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.

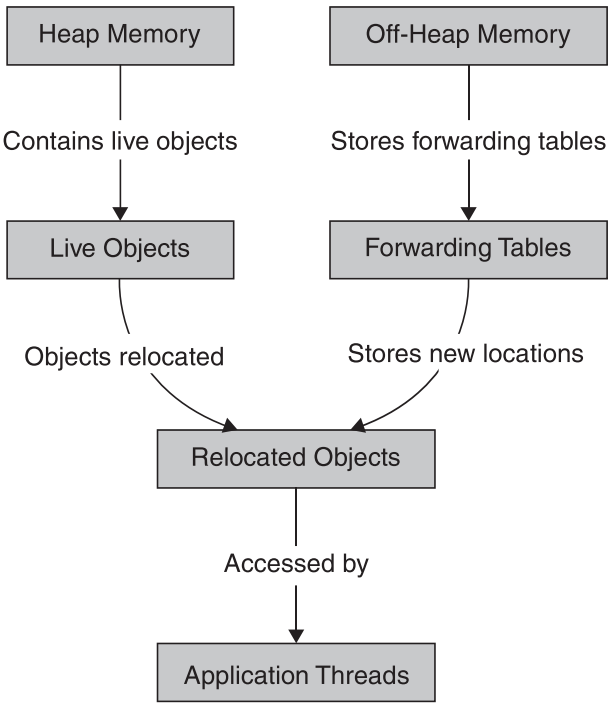

Figure 6.14 ZGC's Off-Heap Forwarding Tables

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.

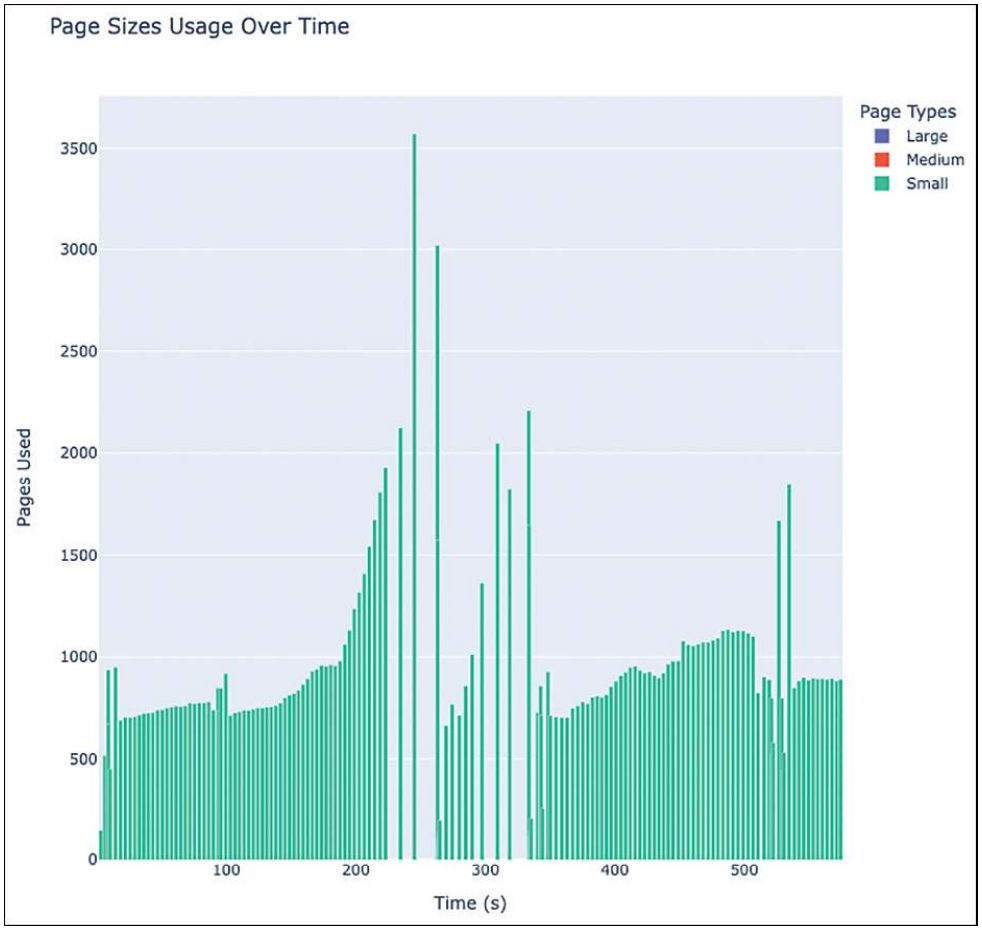

Figure 6.15 Page Sizes Usage Over Time

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

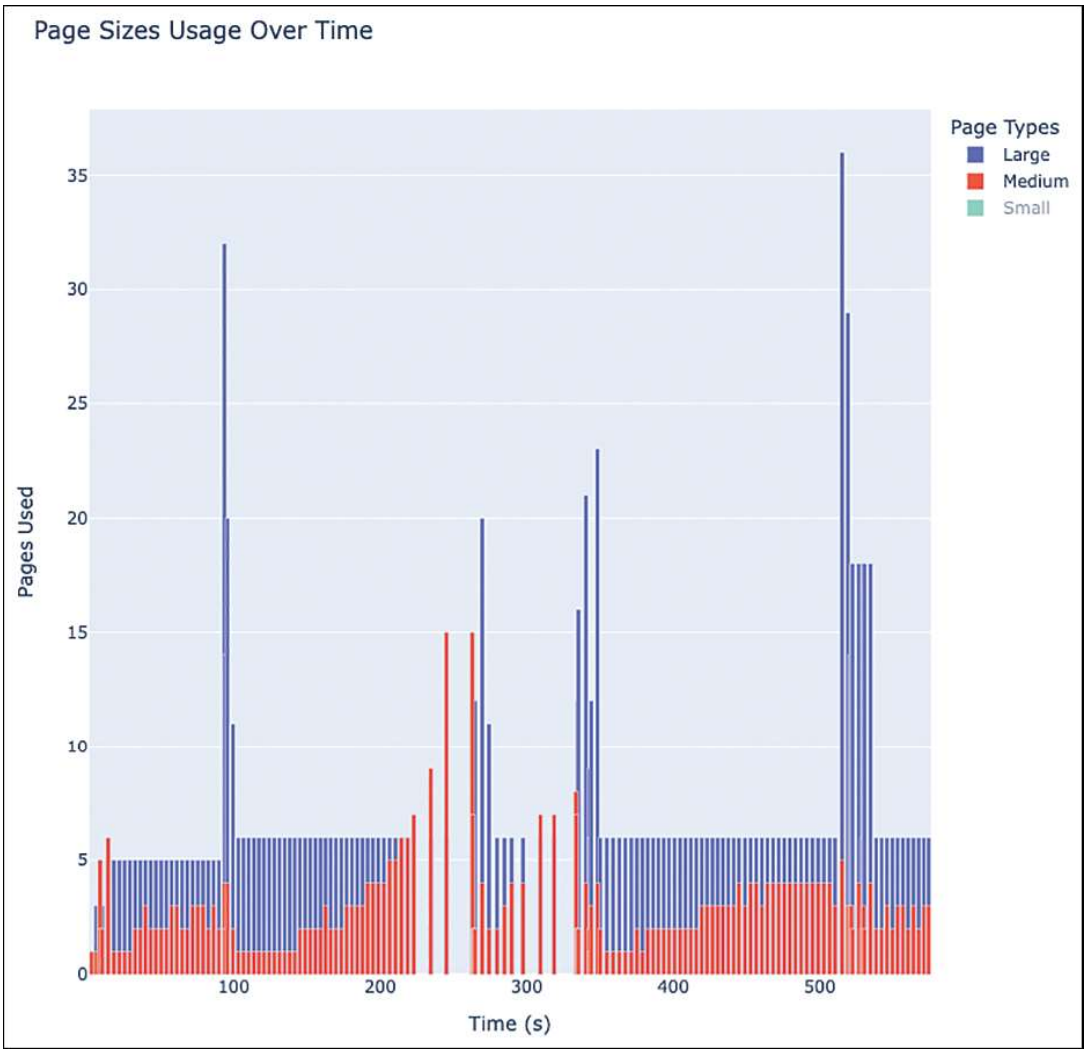

Figure 6.16 Medium and Large ZPages Over Time

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).

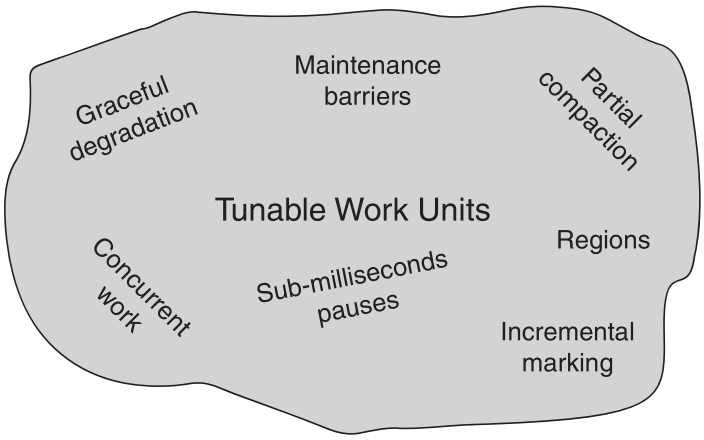

Figure 6.17 A Concurrent Garbage Collector's Tunable Work Units

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).

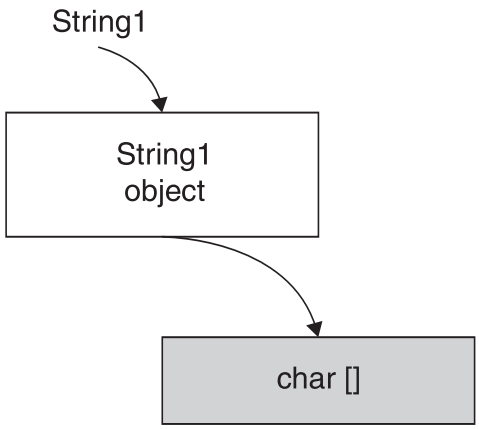

Figure 7.1 A String Object

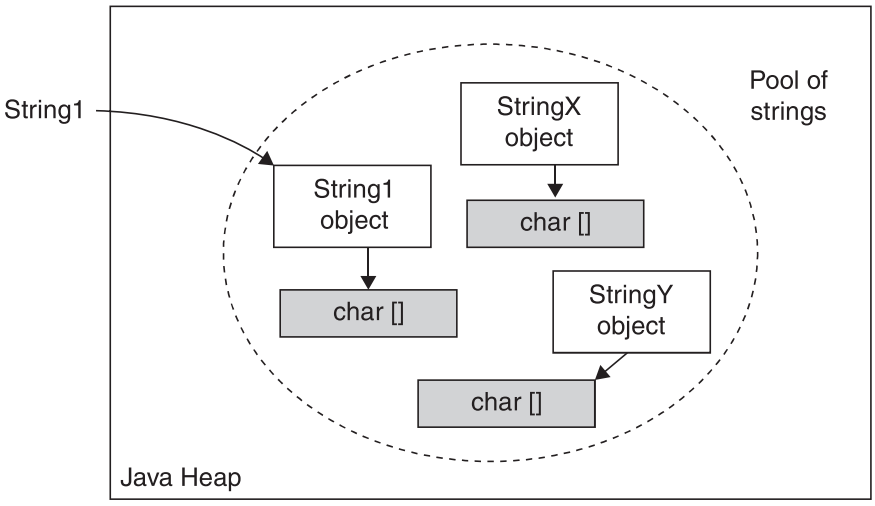

Figure 7.2 A Pool of Strings Within the Java Heap

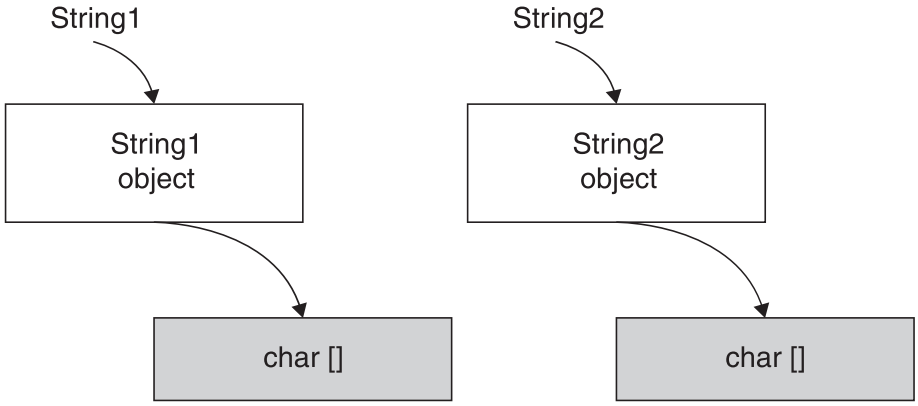

Figure 7.3 Two String Objects Before Interning

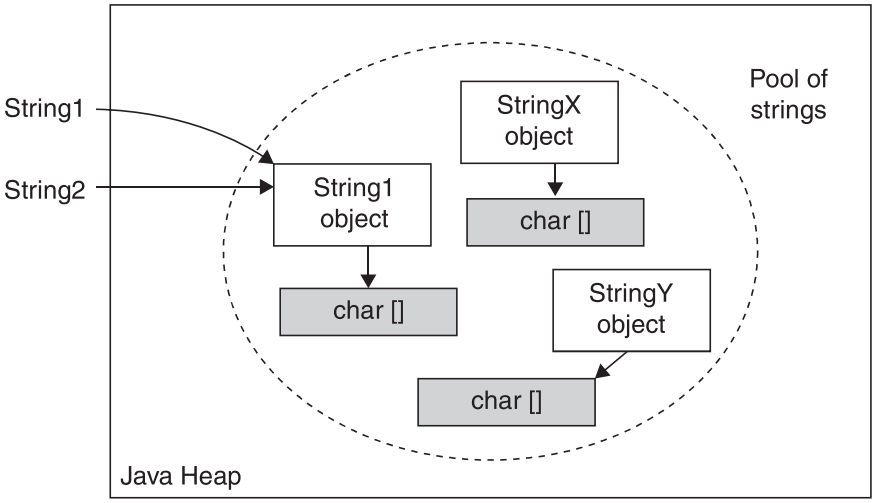

Figure 7.4 Interned String Objects Sharing the Same Reference

This method of storing strings means that memory efficiency is greatly improved because identical strings are stored only once. Furthermore, interned strings can be compared quickly and efficiently using the == operator because they share the same reference, rather than using the `equals()` method. This not only saves memory but also speeds up string comparison operations, contributing to faster execution times for the application.

String Deduplication Optimization and G1 GC

Java 8 update 20 enhanced the G1 GC with a string deduplication (*dedup*) optimization, aimed at reducing the memory footprint. During a stop-the-world (STW) evacuation pause (young or mixed) or the marking phase of the fallback full collection, the G1 GC identifies deduplication candidate objects. These candidates are instances of `String` that have identical `char[]` and have been evacuated from the eden regions into the survivor regions or promoted into the old-generation regions and the object's age is equal to or less than (respectively) the *dedup* threshold. *Dedup* entails reassigning the value field of one `String` instance to another, thereby avoiding the need for a duplicate backing array.

It's crucial to note that most of the work happens in a VM internal thread called the "deduplication thread." This thread runs concurrently with your application threads, processing the *dedup* queues by computing hash codes and deduplicating strings as needed.

Figure 7.5 captures the essence where if `String1` equals `String2` character-by-character, and both are managed by the G1 GC, the duplicate character arrays can be replaced with a single shared array.

The G1 GC's string *dedup* feature is not enabled by default and can be controlled through JVM command-line options. For instance, to enable this feature, use the `-XX:+UseStringDeduplication` option. Also, you can customize the age threshold for a `String` object's eligibility for *dedup* using the `-XX:StringDeduplicationAgeThreshold=<#>` option.

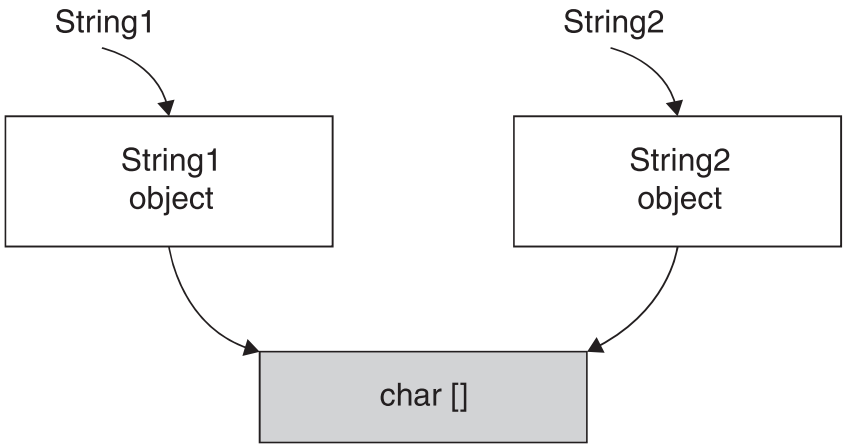

Figure 7.5 String Deduplication Optimization in Java

For developers interested in monitoring the effectiveness of deduplication, you can use the `-Xlog:stringdedup*=debug`[4] option to get detailed insights. Here's an example log output:

```
…
[217.149s][debug][stringdedup,phases,start] Idle start
[217.149s][debug][stringdedup,phases      ] Idle end: 0.017ms
[217.149s][debug][stringdedup,phases,start] Concurrent start
[217.149s][debug][stringdedup,phases,start] Process start
[217.149s][debug][stringdedup,phases      ] Process end: 0.069ms
[217.149s][debug][stringdedup,phases      ] Concurrent end: 0.109ms
[217.149s][info ][stringdedup             ] Concurrent String Deduplication 45/2136.0B (new),
1/40.0B (deduped), avg 114.4%, 0.069ms of 0.109ms
[217.149s][debug][stringdedup             ]   Last Process: 1/0.069ms, Idle: 1/0.017ms, Blocked:
0/0.000ms
[217.149s][debug][stringdedup             ]     Inspected:              64
[217.149s][debug][stringdedup             ]       Known:                19( 29.7%)
[217.149s][debug][stringdedup             ]       Shared:                0(  0.0%)
[217.149s][debug][stringdedup             ]       New:                  45( 70.3%)  2136.0B
[217.149s][debug][stringdedup             ]       Replaced:              0(  0.0%)
[217.149s][debug][stringdedup             ]       Deleted:               0(  0.0%)
[217.149s][debug][stringdedup             ]     Deduplicated:            1(  1.6%)    40.0B(  1.9%)
…
```

It is important for developers to understand that while string deduplication can significantly reduce memory usage, there is a trade-off in terms of CPU overhead. The *dedup* process itself requires computational resources, and its impact should be assessed in the context of the specific application requirements and performance characteristics.

Reducing Strings' Footprint

Java has always been at the forefront of runtime optimization, and with the growing need for efficient memory management, Java community leaders have introduced several enhancements to reduce the memory footprint of strings. This optimization was delivered in two significant parts: (1) the *indy-fication* of string concatenation and (2) the introduction of compact strings by changing the backing array from `char[]` to `byte[]`.

Indy-fication of String Concatenation

Indy-fication involves the use of the `invokedynamic` feature from JSR 292,[5] which introduced support for dynamic languages into JVM as part of Java SE 7. JSR 292 changed the JVM specification by adding a new instruction, `invokedynamic`, that supports dynamic typed languages by deferring the translation to the runtime engine.

[4] Chapter 4, "The Unified Java Virtual Machine Logging Interface," provides an in-depth explanation of the unified logging tags and levels.

[5] https://jcp.org/en/jsr/detail?id=292

String concatenation is a fundamental operation in Java allowing the union of two or more strings. If you've ever used the `System.out.println` method, you're already familiar with the "+" operator for this purpose. In the following example, the phrase `"It is"` is concatenated with the Boolean `trueMorn` and further concatenated with the phrase `"that you are a morning person"`:

```
System.out.println("It is " + trueMorn + " that you are a morning person");
```

The output would vary based on the value of `trueMorn`. Here is one possible output:

```
It is false that you are a morning person
```

To understand the improvements , let's compare the generated bytecode between Java 8 and Java 17. Here's the static method:

```
private static void getMornPers() {
   System.out.println("It is " + trueMorn + " that you are a morning person");
}
```

We can use the `javap` tool to look at the generated bytecode for this method with Java 8. Here's the command line:

```
$ javap -c -p MorningPerson.class
```

Here's the bytecode disassembly:

```
  private static void getMornPers();
    Code:
       0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: new           #9                  // class java/lang/StringBuilder
       6: dup
       7: invokespecial #10                 // Method java/lang/StringBuilder."<init>":()V
      10: ldc           #11                 // String It is
      12: invokevirtual #12                 // Method java/lang/StringBuilder.append:(Ljava/lang/
                                            String;)Ljava/lang/StringBuilder;
      15: getstatic     #8                  // Field trueMorn:Z
      18: invokevirtual #13                 // Method java/lang/StringBuilder.append:(Z)Ljava/
                                            lang/StringBuilder;
      21: ldc           #14                 // String that you are a morning person
      23: invokevirtual #12                 // Method java/lang/StringBuilder.append:(Ljava/lang/
                                            String;)Ljava/lang/StringBuilder;
      26: invokevirtual #15                 // Method java/lang/StringBuilder.toString:()Ljava/
                                            lang/String;
      29: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/
                                            String;)V
      32: return
```

In this bytecode, we can observe that the "+" operator was internally translated into a series of `StringBuilder` API calls to `append()`. We can rewrite the original code using the `StringBuilder` API:

```
String output = new StringBuilder().append("It is ").append(trueMorn).append(" that you are a
morning person").toString();
```

The bytecode for this will be similar to that using the "+" operator.

However, with Java 17, the bytecode disassembly has a significantly different appearance:

```
  private static void getMornPers();
    Code:
       0: getstatic     #7                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: getstatic     #37                 // Field trueMorn:Z
       6: invokedynamic #41,  0             // InvokeDynamic #0:makeConcatWithConstants:(Z)Ljava/
                                               lang/String;
      11: invokevirtual #15                 // Method java/io/PrintStream.println:(Ljava/lang/
                                               String;)V
      14: return
```

This bytecode is optimized to use the `invokedynamic` instruction with the `makeConcatWithConstants` method. This transformation is referred to as *indy-fication* of string concatenation.

The `invokedynamic` feature introduces a bootstrap method (BSM) that assists `invokedynamic` call sites. All `invokedynamic` call sites have method handles that provide behavioral information for these BSMs, which are invoked during runtime. To view the BSM, you add the `-v` option to the `javap` command line:

```
$ javap -p -c -v MorningPerson.class
```

Here's the BSM excerpt from the output:

```
...
SourceFile: "MorningPerson.java"
BootstrapMethods:
  0: #63 REF_invokeStatic java/lang/invoke/StringConcatFactory.makeConcatWithConstants:(Ljava/
lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/MethodType;Ljava/lang/
String;[Ljava/lang/Object;)Ljava/lang/invoke/CallSite;
    Method arguments:
      #69 It is \u0001 that you are a morning person
InnerClasses:
  public static final #76= #72 of #74;    // Lookup=class java/lang/invoke/MethodHandles$Lookup of
class java/lang/invoke/MethodHandles
```

As we can see, the runtime first converts the `invokedynamic` call to an `invokestatic` call. The method arguments for the BSM specify the constants to be linked and loaded from the

constant pool. With the BSM's assistance and by passing a runtime parameter, it becomes possible to precisely place the parameter within the prebuilt string.

The adoption of `invokedynamic` for string concatenation brings several benefits. For starters, special strings, such as empty and *null* strings, are now individually optimized. When dealing with an empty string, the JVM can bypass unnecessary operations, simply returning the non-empty counterpart. Similarly, the JVM can efficiently handle *null* strings, potentially sidestepping the creation of new string objects and avoiding the need for conversions.

One notable challenge with traditional string concatenation, especially when using `StringBuilder`, was the advice to precalculate the total length of the resulting string. This step aimed at ensuring that `StringBuilder`'s internal buffer didn't undergo multiple, performance-costly resizes. With `invokedynamic`, this concern is alleviated—the JVM is equipped to dynamically determine the optimal size for the resulting string, ensuring efficient memory allocation without the overhead of manual length estimations. This functionality is especially beneficial when the lengths of the strings being concatenated are unpredictable.

Furthermore, the `invokedynamic` instruction's inherent flexibility means that string concatenation can continually benefit from JVM advancements. As the JVM evolves, introducing new optimization techniques, string concatenation can seamlessly reap these benefits without necessitating bytecode or source code modifications. This not only ensures that string operations remain efficient across JVM versions, but also future-proofs applications against potential performance bottlenecks.

Thus, for developers keen on performance, the shift to `invokedynamic` represents a significant advancement. It simplifies the developer's task, reducing the need for explicit performance patterns for string concatenation, and ensures that string operations are automatically optimized by the JVM. This reliance on the JVM's evolving intelligence not only yields immediate performance gains but also aligns your code with future enhancements, making it a smart, long-term investment for writing faster, cleaner, and more efficient Java code.

Compact Strings

Before diving into compact strings, it's essential to understand the broader context. Although performance in computing often revolves around metrics like response time or throughput, improvements related to the `String` class in Java have mostly focused on a slightly different domain: memory efficiency. Given the ubiquity of strings in applications, from texts to XML processing, enhancing their memory footprint can significantly impact an application's overall efficiency.

In JDK 8 and earlier versions, the JVM used a character array (`char[]`) to store `String` objects internally. With Java's UTF-16 encoding for characters, each `char` consumes 2 bytes in memory. As noted in JEP 254: *Compact Strings*,[6] the majority of strings are encoded using the Latin-1 character set (an 8-bit extension of 7-bit ASCII), which is the default character set for HTML. Latin-1 can store a character using only 1 byte—so JDK 8 essentially wasted 1 byte of memory for each Latin-1 character stored.

[6] https://openjdk.org/jeps/254

Impact of Compact Strings

JDK 9 introduced a substantial change to address the issue, with JEP 254: *Compact Strings* altering the internal representation of the `String` class. Instead of a `char[]`, a `byte[]` was used to back a string. This fundamental modification reduced memory waste for Latin-1–encoded string characters while keeping the public-facing APIs unchanged.

Reflecting on my time at Sun/Oracle, we had attempted a similar optimization in Java 6 called *compressed strings*. However, the approach taken in Java 6 was not as effective because the compression was done during the construction of the `String` object, but the `String` operations were still performed on `char[]`. Thus, a "decompression" step was necessary, limiting the usefulness of the optimization.

In contrast, the compact strings optimization eliminated the need for decompression, thereby providing a more effective solution. This change also benefits the APIs used for string manipulation, such as `StringBuilder` and `StringBuffer`, along with the JIT compiler *intrinsics*[7] related to `String` operations.

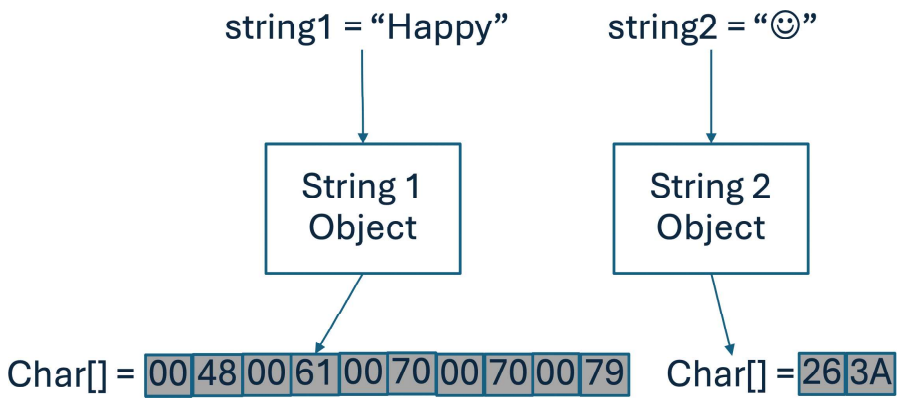

Figure 7.6 Traditional String Representation in JDK 8

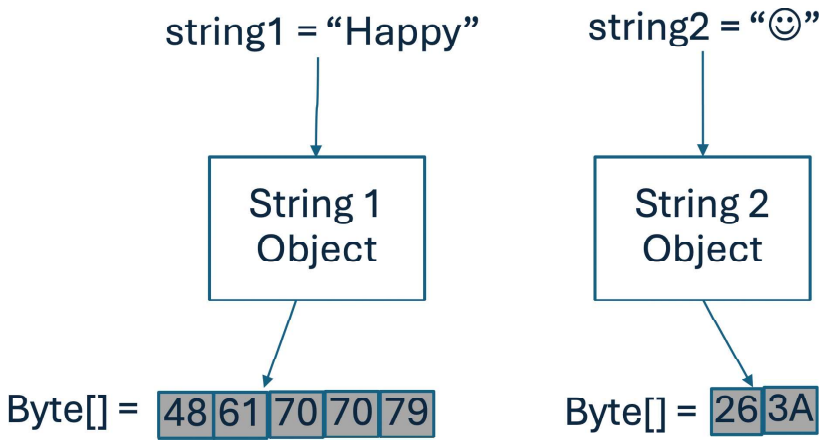

Figure 7.7 Compact Strings Representation in JDK 17

[7] https://chriswhocodes.com/hotspot_intrinsics_openjdk11.html