure, thereby affecting G1's performance.

ZGC, by comparison, is designed to have pause times that are largely independent of the size of the heap or the LDS. ZGC maintains low latency by performing most of its work concurrently without stopping application threads. However, under high memory pressure, it employs load-shedding strategies, such as throttling the allocation rate, to maintain its low-pause guarantees. Thus, while ZGC is designed to handle larger LDS more gracefully, extreme memory pressures can still affect its operations.

Optimizing Memory Management

Understanding the lifespan patterns of your application's data is a key factor in effectively tuning your garbage collection strategy. This knowledge allows you to strike a balance between throughput and pause times, which are the primary trade-offs offered by different GC algorithms. Ultimately, this leads to more efficient memory management.

For instance, consider an application with a high rate of transient or short-lived data creation. Such an application might benefit from an algorithm that optimizes minor GC cycles. This optimization could involve allowing objects to age within the young generation, thereby reducing the frequency of old-generation collections and improving overall throughput.

Now consider an application with a significant amount of long-lived data. This data might be spread throughout the lifespan of the application, serving as an active cache for transactions. In this scenario, a GC algorithm that incrementally handles old-generation collections could be beneficial. Such an approach can help manage the larger LDS more efficiently, reducing pause times and maintaining application responsiveness.

Obviously, the behavior and performance of GCs can vary based on the specific characteristics of your workloads and the configuration of your JVM. As such, regular monitoring and tuning are essential. By adapting your GC strategy to the changing needs of your application, you can maintain optimal GC performance and ensure the efficient operation of your application.

This page intentionally left blank

Chapter 7

Runtime Performance Optimizations: A Focus on Strings, Locks, and Beyond

Introduction

Efficiency in runtime performance is crucial for scaling applications and meeting user demands. At the heart of this is the JVM, managing runtime facets such as just-in-time (JIT) compilation, garbage collection, and thread synchronization—each profoundly influencing Java applications' performance.

Java's evolution from Java 8 to Java 17 showcases a commitment to performance optimization, evident in more than 350 JDK Enhancement Proposals (JEPs). Noteworthy are optimizations like the reduced footprint of `java.lang.String`, advancements in contended locking mechanisms, and the *indy-fication* of string concatenation,[1] all significantly contributing to runtime efficiency. Additionally, optimizations such as spin-wait hints, though less apparent, are crucial in enhancing system performance. They are particularly effective in cloud platforms and multithreaded environments where the wait time is expected to be very short. These intricate optimizations, often concealed beneath the complexities of large-scale environments, are central to the performance gains realized by service providers.

Throughout my career, I have seen Java's runtime evolve dynamically. Major updates and preview features, like virtual threads, are increasingly supported by various frameworks[2] and libraries and adeptly integrated by large organizations. This integration, as discussed by industry leaders at a QCon San Francisco conference,[3] underscores Java's strategic role in these organizations and points toward a shift to a deeper, architectural approach to JVM optimizations.

[1] https://openjdk.org/jeps/280

[2] https://spring.io/blog/2023/11/23/spring-boot-3-2-0-available-now

[3] https://qconsf.com/track/oct2023/jvm-trends-charting-future-productivity-performance

This trend is particularly evident in high-performance databases and storage systems where efficient threading models are critical. The chapter aims to demystify the "how" and "why" behind these performance benefits, covering a broad spectrum of runtime aspects from bytecode engineering to string pools, locks, and the nuances of threading.

Further emphasizing the runtime, this chapter explores G1's string deduplication and transitions into the realm of concurrency and parallelism, unveiling Java's Executor Service, thread pools, and the `ForkJoinPool` framework. Additionally, it introduces Java's `CompletableFuture` and the groundbreaking concepts of virtual threads and *continuations,* providing a glimpse into the future of concurrency with Project Loom. While various frameworks may abstract these complexities, an overview and reasoning behind their evaluation and development is imperative for leveraging the JVM's threading advancements.

Understanding the holistic optimization of the internal frameworks within the JVM, as well as their coordination and differentiation, empowers us to better utilize the tools that support them, thereby bridging the benefits of JVM's runtime improvements to practical applications.

A key objective of this exploration is to discuss how the JVM's internal engineering augments code execution efficiency. By dissecting these enhancements through the lens of performance engineering tools, profiling instruments, and benchmarking frameworks, we aim to reveal their full impact and offer a thorough understanding of JVM performance engineering. Although these improvements to the JVM, may manifest differently in distributed environments, this chapter distills the core engineering principles of the JVM alongside the performance engineering tools available to us.

As we journey through runtime's performance landscape, novices and seasoned developers alike will appreciate how the JVM's continual enhancements streamline the development process. These refinements contribute to a more stable and performant foundation, which leads to fewer performance-related issues and a smoother deployment process, refining the comprehensive system experience in even the most demanding of environments.

String Optimizations

As the Java platform has evolved, so, too, has its approach to one of the most ubiquitous constructs in programming: the string. Given that strings are a fundamental aspect of nearly every Java application, any improvements in this area can significantly impact the speed, memory footprint, and scalability of an application. Recognizing this relationship, the Java community has dedicated considerable effort to string optimization. This section focuses on four significant optimizations that have been introduced across various Java versions:

- **Literal and interned string optimization:** This optimization involves the management of a "pool of strings" to reduce heap occupancy, thereby enhancing memory utilization.
- **String deduplication in G1:** The deduplication feature offered by the G1 garbage collector aims to reduce heap occupancy by deduplicating identical `String` objects.
- **Indy-fication of string concatenation:** We'll look at the change in how string concatenation is managed at the bytecode level, which enhances performance and adds flexibility.

- **Compact strings:** We will examine how the shift from `char[]` to `byte[]` backing arrays for strings can reduce applications' memory footprint, halving the space requirements for ASCII text.

Each optimization not only brings unique advantages but also presents its own set of considerations and caveats. As we explore each of these areas, we aim to understand both their technical aspects and their practical implications for application performance.

Literal and Interned String Optimization in HotSpot VM

Reducing the allocation footprint in Java has always been a significant opportunity for enhancing performance. Long before Java 9 introduced the shift from `char[]` to `byte[]` for backing arrays in strings, the OpenJDK HotSpot VM employed techniques to minimize memory usage, particularly for strings.

A principal optimization involves a "pool of strings." Given strings are immutable and often duplicated across various parts of an application, the `String` class in Java utilizes a pool to conserve heap space. When the `String.intern()` method is invoked, it checks if an identical string is already in the pool. If so, the method returns a reference to the pre-existing string instead of allocating a new object.

Consider an instance, *String1*, which would have had a `String1` object and a backing `char[]` in the days before Java 9 (Figure 7.1).

Invoking `String1.intern()` checks the pool for an existing string with the same value. If none is found, `String1` is added to the pool (Figure 7.2).

Figure 7.3 depicts the situation before the strings are interned. When a new string, `String2`, is created and interned, and it equals `String1`, it will reference the same `char[]` in the pool (Figure 7.4).

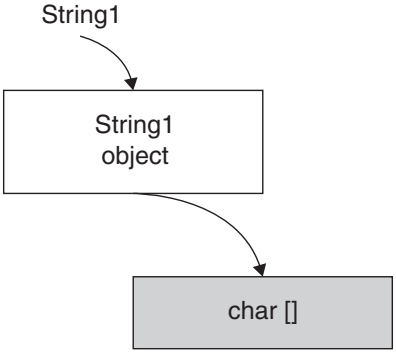

Figure 7.1 A String Object