Visualizing String Representations

To better grasp this concept, let's diagrammatically visualize the representation of strings in JDK 8. Figure 7.6 illustrates how a `String` object is backed by a character array (`char[]`), with each character occupying 2 bytes.

Moving to JDK 9 and beyond, the optimization introduced by compact strings can be seen in Figure 7.7, where the `String` object is now backed by a byte array (`byte[]`), using only the necessary memory for Latin-1 characters.

Profiling Compact Strings: Insights into JVM Internals Using NetBeans IDE

To illustrate the benefits of compact strings optimization and to empower performance-minded Java developers with the means to explore the JVM's internals, we focus on the implementation of `String` objects and their backing arrays. To do this, we will use the NetBeans integrated development environment (IDE) to profile a simple Java program, providing an illustrative demonstration rather than a comprehensive benchmark. This approach is designed to highlight how you can analyze objects and class behaviors in the JVM—particularly how the JVM handles strings in these two different versions.

The Java code we use for this exercise is intended to merge content from two log files into a third. The program employs several key Java I/O classes, such as `BufferedReader`, `BufferedWriter`, `FileReader`, and `FileWriter`. This heavy engagement with string processing, makes it an ideal candidate to highlight the changes in `String` representation between JDK 8 and JDK 17. It is important to note that this setup serves more as an educational tool rather than a precise gauge of performance. The aim here is not to measure performance impact, but rather to provide a window into the JVM's internal string handling mechanics.

```
package mergefiles;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.PrintWriter;
import java.io.IOException;

public class MergeFiles {

    public static void main(String[] args) {

       try
       {
            BufferedReader appreader = new BufferedReader(new FileReader( "applog.log"));
            BufferedReader gcreader = new BufferedReader(new FileReader( "gclog.log"));
            PrintWriter writer = new PrintWriter(new BufferedWriter(new FileWriter("all_my_logs.
            log")));
```

```
            String appline, gcline;

                while ((appline = appreader.readLine()) != null)
                {
                    writer.println(appline);
                }

                while((gcline = gcreader.readLine()) != null)
                {
                    writer.println(gcline);
                }
        }
        catch (IOException e)
        {
            System.err.println("Caught IOException: " +  e.getMessage());
        }

    }

}
```

The NetBeans IDE, a popular tool for developers, comes equipped with powerful profiling tools, allowing us to inspect and analyze the runtime behavior of our program, including the memory consumption of our `String` objects. The profiling steps for JDK 8 are as follows:

1. **Initiate profiling**: Select "Profile Project" from the "Profile" menu as shown in Figure 7.8.

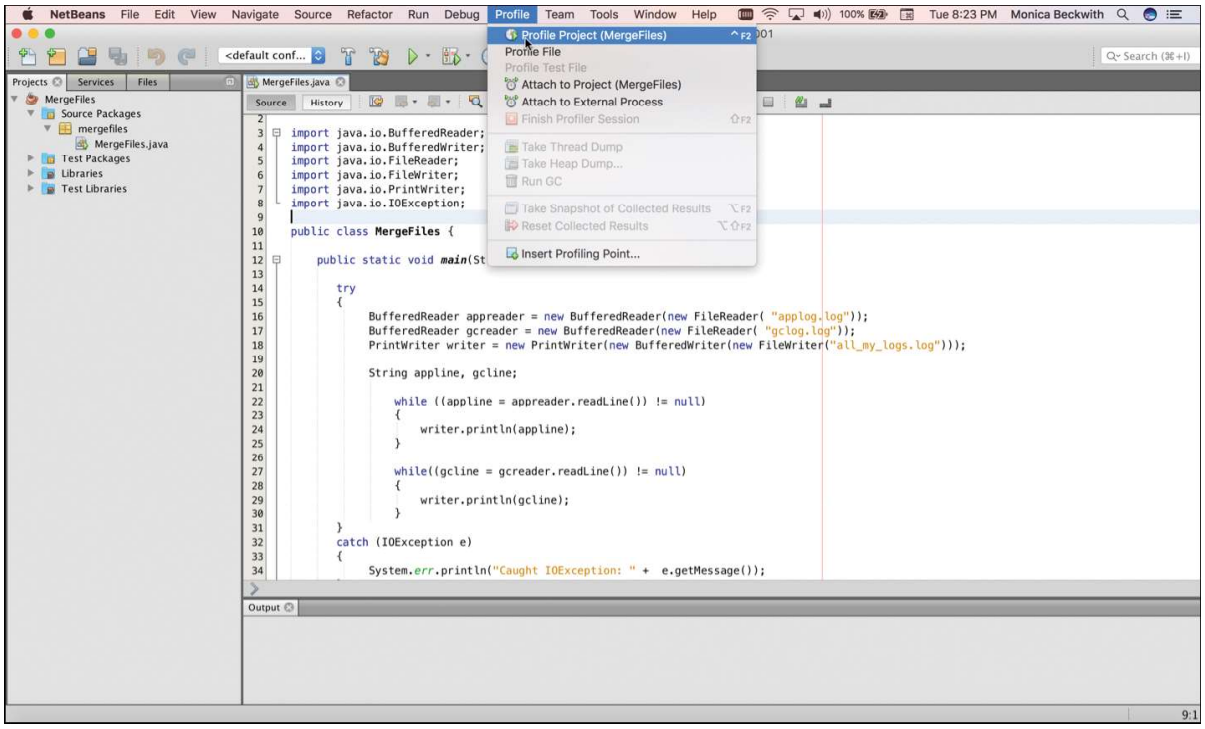

Figure 7.8 Apache NetBeans Profile Menu

2. **Configure profiling session:** From the "Configure Session" option, select "Objects" as shown in Figure 7.9.
3. **Start profiling:** In Figure 7.10, notice that the "Configure Session" button has changed to "Profile" with a "▶" icon, indicating that you click this button to start profiling.

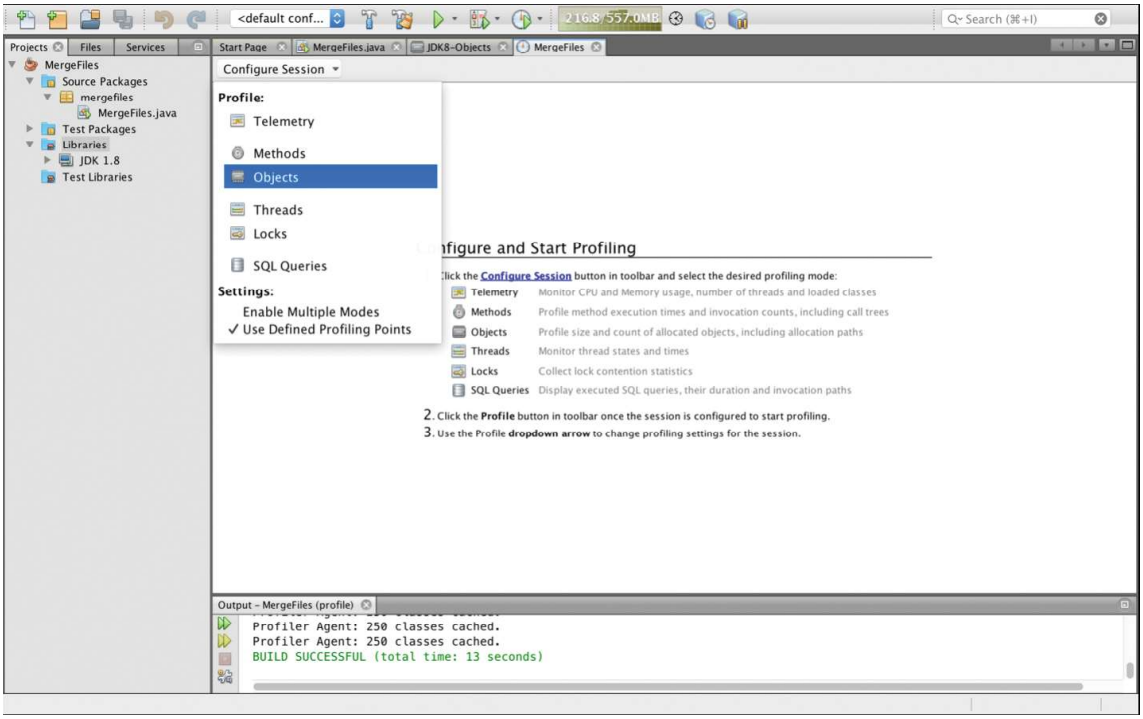

Figure 7.9 Apache NetBeans Configure Session

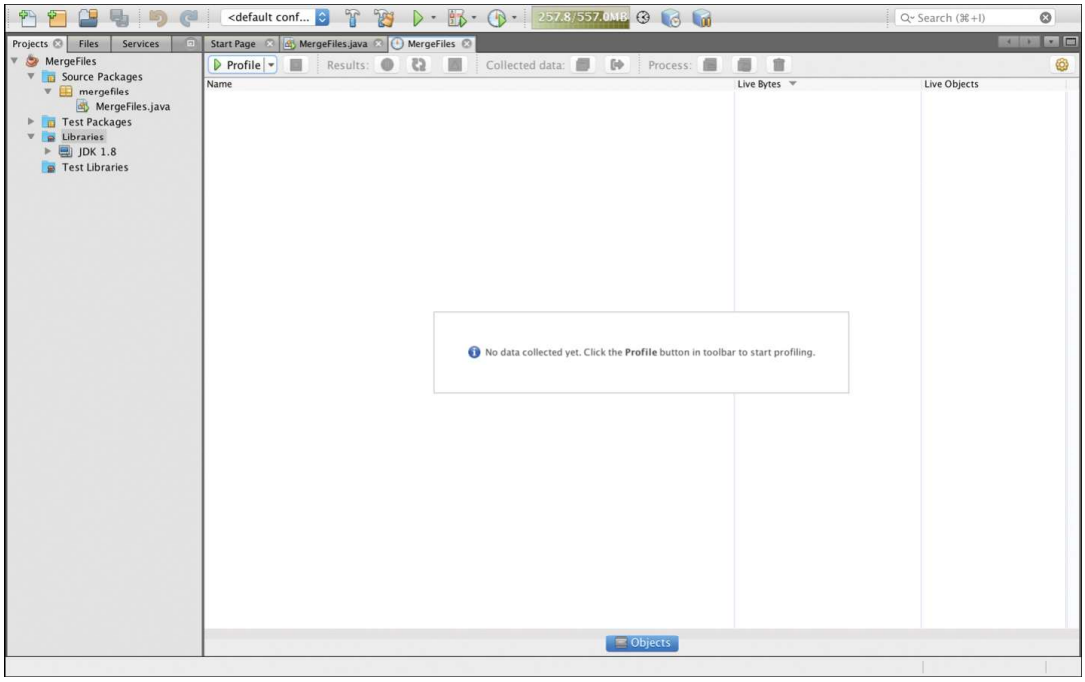

Figure 7.10 Apache NetBeans Profiling Setup

4. **Observing profiling output:** After you click the button, the profiling starts. The output appears in the Output window with the text "Profile Agent: ..." (Figure 7.11). Once the profiling is complete, you can save the snapshot. I saved mine as `JDK8-Objects`.
5. **Snapshot analysis:** Now let's zoom into the snapshot, as shown in Figure 7.12.
6. **Class-based profiling (JDK 8):** Given that the backing array in Java 8 is a `char[]`, it's not surprising that `char[]` appears at the top. To investigate further, let's profile based on the `char[]`. We'll look at the classes and aim to verify that this `char[]` is, in fact, the backing array for `String`. (See Figure 7.13.)

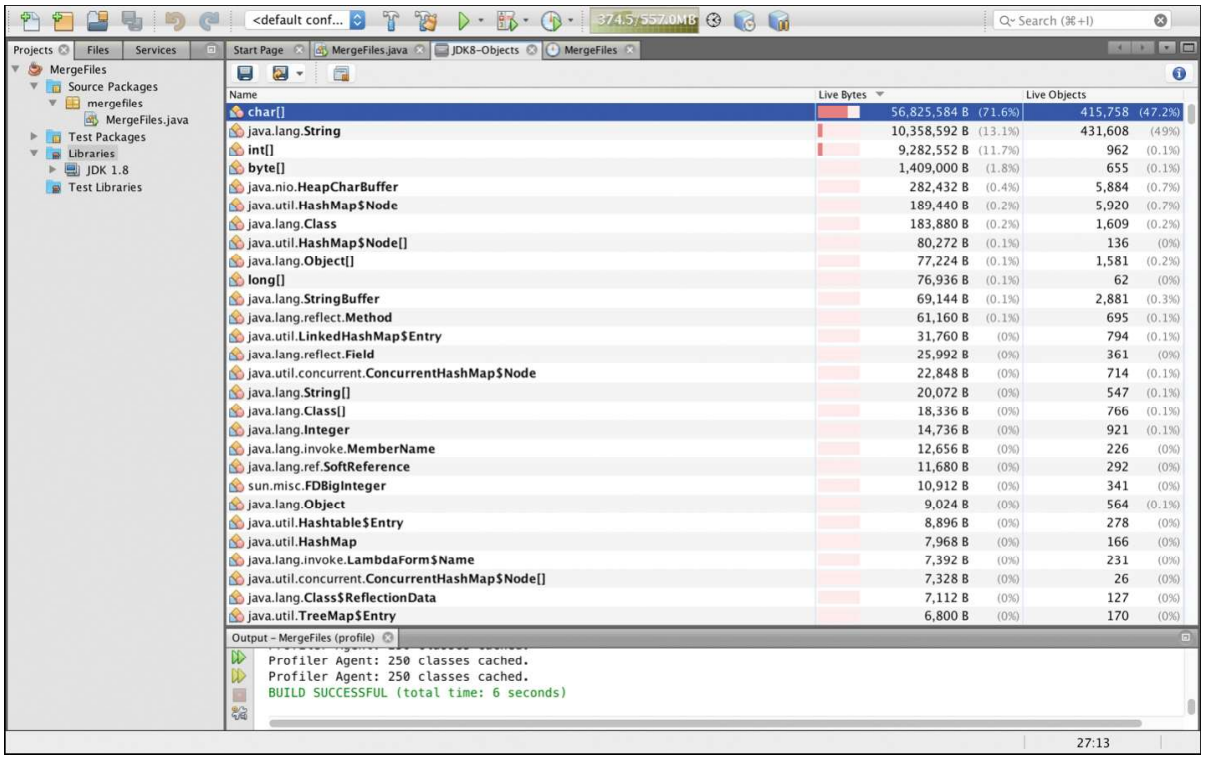

Figure 7.11 Apache NetBeans Profiling Complete for All Classes

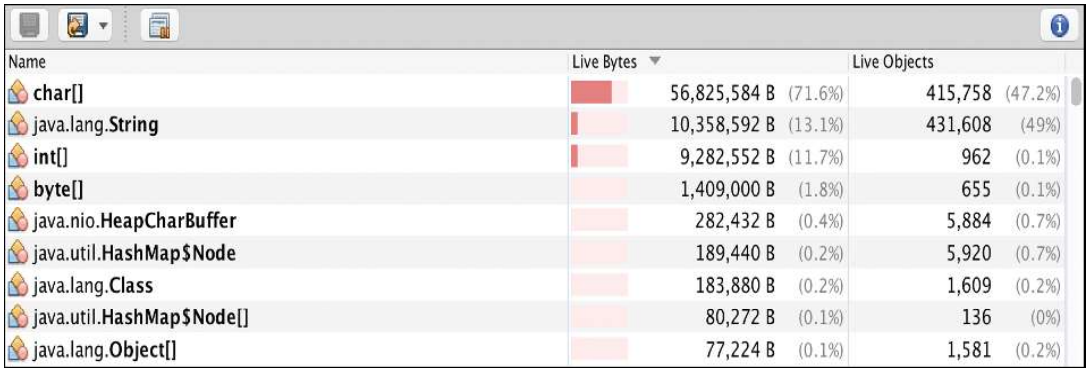

Figure 7.12 Zoomed-in Profiling Window

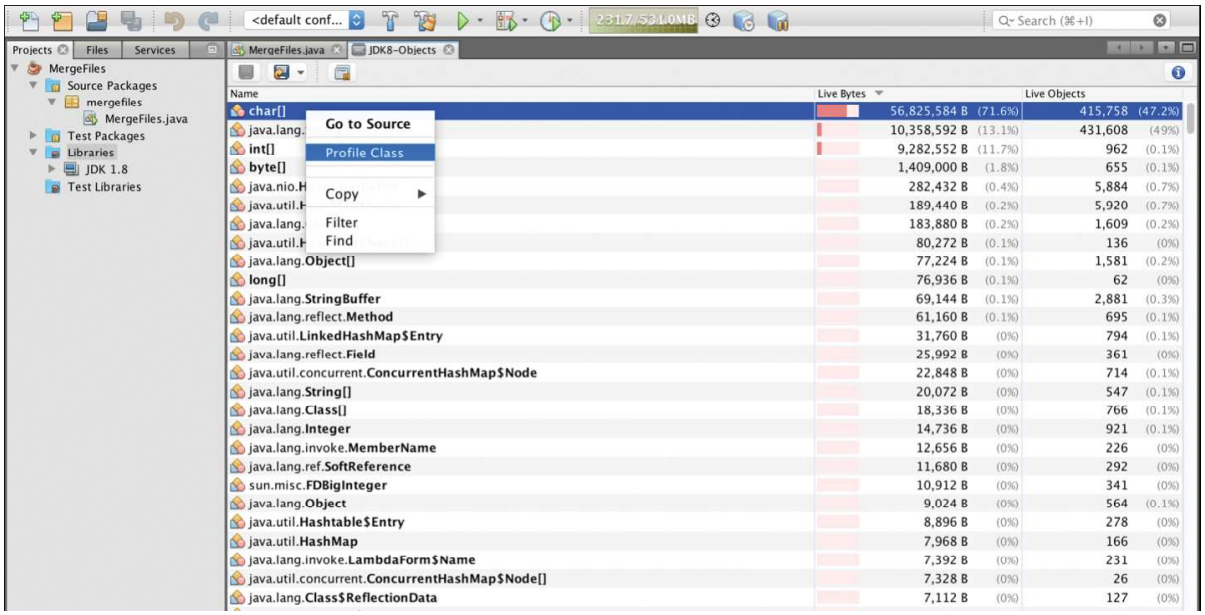

Figure 7.13 Apache NetBeans Profile Class Menu

7. **In-depth class analysis (JDK 8):** Once you select "Profile Class," you will see (as shown in Figure 7.14) that one class is selected. You can then click the "Profile" button to start profiling.

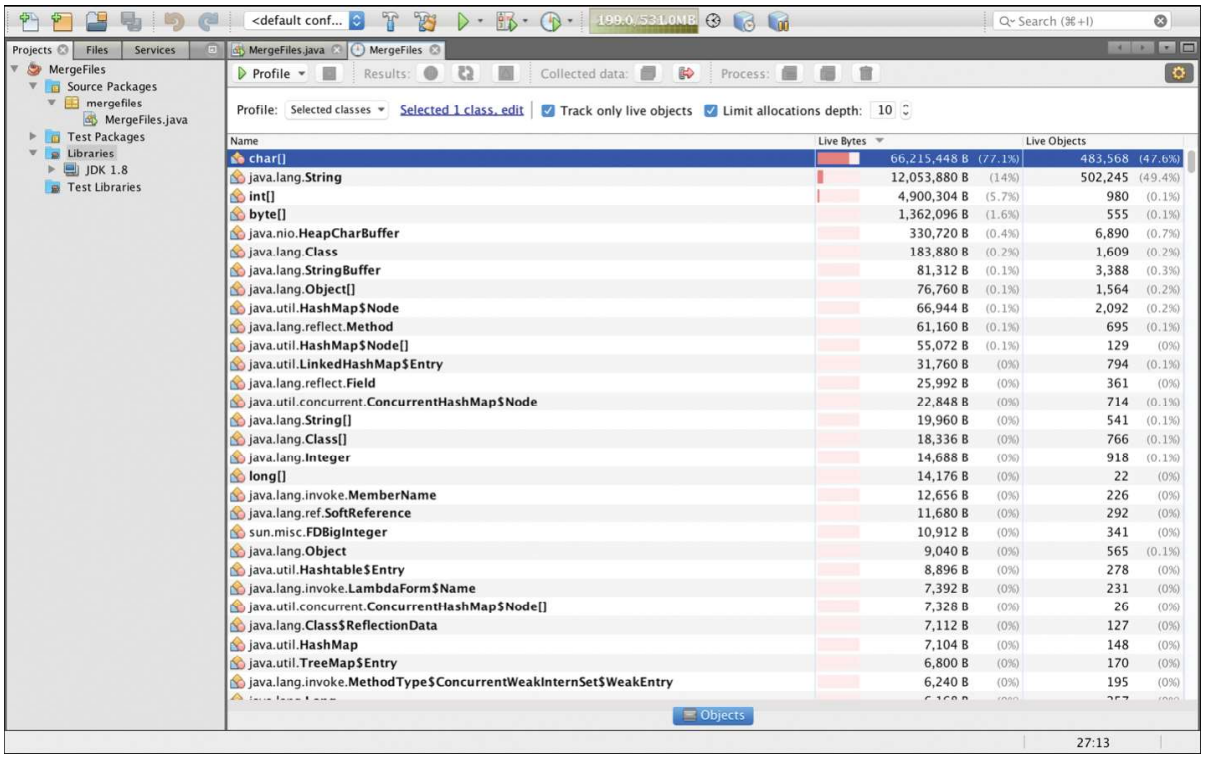

Figure 7.14 Apache NetBeans Profiling Single Class

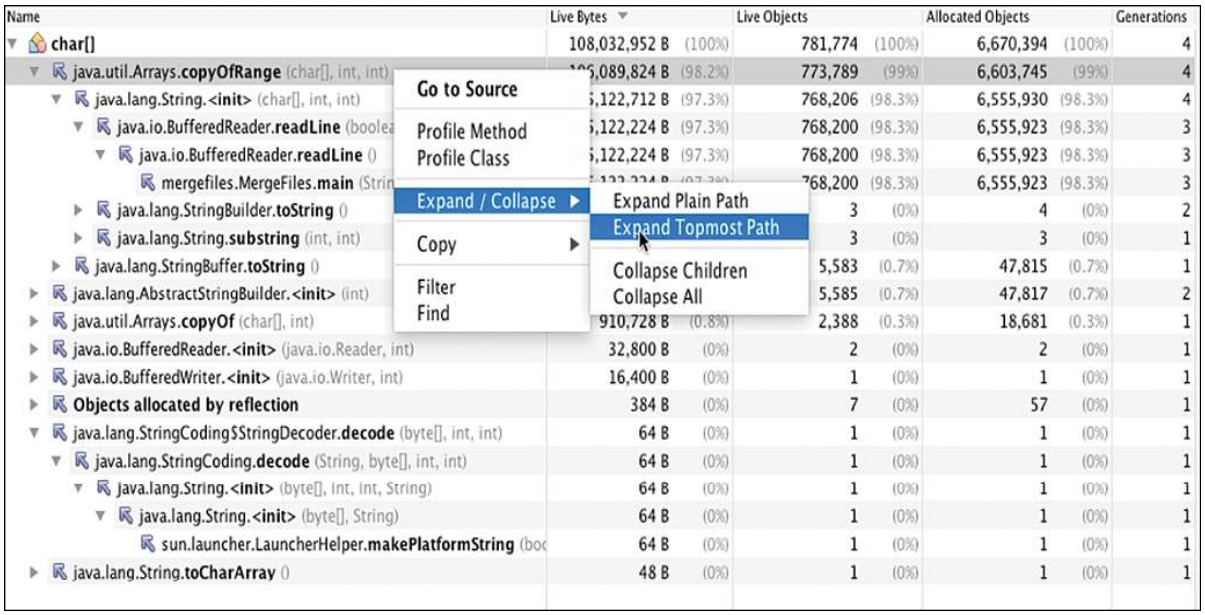

Figure 7.15 Apache NetBeans Expanded Class Menu

8. **Further analysis (JDK 8):** The profiling could take a couple of seconds depending on your system. When you expand the first, topmost path, the final window looks something like Figure 7.15.

Now, let's look at how this process changes with JDK 17:

1. **Repeat the initial steps**: For Java 17, I've used the Apache NetBeans 18 profiler, and I repeated steps 1 to 3 to profile for objects and the captured profile, as shown in Figure 7.16.
2. **Class-based profiling (JDK 17):** Since this is Java 17, the backing array has been changed to `byte[]`; hence, we see the `byte[]` object at the top. Thus, the compact strings change is in effect. Let's select `byte[]` and profile for classes to confirm that it is, in fact, a backing array for `String`. Figure 7.17 shows the result.
3. **String initialization analysis (JDK 17):** We can see the call to `StringUTF16.compress` from the `String` initialization. Also, we see `StringLatin1.newString()`, which ends up calling `Arrays.copyOfRange`. This call tree reflects the changes related to compact strings.
4. **Observations (JDK 17):** When we compare Figure 7.17 to the profile obtained from JDK 8, we note a change in memory usage patterns. Notably, `StringUTF16.compress` and `StringLatin1.newString` are missing from the JDK 8 profile (see Figure 7.18).

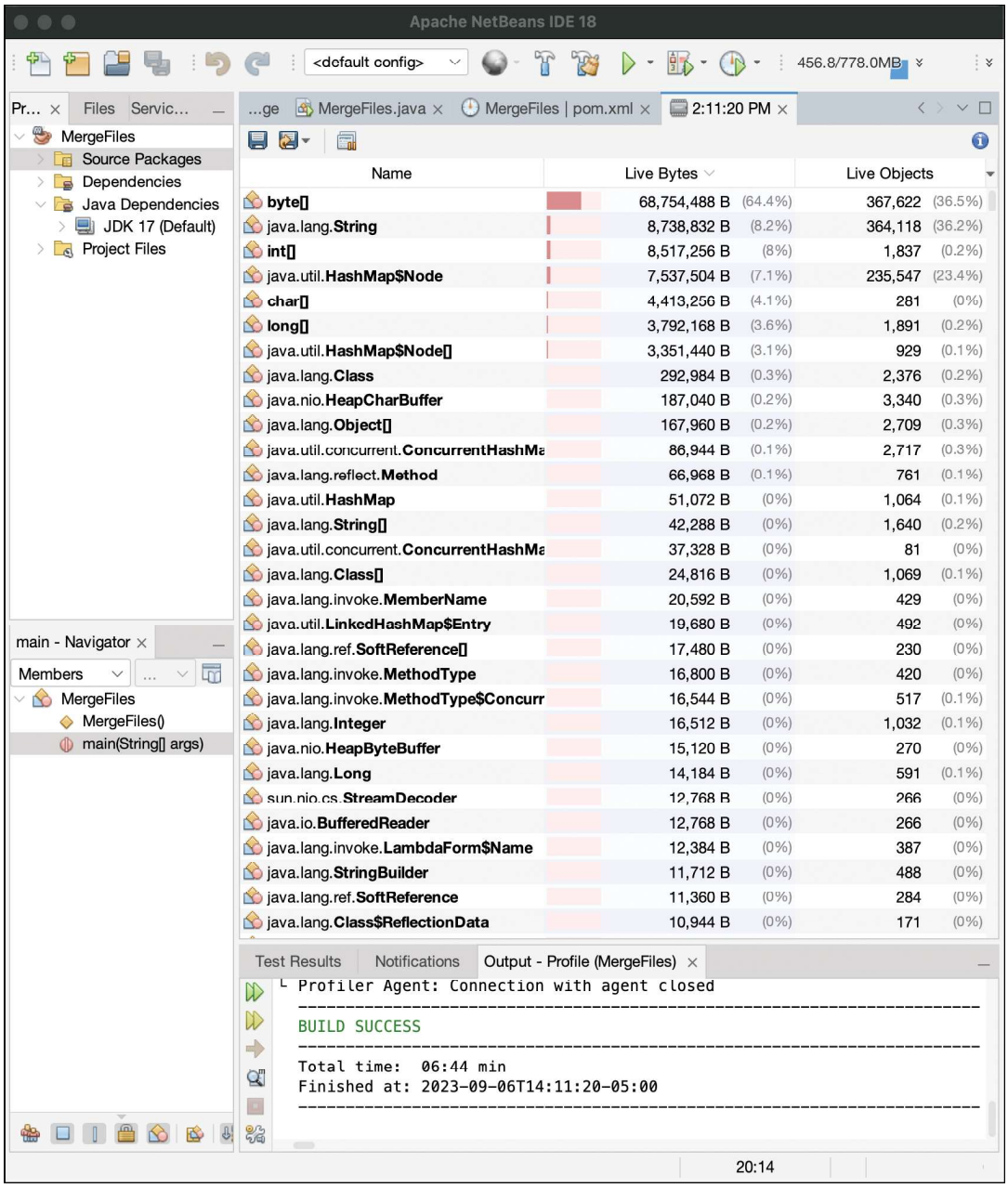

Figure 7.16 Apache NetBeans 18 Captured Profile

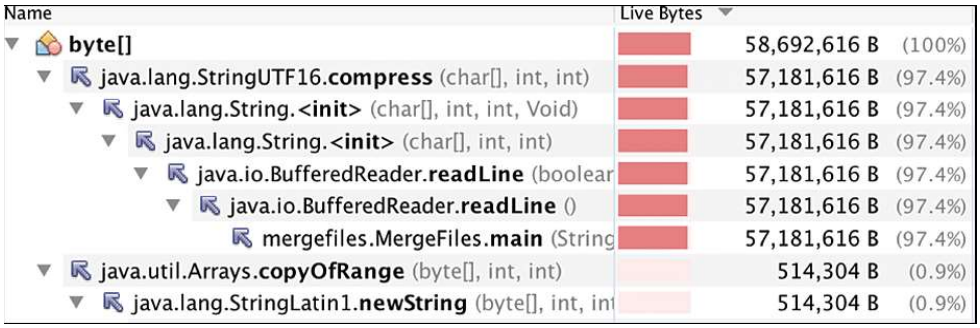

Figure 7.17 Zoomed-in Profile Showing byte[] at the Top

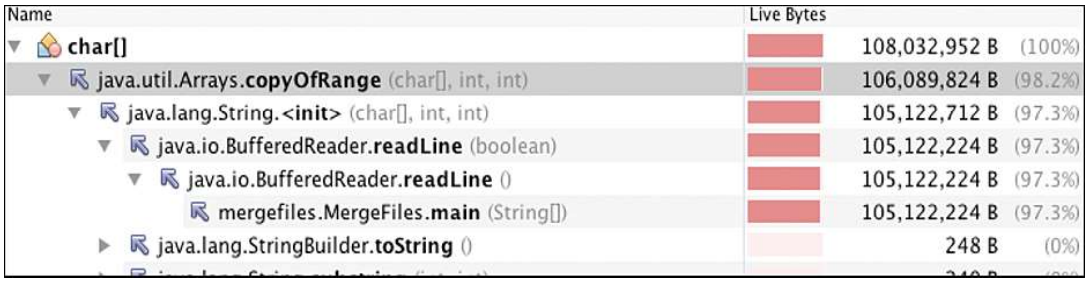

Figure 7.18 Zoomed-in Profile Showing Char[] at the Top

By following these steps and comparing the results, the impact of compact strings becomes evident. In JDK 8, `char[]` emerges as the main memory-consuming object, indicating its role as the backing array for `String`. In JDK 17, `byte[]` takes on this role, suggesting a successful implementation of compact strings. By analyzing the profiling results, we validate the efficiency of byte-backed arrays in JDK 17 for compressing UTF16 strings and offer insights into the JVM's internal string management mechanisms.

Given the marked efficiency enhancements in String objects with JDK 17 (and JDK 11), it becomes equally important to explore other areas where the runtime can lead to better performance. One such critical area, which plays a pivotal role in ensuring the smooth operation of multithreaded applications, involves grasping the fundamentals of synchronization primitives and the runtimes' advancements in this field.

Enhanced Multithreading Performance: Java Thread Synchronization

In today's high-performance computing landscape, the ability to execute tasks concurrently is not just an advantage—it's a necessity. Effective multithreading allows for efficient scaling, harnessing the full potential of modern processors, and optimizing the utilization of various resources. Sharing data across threads further enhances resource efficiency. However, as systems scale, it's not just about adding more threads or processors; it's also about understanding the underlying principles that govern scalability.

Enter the realm of scalability laws. Many developers are familiar with Amdahl's law, which highlights the limits of parallelization by focusing on the serial portion of a task, but doesn't capture all the nuances of real-world parallel processing. Gunther's Universal Scalability Law (USL) provides a more comprehensive perspective. USL considers both contention for shared resources and the coherency delays that arise when ensuring data updates are consistent across processors. By understanding these factors, developers can anticipate potential bottlenecks and make informed decisions about system design and optimization.

As it stands, a theoretical understanding of scalability, while essential, is only part of the equation. Practical challenges arise when multiple threads, in their quest for efficiency, attempt concurrent access to shared resources. Mutable shared state, if not managed judiciously, becomes a hotbed for data races and memory anomalies. Java, with its rich concurrent programming model, rises to meet this challenge. Its suite of concurrent API utilities and synchronization constructs empower developers to craft applications that are both performant and thread safe.

Central to Java's concurrency strategy is the Java Memory Model (JMM). The JMM defines the legal interactions of threads with memory in a Java application. It introduces the concept of the *happens-before* relationship—a fundamental ordering constraint that guarantees memory visibility and ensures that certain memory operations are sequenced before others.

For instance, consider two threads, "a" and "b." If thread "a" modifies a shared variable and thread "b" subsequently reads that variable, a *happens-before* relationship must be established to ensure that "b" observes the modification made by "a."

To facilitate such ordered interactions and prevent memory anomalies, Java provides synchronization mechanisms. One of the primary tools in a developer's arsenal is synchronization using monitor locks, which are intrinsic to every Java object. These locks ensure exclusive access to an object, establishing a *happens-before* relationship between threads vying for the same resource.

NOTE Traditional Java objects typically exhibit a one-to-one relationship with their associated monitors. However, the Java landscape is currently undergoing a radical transformation with projects like Lilliput, Valhalla, and Loom. These initiatives herald a paradigm shift in the Java ecosystem and established norms—from object header structures to the very essence of thread management—and pave the way for a new era of Java concurrency. Project Lilliput[8] aims to minimize overhead by streamlining Java object header overheads, which may lead to decoupling the object monitor from the Java object it synchronizes. Project Valhalla introduces value types, which are immutable and lack the identity associated with regular Java objects, thereby reshaping traditional synchronization models. Project Loom, while focusing on lightweight threads and enhancing concurrency, further signifies the evolving nature of Java's synchronization and concurrency practices. These projects redefine Java's approach to object management and synchronization, reflecting ongoing efforts to optimize the language for modern computing paradigms.

To illustrate Java's synchronization mechanisms, consider the following code snippet:

```
public void doActivity() {
     synchronized(this) {
          // Protected work inside the synchronized block
     }
     // Regular work outside the synchronized block
}
```

In this example, a thread must acquire the monitor lock on the instance object (`this`) to execute the protected work. Outside this block, no such lock is necessary.

[8] https://wiki.openjdk.org/display/lilliput

Synchronized methods offer a similar mechanism:

```
public synchronized void doActivity() {
     // Protected work inside the synchronized method
}
// Regular work outside the synchronized method
```

NOTE

- Synchronized blocks are essentially encapsulated synchronized statements.
- Instance-level synchronization is exclusive to nonstatic methods, whereas static methods necessitate class-level synchronization.
- The monitor of an instance object is distinct from the class object monitor for the same class.

The Role of Monitor Locks

In Java, threads synchronize their operations to communicate effectively and ensure data consistency. Central to this synchronization is the concept of monitor locks. A monitor lock, often simply referred to as a monitor, acts as a mutual exclusion lock, ensuring that at any given time only one thread can own the lock and execute the associated synchronized code block.

When a thread encounters a synchronized block, it attempts to acquire the monitor lock associated with the object being synchronized upon. If the lock is already held by another thread, the attempting thread is suspended and joins the object's *wait set*—a collection of threads queued up, waiting to acquire the object's lock. Every object in Java inherently has an associated wait set, which is empty upon the object's creation.

The monitor lock plays a pivotal role in ensuring that operations on shared resources are atomic, preventing concurrent access anomalies. Upon completing its operations within the synchronized block, the thread relinquishes the monitor lock. Before doing so, it can optionally call the object's `notify()` or `notifyAll()` method. The `notify()` method awakens a single suspended thread from the object's wait set, whereas `notifyAll()` wakes up all the waiting threads.

Lock Types in OpenJDK HotSpot VM

In the context of Java's locking mechanisms, locks are user space events, and such events show up as voluntary context switches to the operating system (OS). In consequence, the Java runtime optimizes and schedules the threads based on their state with respect to the locked object. At any given time, a lock can be contended or uncontended.

Lock Contention Dynamics

Lock contention occurs when multiple threads attempt to acquire a lock simultaneously. If a thread "t" is active within a synchronized block and another thread "u" tries to enter the same

block, the lock is uncontended if no other threads are waiting. However, once "u" attempts to enter and is forced to wait due to "t" holding the lock, contention arises. Java's locking mechanism is optimized to handle such scenarios, transitioning from a lightweight lock to a heavyweight lock when contention is frequent.

Uncontended and Deflated Locks

Uncontended and deflated locks are lightweight, stack-based locks optimized for scenarios without contention. The HotSpot VM implements a thin lock through a straightforward atomic *compare-and-swap* instruction (CAS). This CAS operation places a pointer to a lock record, generated on the thread's stack, into the object's header word. The lock record consists of the original header word and the actual pointer to the synchronized object—a mechanism termed a *displaced header*. This lightweight lock mechanism, integrated into the interpreter and JIT-compiled code, efficiently manages synchronization for objects accessed by a single thread. When contention is detected, this lock is inflated to a heavyweight lock.

> **NOTE** The intricacies of atomic instructions, especially CAS, were discussed in Chapter 5, "End-to-End Java Performance Optimization: Engineering Techniques and Micro-benchmarking with JMH."

Contended and Inflated Locks

Contended and inflated locks are heavyweight locks that are activated when contention occurs among threads. When a thin lock experiences contention, the JVM inflates it to a heavyweight lock. This transition necessitates more intricate interactions with the OS, especially when parking contending threads and waking up parked threads. The heavyweight lock uses the OS's native mutex mechanisms to manage contention, ensuring that the synchronized block is executed by only one thread at a time.

Code Example and Analysis

Consider the following code snippet, which demonstrates the use of monitor locks and the interplay of the `wait()` and `notifyall()` methods.

```
public void run() {
    // Loop for a predefined number of iterations
    for (int j = 0; j < MAX_ITERATIONS; j++) {

        // Synchronize on the shared resource to ensure thread safety
        synchronized(sharedResource) {

            // Check if the current thread type is DECREASING
            if(threadType == ThreadType.DECREASING) {

                // Wait until the counter is not zero
                while (sharedResource.getCounter() == 0) {
```

```
                try {
                    // Wait until another thread gets to notify() or notifyAll()for the object
                    sharedResource.wait();
                } catch (InterruptedException e) {
                    // Handle the interrupted exception and exit the thread
                    Thread.currentThread().interrupt();
                    return;
                }
            }

            // Decrement the counter of the shared resource
            sharedResource.decrementCounter();

            // Print the current status (this method would provide more details about the
            current state)
            printStatus();
        }
        // If the current thread type is not DECREASING (i.e., INCREASING)
        else {

            // Wait until the counter is not at its maximum value (e.g., 10)
            while (sharedResource.getCounter() == 10) {
                try {
                    // Wait until another thread gets to notify() or notifyAll()for the object
                    sharedResource.wait();
                } catch (InterruptedException e) {
                    // Handle the interrupted exception and exit the thread
                    Thread.currentThread().interrupt();
                    return;
                }
            }

            // Increment the counter of the shared resource
            sharedResource.incrementCounter();

            // Print the current status (this method would provide more details about the
            current state)
            printStatus();
        }

        // Notify all threads waiting on this object's monitor to wake up
        sharedResource.notifyAll();
    }
  }
}
```

The code takes the following actions:

- **Synchronization:** The `synchronized(sharedResource)` block ensures mutual exclusion. Only one thread can execute the enclosed code at any given time. If another thread attempts to enter this block while it's occupied, it's forced to wait.
- **Counter management:**
  - For threads of type `DECREASING`, the counter of the `sharedResource` is checked. If it's zero, the thread enters a waiting state until another thread modifies the counter and sends a notification. Once awakened, it decrements the counter and prints the status.
  - Conversely, for threads of type `INCREASING`, the counter is checked against its maximum value (ten). If the counter is maxed out, the thread waits until another thread modifies the counter and sends a notification. Once notified, it increments the counter and prints the status.
- **Notification:** After modifying the counter, the thread sends a wake-up signal to all other threads waiting on the `sharedResource` object's monitor using the `sharedResource.notifyAll()` method.

Advancements in Java's Locking Mechanisms

Java's locking mechanisms have seen significant refinements over the years, both at the API level and in the HotSpot VM. This section provides a structured overview of these enhancements up to Java 8.

API-Level Enhancements

- **`ReentrantLock` (Java 5):** `ReentrantLock`, which is part of the `java.util.concurrent.locks` package, offers more flexibility than the intrinsic locking provided by the `synchronized` keyword. It supports features such as timed lock waits, interruptible lock waits, and the ability to create fair locks.

  Thanks to `ReentrantLock`, we have `ReadWriteLock`, which uses a pair of locks: one lock for supporting concurrent read-only operations and another exclusive lock for writing.

> **NOTE** Although `ReentrantLock` offers advanced capabilities, it's not a direct replacement for `synchronized`. Developers should weigh the benefits and trade-offs before choosing between them.

- **`StampedLock` (Java 8):** `StampedLock` provides a mechanism for optimistic reading, which can improve throughput under the condition of high contention. It supports both read and write locks, like `ReentrantReadWriteLock`, but with better scalability. Unlike other locks, `StampedLock` does not have an "owner." This means any thread can release a lock, not just the thread that acquired it. Careful management to avoid potential issues is critical when `StampedLock` is used.

NOTE `StampedLock` doesn't support reentrancy, so care must be taken to avoid deadlocks.

HotSpot VM Optimizations

- **Biased locking (Java 6):** Biased locking was a strategy aimed at improving the performance of uncontended locks. With this approach, the JVM employed a single atomic CAS operation to embed the thread ID in the object header, indicating the object's bias toward that specific thread. This eliminated the need for atomic operations in subsequent accesses by the biased thread, providing a performance boost in scenarios where the same thread repeatedly acquires a lock.

NOTE Biased locking was introduced to enhance performance, but it brought complexities to the JVM. One significant challenge was the unexpected latencies during bias revocation, which could lead to unpredictable performance in certain scenarios. Given the evolution of modern hardware and software patterns, which diminished the advantages of biased locking, it was deprecated in JDK 15, paving the way for more consistent and predictable synchronization techniques. If you try to use it in versions later than Java 15, the following warning will be generated:

```
Java HotSpot™ 64-Bit Server VM warning: Option UseBiasedLocking was deprecated in
version 15.0 and will likely be removed in a future release.
```

- **Lock elision:** This technique harnesses the JVM's escape analysis optimization. The JVM checks whether the lock object "escapes" from its current scope—that is, whether it's accessed outside the method or thread where it was created. If the lock object remains confined, the JVM can eliminate the lock and associated overhead, retaining the field value in a register (scalar replacement).

  Example: Consider a method in which an object is created, locked, modified, and then unlocked. If this object doesn't escape the method, the JVM can elide the lock.

- **Lock coarsening:** We often encounter code with multiple synchronized statements. However, these could be locking on the same object. Rather than issuing multiple locks, the JVM can combine these locks into a single, coarser lock, reducing the locking overhead. The JVM applies this optimization to both contended and uncontended locks.

  Example: In a loop where each iteration has a synchronized block locking on the same object, the JVM can coarsen these into one lock outside the loop.

- **Adaptive spinning (Java 7):** Prior to Java 7, threads that encountered a locked monitor would immediately enter a wait state. The introduction of adaptive spinning allowed a thread to briefly spin (busy-wait or spin-wait) in anticipation of the lock being released soon. This improvement significantly boosted performance on systems with multiple processors, where a thread might need to wait for only a short amount of time before acquiring a lock.

Although these are certainly some of the significant improvements related to locking, numerous other enhancements, bug fixes, and performance tweaks related to synchronization and concurrency have appeared throughout the various Java releases. For readers interested in gaining comprehensive insights, please refer to the official Java documentation.[9, 10]

Optimizing Contention: Enhancements since Java 9

Contended locking is instrumental in ensuring the smooth operation of multithreaded Java applications. Over the years, Java's evolution has consistently focused on enhancing its contended locking mechanisms. Java 9, inspired by JEP 143: *Improve Contended Locking*,[11] introduced a series of pivotal improvements, which have been further refined in subsequent versions. Among the key enhancements, the acceleration of Java *monitor enter* operations and the efficiency of *monitor exit* operations stand out. These operations are foundational to synchronized methods and blocks in Java.

To appreciate the significance of *monitor enter* and *monitor exit* operations, let's revisit some foundational concepts. We've previously touched upon the `notify` and `notifyAll` operations in the context of contended locks. However, the intricacies of monitor operations—which are at the heart of synchronized method invocation—might be less familiar to some readers. For instance, in our synchronized statement example, these calls operate behind the scenes. A glimpse at the bytecode generated with `javap -c` for the `sharedResource` offers clarity:

```
public void run();
 Code:
 ...
  9: getfield #9 // Field sharedResource:LReservationSystem$SharedResource;
 12: dup
 13: astore_2
 14: monitorenter
 15: aload_0
 ...
 52: aload_2
 53: monitorexit
 ...
```

[9] https://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html

[10] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/locks/package-summary.html

[11] https://openjdk.org/jeps/143

In essence, `monitorenter` and `monitorexit` handle the acquisition and release of a lock on an object during execution of a synchronized method or block. When a thread encounters a `wait()` call, it transitions to the wait queue. Only when a `notify()` or a `notifyAll()` call is made does the thread move to the entry queue, potentially regaining the lock (Figure 7.19).

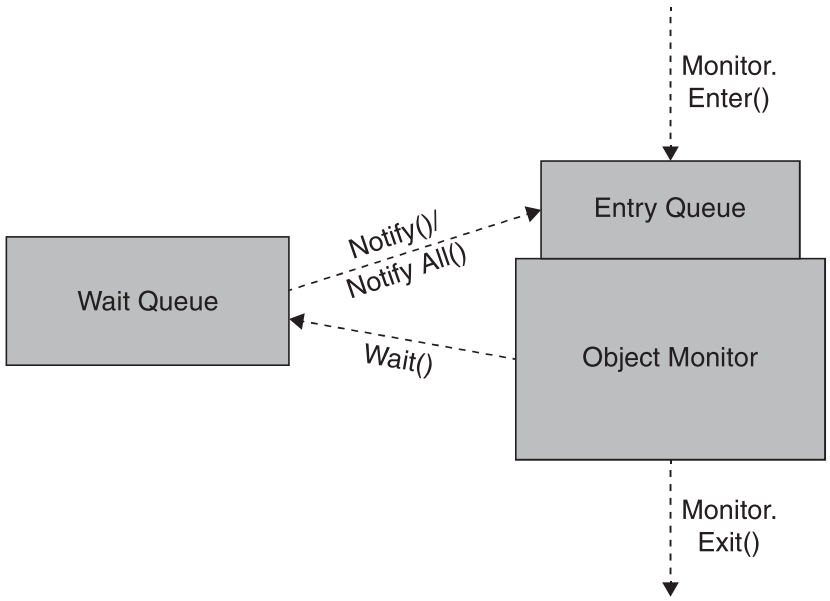

Figure 7.19 Object's Monitor Wait and Entry Queue

Improved Java Monitor Enter and Fast Exit Operations

Contended locking is a resource-intensive operation, and any enhancement that reduces the time spent in contention directly impacts the efficiency of Java applications. Over time, the JVM has been meticulously refined to improve synchronization performance, with notable advancements from Java 8 to Java 17. A critical aspect of JVM performance engineering involves unraveling and understanding these runtime enhancements.

In Java 8, entering an inflated contended lock involves the `InterpreterRuntime::monitorenter` method (Figure 7.20). If the lock isn't immediately available, the slower `ObjectSynchronizer::slow_enter` method is invoked. The introduction of enhancements through JEP 143 has refined this process, with improvements clearly evident by Java 17. The `ObjectSynchronizer::enter` method in Java 17 optimizes frequently used code paths—for example, by setting the *displaced header* to a specific value it indicates the type of lock in use. This adjustment provides a streamlined path for the efficient management of common lock types, as illustrated by the "optimized ObjectSynchronizer::enter" route in Figure 7.21.

The process of exiting a lock has also undergone optimization. Java 8's `InterpreterRuntime::monitorexit` can defer to slower exit routines, while Java 17's optimized `ObjectSynchronizer::exit` method ensures a faster path to releasing locks.

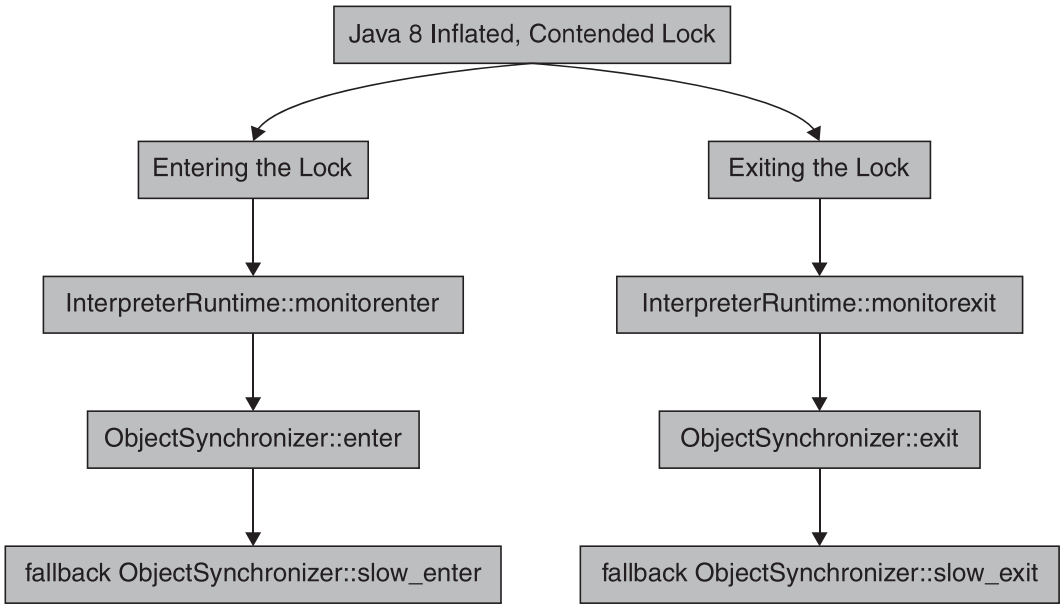

Figure 7.20 High-Level Overview of Contended Lock Enter and Exit Call Paths for Java 8

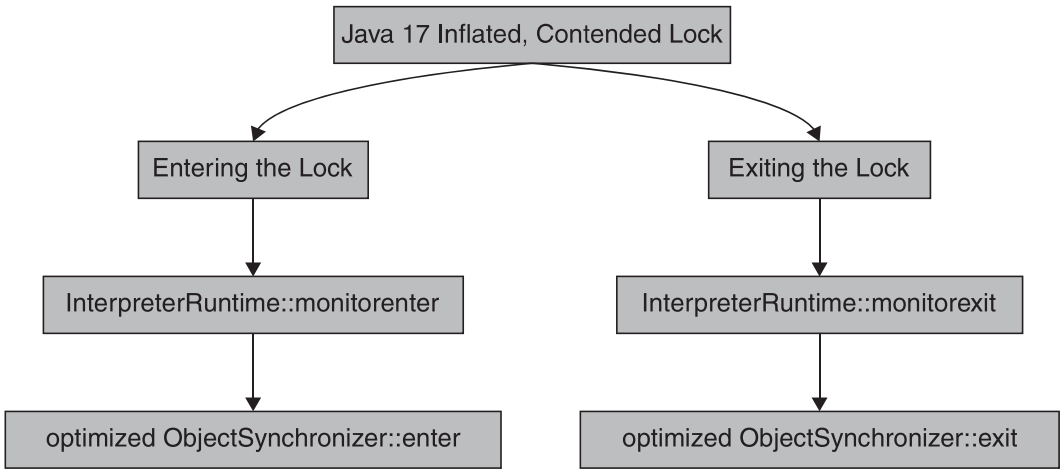

Figure 7.21 High-Level Overview of Contended Lock Enter and Exit Call Paths for Java 17

Further enhancements in JDK 17 pertain to the acceleration of Java's `monitor wait` and `notify/notifyAll` operations. These operations use a `quick_notify` routine, which efficiently manages threads in the wait queue and ensures they move to the monitor enter queue promptly, thereby eliminating the need for slow fallback methods.

Visualizing Contended Lock Optimization: A Performance Engineering Exercise

JVM performance engineering isn't just about identifying bottlenecks—it enables us to gather a granular understanding of the enhancements that can yield runtime efficiency gains. Hence,

This section will undertake a hands-on exploration to illustrate the performance improvements in Java 17's handling of contended locks compared to Java 8.

We'll apply the performance engineering methodologies discussed in Chapter 5, "End-to-End Java Performance Optimization: Engineering Techniques and Micro-benchmarking with JMH," aiming to demystify the approaches by leveraging profilers and the JMH benchmarking tool.

Our exploration centers on two pivotal aspects:

- **Performance engineering in action:** A practical illustration of tackling performance issues, from problem identification to the application of benchmarking and profiling for in-depth analysis.
- **Theory to practice:** Bridging the concepts from Chapter 5 to their real-world application and to demonstrate contended lock optimization.

Additionally, we will explore A/B performance testing, a targeted experimental design[12] approach that aims to compare the performance outcomes of two JDK versions. A/B performance testing is a powerful technique to empirically verify performance enhancements, ensuring that any changes introduced lead to tangible benefits.

In our exploration of contended lock optimization, we'll harness the power of the Java Micro-benchmarking Harness (JMH). As discussed in Chapter 5, JMH is a powerful tool for performance benchmarking. Here's a micro-benchmark tailored to evaluate recursive lock and unlock operations for synchronized blocks:

```
/** Perform recursive synchronized operations on local objects within a loop. */
@Benchmark
public void testRecursiveLockUnlock() {
    Object localObject = lockObject1;
    for (int i = 0; i < innerCount; i++) {
        synchronized (localObject) {
            synchronized (localObject) {
                dummyInt1++;
                dummyInt2++;
            }
        }
    }
}
```

The `testRecursiveLockUnlock` method is designed to rigorously assess the performance dynamics of contended locks, specifically the `monitorenter` and `monitorexit` operations. The method does this by performing `synchronized` operations on a local object (`localObject`) in a nested loop. Each iteration of the loop results in the `monitorenter` operation being called twice (due to the nested synchronized blocks) and the `monitorexit` operation also being called twice when exiting the synchronized blocks. For those keen on exploring further, this and other lock/unlock micro-benchmarks are available in the Code Tools repository

[12] See Chapter 5 for more information.

for JMH JDK Micro-benchmarks on GitHub,[13] all thanks to the OpenJDK community of contributors.

Finally, to enrich our analysis, we'll incorporate *async-profiler*.[14] This profiler will be utilized in both the *flamegraph* and *call tree* modes. The *call tree* mode offers valuable granular insights into interactions, shedding light on the specific paths chosen during execution.

Setup and Data Collection

Our primary objective is to unravel the depths of the contended locking improvements and gauge the performance enhancements they introduce. For this, we need a meticulous setup and data collection process. By using Java 8 as our performance baseline, we can draw insightful comparisons with its successor, Java 17. Here's a comprehensive overview of the exercise setup:

1. **Environment setup:**
   - Ensure you have both JDK 8 and JDK 17 installed on your system. This dual setup allows for a side-by-side performance evaluation.
   - Install JMH to make performance measurements and *async-profiler* to take a deep dive into the Java runtime. Familiarize yourself with *async-profiler*'s command-line options.
2. **Repository cloning:**
   - Clone the JMH JDK Micro-benchmarks repository from GitHub. This repo is a treasure trove of micro-benchmarks tailored for JDK enhancements and measurements.
   - Navigate to the `src` directory. Here you will find the `vm/lang/LockUnlock.java` benchmark, which will be our primary tool for this exploration.
3. **Benchmark compilation:**
   - Compile the benchmark using the Maven project management tool. Ensure that you specify the appropriate JDK version to maintain version consistency.
   - After compilation is complete, verify and validate the benchmark's integrity and readiness for execution.

Once you have performed the preceding steps, you will have a robust setup for conducting tests and analyzing the contended locking improvements and their impact on JVM performance.

Testing and Performance Analysis

To derive a comprehensive and precise understanding of the JVM's contended locking performance, it's imperative to follow a structured approach with specific configurations. This section highlights the steps and considerations essential for an accurate performance analysis.

[13] https://github.com/openjdk/jmh-jdk-microbenchmarks

[14] https://github.com/async-profiler/async-profiler

Benchmarking Configuration

Having selected the `RecursiveLockUnlock` benchmark for our study, our aim is to evaluate both monitor entry and exit pathways under consistent conditions, minimizing any external or internal interferences. Here's a refined set of recommendations:

- **Consistency in benchmark selection:** The `LockUnlock.testRecursiveLockUnlock` benchmark aligns with the specific objective of measuring both the monitor enter and exit pathways. This ensures that the results are directly relevant to the performance aspect under scrutiny.
- **Neutralizing biased locking:** Since Java 8 incorporates the biased locking optimization, disable it using the `-XX:-UseBiasedLocking` JVM option to negate its potential impact on our results. This step is crucial to ensure unbiased results.
- **Exclude stack-based locking:** Incorporate `-XX:+UseHeavyMonitors` in the JVM arguments. This precaution ensures that the JVM will refrain from leveraging optimized locking techniques, such as stack-based locking, which could skew the results.
- **Sufficient warm-up:** The essence of the benchmark is the repetitive lock and unlock operations. Therefore, it's important to sufficiently warm up the micro-benchmark runs, ensuring that the JVM operates at its peak performance. Additionally, segregate the timed measurements from the profiling runs to account for the inherent overhead of profiling.
- **Consistent garbage collection (GC) strategy:** As highlighted in Chapter 6, "Advanced Memory Management and Garbage Collection in OpenJDK," Java 17 defaults to the G1 GC. For a fair comparison, match the GC strategy across versions by configuring Java 17 to use the throughput collector, Java 8's default. Although GC might not directly influence this micro-benchmark, it's vital to maintain a uniform performance benchmarking environment.

NOTE Warm-up iterations prime the JVM for accurate performance testing, eliminating anomalies from JIT compilation, class loading, and other JVM optimizations.

Benchmarking Execution

To accurately gauge the performance of contended JVM locking, execute the `testRecursive LockUnlock` benchmark within the JMH framework for both Java 8 and Java 17. Save the output to a file for a detailed analysis. The command line for execution would look something like this:

```
jmh-micros mobeck$ java -XX:+UseHeavyMonitors -XX:+UseParallelGC -XX:-UseBiasedLocking -jar
micros-jdk8/target/micros-jdk8-1.0-SNAPSHOT.jar -wi 50 -i 50 -f 1 .RecursiveLockUnlock.* 2>&1 |
tee <logfilename.txt>
```

In the preceding line, note the following options:

- **`-wi 50`:** Specifies the number of warm-up iterations.
- **`-i 50`:** Specifies the number of measurement iterations.
- **`-f 1`:** Specifies the number of forks. Forks create separate JVM processes to ensure results aren't skewed by JVM optimizations. Since we are already maximizing the CPU resources, we will work with just one fork.

For the rest of the setup, use the default values in the JMH suite. After running the benchmark and capturing the results, we can initiate the profiler measurements. For *flamegraph* mode, use the following command:

```
async-profiler-2.9 mobeck$ ./profiler.sh -d 120 -j 50 -a -t -s -f <path-to-logs-dir>/<filename>.
html <pid-of-RecursiveLockUnlock-process>
```

- **`-d 120`:** Sets the profiling duration to 120 seconds.
- **`-j 50`:** Samples every 50ms.
- **`-a`:** Profiles all methods.
- **`-t`:** Adds thread names.
- **`-s`:** Simplifies class names.
- **`-f`:** Specifies the output file.

For a call tree breakdown, add `-o tree` to organize data into a tree structure.

Comparing Performance Between Java 8 and Java 17

With the profiling data for Java 8 and Java 17 at hand, we move forward to contrast their performance. Open the generated HTML files for both versions, examining the outputs for the *call tree* and *flamegraph* modes side-by side to discern differences and improvements.

Flamegraph Analysis

A *flamegraph* provides a visual representation of call stacks, which helps identify performance bottlenecks. The *flamegraph* for Java 17 (Figure 7.22), is comparable to Java 8's profile.

As you drill down deeper into the *flamegraph* view, hovering over each section will reveal the percentage of total samples for each method in the call stack. Figures 7.23 and 7.24 provide zoomed-in views of specific call stacks and their associated methods.

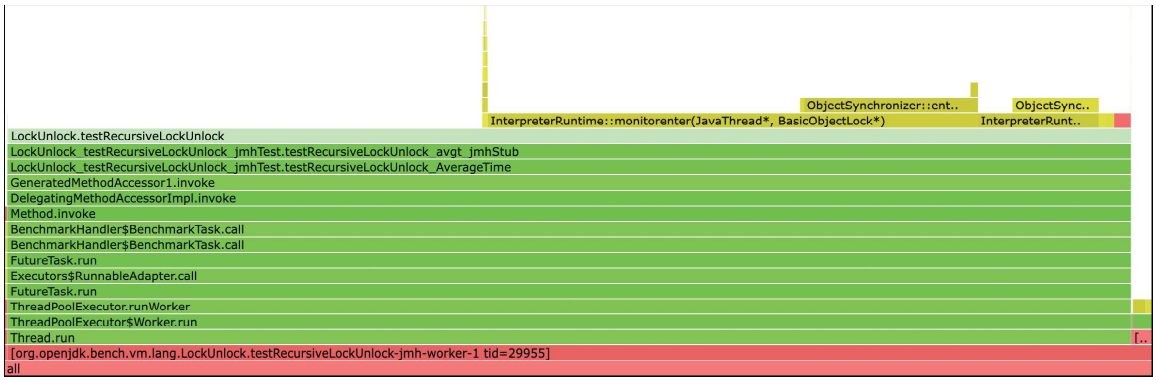

Figure 7.22 Zoomed-in Call Stack of the RecursiveLockUnLock Test on Java 17

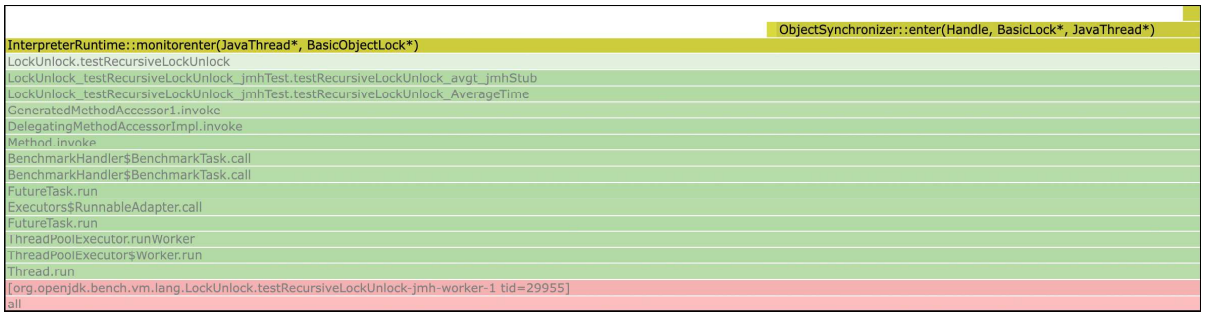

Figure 7.23 Zooming Further into Java 17's Monitor Enter Stack

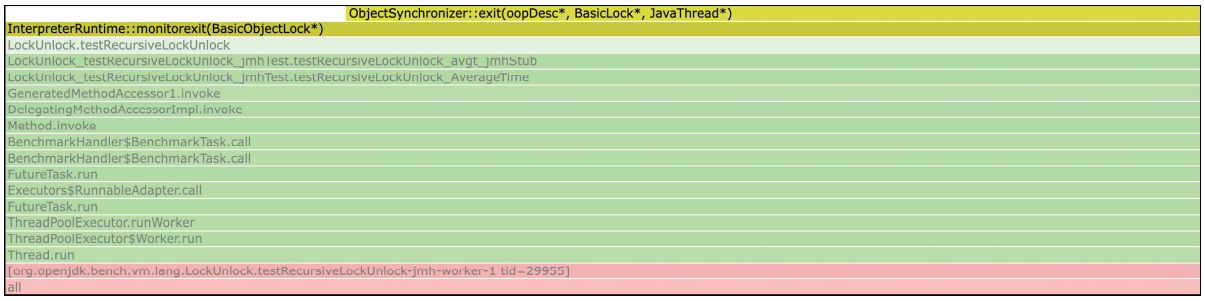

Figure 7.24 Zoomed-in View of Java 17's Monitor Exit Stack

Contrast them with Figures 7.25 and 7.26, Java 8's monitor enter and exit stacks, respectively.

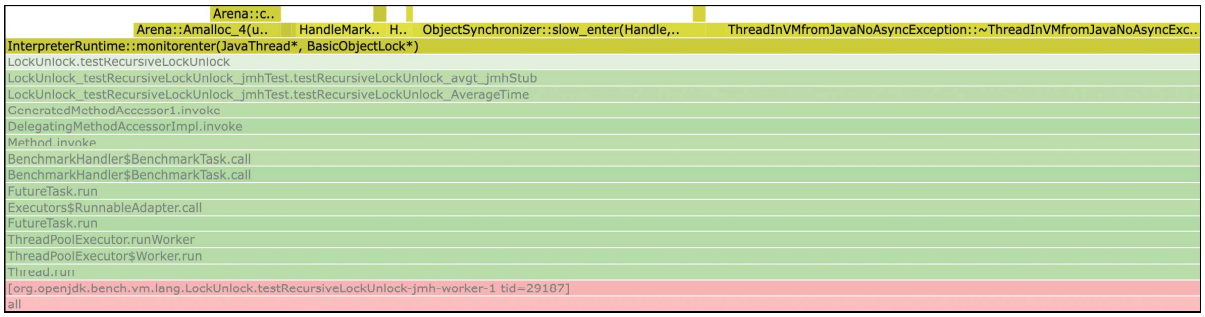

Figure 7.25 Zoomed-in View of Java 8's Monitor Enter Stack

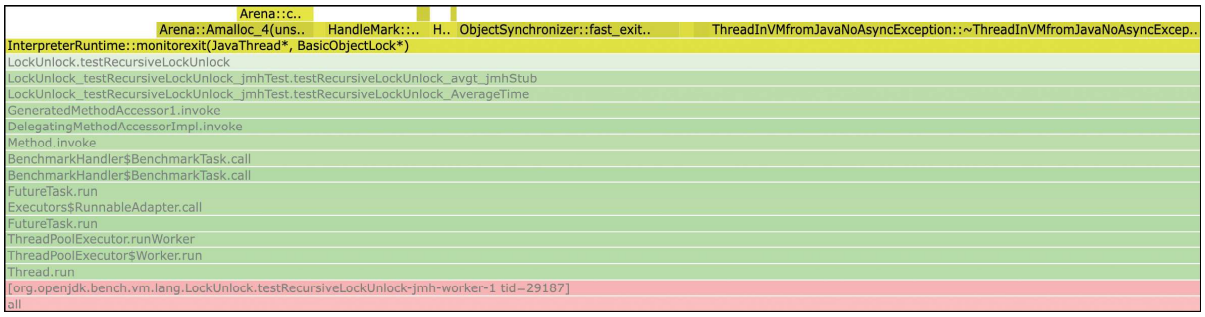

Figure 7.26 Zoomed-in View of Java 8's Monitor Exit Stack

Benchmarking Results

Table 7.1 showcases Java 17's enhanced performance over Java 8 on the `testRecursiveLockUnlock` benchmark.

The **Score (ns/op)** column represents the time taken per operation in nanoseconds. A lower score indicates better performance. Additionally, the **Error (ns/op)** column represents the variability in the score. A higher error indicates more variability in performance.

Table 7.1 **JMH Results for Java 17 Versus Java 18**

Java Version	Benchmark Test	Mode	Count	Score (ns/op)	Error (ns/op)
Java 17	`LockUnlock.testRecursiveLockUnlock`	avgt	50	7002.294	±86.289
Java 8	`LockUnlock.testRecursiveLockUnlock`	avgt	50	11,198.681	±27.607

Java 17 has a significantly lower score compared to Java 8, indicating better performance via higher efficiency in handling contended locks—an affirmation of Java 17's advancements in lock optimization. This improvement is evident not just in micro-benchmarks, but also translates to real-world applications, ensuring smoother and faster execution in scenarios with contended locks.

Call Tree Analysis

While *flamegraphs* are excellent for a high-level overview of performance bottlenecks, *async-profiler's call tree* map offers the detail needed for deeper analysis. It dissects the program's function calls, aiding developers in tracing the execution paths and pinpointing performance improvements in Java 17 over Java 8.

Figures 7.27, 7.28, and 7.29 visualize the call trees for both Java versions, illustrating the monitor entry and exit paths. These visuals guide us through the intricacies of contended locking paths and showcase where Java 17's optimizations shine.

```
- [14] 97.69% 9,648 self: 39.04% 3,856 LockUnlock.testRecursiveLockUnlock
  - [15] 44.52% 4,397 self: 28.24% 2,789 InterpreterRuntime::monitorenter(JavaThread*, BasicObjectLock*)
    + [16] 15.92% 1,572 self: 15.29% 1,510 ObjectSynchronizer::enter(Handle, BasicLock*, JavaThread*)
      [16] 0.36% 36 self: 0.36% 36 JavaThread::is_lock_owned(unsigned char*) const
  - [15] 10.74% 1,061 self: 3.16% 312 InterpreterRuntime::monitorexit(BasicObjectLock*)
      [16] 7.58% 749 self: 7.58% 749 ObjectSynchronizer::exit(oopDesc*, BasicLock*, JavaThread*)
    [15] 1.51% 149 self: 1.51% 149 tlv_get_addr
    [15] 0.74% 73 self: 0.74% 73 ObjectSynchronizer::enter(Handle, BasicLock*, JavaThread*)
    [15] 0.58% 57 self: 0.58% 57 ObjectSynchronizer::exit(oopDesc*, BasicLock*, JavaThread*)
  + [15] 0.56% 55 self: 0.00% 0 InterpreterRuntime::frequency_counter_overflow(JavaThread*, unsigned char*)
```

Figure 7.27 Java 17's Call Tree View for the testRecursiveLockUnlock's monitorenter and monitorexit Routines

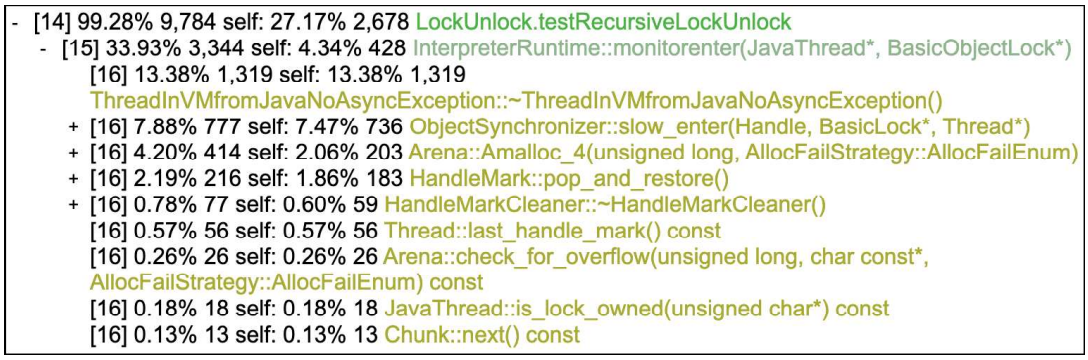

Figure 7.28 Java 8's Call Tree View for the testRecursiveLockUnlock's monitorenter Routine

Figure 7.29 Java 8's Call Tree View for the testRecursiveLockUnlock's monitorexit Routine

Detailed Comparison

Comparing call trees between Java 8 and Java 17 reveals underlying performance differences crucial for optimization. Key elements from the call tree excerpts to consider are

- **Call hierarchy:** Indicated by indentation and bracketed numbers, depicting the sequence of method calls.
- **Inclusive and exclusive samples:** Represented by percentages and numbers following the brackets, with "self" indicating exclusive samples for methods.
- **Profiling time:** Percentages are derived from the total samples over a profiling period of 120 seconds to capture the full performance spectrum.

Excerpts from Java 17's call tree show:

```
…
[14] 97.69% 9,648 self: 39.04% 3,856 LockUnlock.testRecursiveLockUnlock
   [15] 44.52% 4,397 self: 28.24% 2,789 InterpreterRuntime::monitorenter(JavaThread*,
   BasicObjectLock*)
   [15] 10.74% 1,061 self: 3.16% 312 InterpreterRuntime::monitorexit(BasicObjectLock*)
   [15] 1.51% 149 self: 1.51% 149 tlv_get_addr
   [15] 0.74% 73 self: 0.74% 73 ObjectSynchronizer::enter(Handle, BasicLock*, JavaThread*)
   [15] 0.58% 57 self: 0.58% 57 ObjectSynchronizer::exit(oopDesc*, BasicLock*, JavaThread*)
```

```
   [15] 0.56% 55 self: 0.00% 0 InterpreterRuntime::frequency_counter_overflow(JavaThread*,
   unsigned char*)
…
```

Inclusive time for `monitorenter` and `monitorexit` =

9,648 (inclusive time for `testRecursiveLockUnLock`)

- 3,856 (exclusive time for `testRecursiveLockUnLock`)
- 149 (inclusive time for `tlv_get_addr`)
- 73 (inclusive time for `ObjectSynchronizer::enter`)
- 57 (inclusive time for `ObjectSynchronizer::exit`)
- 55 (inclusive time for `InterpreterRuntime::frequency_counter_overflow`)

= 5,458 samples

= 4397 (inclusive time for `monitorenter`) + 1061 (inclusive time for `monitorexit`)

Excerpts from Java 8's call tree show:

```
…
[14] 99.28% 9,784 self: 27.17% 2,678 LockUnlock.testRecursiveLockUnlock
   [15] 33.93% 3,344 self: 4.34% 428 InterpreterRuntime::monitorenter(JavaThread*,
   BasicObjectLock*)
   [15] 33.75% 3,326 self: 4.64% 457 InterpreterRuntime::monitorexit(JavaThread*,
   BasicObjectLock*)
   [15] 1.42% 140 self: 1.42% 140 HandleMarkCleaner::~HandleMarkCleaner()
   [15] 0.74% 73 self: 0.74% 73
   ThreadInVMfromJavaNoAsyncException::~ThreadInVMfromJavaNoAsyncException()
   [15] 0.56% 55 self: 0.56% 55 HandleMark::pop_and_restore()
   [15] 0.44% 43 self: 0.44% 43 ObjectSynchronizer::slow_exit(oopDesc*, BasicLock*, Thread*)
   [15] 0.43% 42 self: 0.43% 42 ObjectSynchronizer::slow_enter(Handle, BasicLock*, Thread*)
   [15] 0.43% 42 self: 0.43% 42 Arena::Amalloc_4(unsigned long, AllocFailStrategy::AllocFailEnum)
   [15] 0.42% 41 self: 0.00% 0 InterpreterRuntime::frequency_counter_overflow(JavaThread*,
   unsigned char*)
…
```

Inclusive time for `monitorenter` and `monitorexit` =

9,784 (inclusive time for `testRecursiveLockUnLock`)

- 2,678 (exclusive time for `testRecursiveLockUnLock`)
- 140 (inclusive time for `HandleMarkCleaner::~HandleMarkCleaner`)
- 73 (inclusive time for `ThreadInVMfromJavaNoAsyncException`)
- 55 (inclusive time for `HandleMark::pop_and_restore`)
- 43 (inclusive time for `ObjectSynchronizer::slow_exit`)
- 42 (inclusive time for `ObjectSynchronizer::slow_enter`)
- 42 (inclusive time for `Arena::Amalloc_4`)
- 41 (inclusive time for `InterpreterRuntime::frequency_counter_overflow`)

= 6670 samples

= 3344 (inclusive time for `monitorenter`) + 3326 (inclusive time for `monitorexit`)

These textual excerpts, derived from the *call tree* HTML outputs, underscore the distinct performance characteristics of Java 17 and Java 8. The notable differences in the samples for the combined `monitorenter` and `monitorexit` operations signal better performance in Java 17.

Java 17 Versus Java 8: Interpreting the Data

From the runtimes and call tree, it's evident that Java 17 has made significant strides in optimizing the `LockUnlock.testRecursiveLockUnlock` method. Here's what we can tell:

Performance improvement in percent speed increase:

- Java 8 took approximately 11,198.681 ns/op.
- Java 17 took approximately 7002.294 ns/op.
- Improvement percentage = [(Old time – New time) / Old time] * 100
- Improvement percentage = [(11,198.681 – 7,002.294) / 11,198.681] * 100 ≈ 37.47%
- Java 17 is approximately 37.47% faster than Java 8 for this benchmark.

Let's see what we can learn from the test conditions, the call tree samples per method, and the performance measurements.

Consistent profiling conditions:

- Both Java 17 and Java 8 were profiled under the same benchmarking conditions during the execution of the `LockUnlock.testRecursiveLockUnlock` method.
- The benchmark was set to measure the average time taken for each operation, with the results presented in nanoseconds.
- The `innerCount` parameter was statically set to 100, ensuring that the workload in terms of synchronized operations remained consistent across both versions.

The inclusive and exclusive sample counts indicate that the performance differences between the two Java versions influenced the number of times the method was executed during the profiling period.

Inclusive time for monitor entry and exit:

- Java 17 spent 5458 samples (4397 for `monitorenter` + 1061 for `monitorexit`) in inclusive time for monitor operations.
- Java 8 spent 6670 samples (3344 for `monitorenter` + 3326 for `monitorexit`) in inclusive time for monitor operations.
- This indicates that Java 8 spent more time in monitor operations than Java 17.

Exclusive time in `testRecursiveLockUnlock` method:

- Java 17's exclusive time: 3856 samples
- Java 8's exclusive time: 2678 samples

- The exclusive time represents the time spent in the `testRecursiveLockUnlock` method outside of the `monitorenter` and `monitorexit` operations. The higher exclusive time in Java 17 indicates that Java 17 is more efficient in executing the actual work within the `testRecursiveLockUnlock` method, which is the increment operations on `dummyInt1` and `dummyInt2` inside the synchronized blocks.

Table 7.2 shows the extra methods that appear in the `monitorenter` call tree for Java 8:

- From the *call trees*, we can see that Java 8 has more method calls related to locking and synchronization. This includes methods like `ObjectSynchronizer::slow_enter`, `ThreadInVMfromJavaNoAsyncException::~ThreadInVMfromJavaNoAsync Exception`, and others.
- Java 8 also has deeper call stacks for these operations, indicating more overhead and complexity in handling locks.

Table 7.2 **Extra Methods for monitorenter in Java 8**

Stack Depth	Method Name
16	`ThreadInVMfromJavaNoAsyncException:: ~ThreadInVMfromJavaNoAsyncException()`
16	`ObjectSynchronizer::slow_enter(Handle, BasicLock*, Thread*)`
16	`Arena::Amalloc_4(unsigned long, AllocFailStrategy::AllocFailEnum)`
16	`HandleMark::pop_and_restore()`
16	`Thread::last_handle_mark() const`
16	`Arena::check_for_overflow(unsigned long, char const*, AllocFailStrategy::AllocFailEnum) const`
16	`JavaThread::is_lock_owned(unsigned char*) const`
16	`Chunk::next() const`

Similarly, for `monitorexit`, we have a lot more and a lot deeper stacks for Java 8, as shown in Table 7.3.

Table 7.3 **Extra Methods for monitorexit in Java 8**

Stack Depth	Method Name
16	`ThreadInVMfromJavaNoAsyncException:: ~ThreadInVMfromJavaNoAsyncException()`
16	`ObjectSynchronizer::fast_exit(oopDesc*, BasicLock*, Thread*)`
16	`Arena::Amalloc_4(unsigned long, AllocFailStrategy::AllocFailEnum)`

Stack Depth	Method Name
16	`HandleMark::pop_and_restore()`
16	`HandleMarkCleaner::~HandleMarkCleaner()`
16	`ObjectSynchronizer::slow_exit(oopDesc*, BasicLock*, Thread*)`
16	`Arena::check_for_overflow(unsigned long, char const*, AllocFailStrategy::AllocFailEnum) const`
16	`Thread::last_handle_mark() const`

Java 17 optimizations:

- Java 17 has fewer method calls related to locking, indicating optimizations in the JVM's handling of synchronization with less time in common methods.
- The reduced depth and fewer calls in the call tree for Java 17 suggest that the JVM has become more efficient in managing locks, reducing the overhead and time taken for these operations.

Table 7.4 shows Java 17's `monitorenter` stack, and Table 7.5 shows its `monitorexit` stack, which has just one entry.

Table 7.4 **Java 17 monitorenter Stack**

Stack Depth	Method Name
16	ObjectSynchronizer::enter(Handle, BasicLock*, JavaThread*)
16	JavaThread::is_lock_owned(unsigned char*) const

Table 7.5 **Java 17 monitorexit Stack**

Stack Depth	Method Name
16	ObjectSynchronizer::exit(oopDesc*, BasicLock*, JavaThread*)

Synthesizing Contended Lock Optimization: A Reflection

Our comprehensive analysis of Java 17's runtime has revealed substantial performance optimizations, particularly in contended lock management. Java 17 showcases a remarkable performance boost, outpacing Java 8 by about 37.47% in the `testRecursiveLockUnlock` benchmark—a testament to its efficiency and the effectiveness of its synchronization enhancements.

By leveraging tools like JMH for rigorous benchmarking and *async-profiler* for in-depth *call tree* analysis, the intricate work of JVM engineers comes to light. The call trees reveal a more streamlined execution path in Java 17, marked by fewer method invocations and a reduction in call stack depth, indicative of a more efficient lock management strategy when compared to Java 8.