2. **Data storage or preparation tasks:** Operations needed to store or prepare data for quick access and modifications for OLTP systems that handle a large number of transactions. Here, we'll identify critical data transfer and data feed paths to understand how data flows within the system.
3. **Data processing tasks:** The actual work patterns, such as sorting, structuring, and indexing, which are directly tied to the system's ability to execute transactions promptly. For these tasks, we'll identify critical data paths and transactions. We'll also pinpoint caches or other optimization techniques used to accelerate these operations.

For each category, we will gather the following information:

- Costly transactions and their paths
- Data transfer and data feed paths
- Libraries and their interactions
- Caches or other optimization techniques to accelerate transactions or data paths

By doing so, we can gain insights into where inefficiencies might lie, and which components might benefit from performance optimization. The categorized operations—data preprocessing tasks, data storage or preparation tasks, and data processing tasks—are integral parts of a database system. Enhancing these areas will help improve transaction latency (performance) and allow the system to handle an increasing number of transactions (scalability).

For example, identifying costly transactions (those that take a long time to process or that lock up other critical operations) and their paths can help streamline data preprocessing, reduce tail latencies, and improve the call chain traversal time. Similarly, optimizing data storage or preparation tasks can lead to quicker retrievals and efficient state management. Identifying critical data transfer paths and optimizing them can significantly increase system utilization during peak loads. Lastly, focusing on data processing tasks and optimizing the use of caches or other techniques can expedite transactions and improve overall system scalability.

With an understanding of the system-wide dynamics, we can now proceed to focus on specific components of the system. Our first stop: the JVM and its runtime environment.

JVM and Runtime Environment

The JVM serves as a bridge between our application and the container, OS, and hardware it runs on—which means that optimizing the JVM can result in significant performance gains. Here, we discuss the configuration options that most directly impact performance, such as JIT compiler optimizations, GC algorithms, and JVM tuning.

We'll start by identifying the JDK version, JVM command line, and GC deployment information. The JVM command line refers to the command-line options that can be used when starting a JVM. These options can control various aspects of JVM behavior, such as setting the maximum heap size or enabling various logging options, as elaborated in Chapter 4, "The Unified Java Virtual Machine Logging Interface." The GC deployment information refers to the configuration and operational details of the GC in use by the JVM, such as the type of GC, its settings, and its performance metrics.

Considering the repetitive and transaction-intensive nature of our OLTP system, it's important to pay attention to JIT compiler optimizations like method inlining and loop optimizations. These optimizations are instrumental in reducing method call overheads and enhancing loop execution efficiency, both of which are critical for processing high-volume transactions swiftly and effectively.

Our primary choice for a GC is the Garbage First Garbage Collector (G1 GC), owing to its proficiency in managing the short-lived transactional data that typically resides in the young generation. This choice aligns with our objective to minimize tail latencies while efficiently handling numerous transactions. In parallel, we should evaluate ZGC and keep a close watch on the developments pertaining to its generational variant, which promises enhanced performance suitable for OLTP scenarios.

Subsequently, we will gather the following data:

- GC logging to help identify GC thread scaling, transient data handling, phases that are not performing optimally, GC locker impact, time to safe points, and application stopped times outside of GC
- Thread and locking statistics to gain insights into thread locking and to get a better understanding of the concurrency behaviors of our system
- Java Native Interface (JNI) pools and buffers to ensure optimal interfacing with native code
- JIT compilation data on inline heuristics and loop optimizations, as they are pivotal for accelerating transaction processing

NOTE We'll gather profiling data separately from the monitoring data due to the potentially intrusive nature of profiling tools like Java Flight Recorder and VisualVM.

Garbage Collectors

When we talk about performance, we inevitably land on a vital topic: optimization of GCs. Three elements are crucial in this process:

- **Adaptive reactions:** The ability of the GC to adjust its behavior based on the current state of the system, such as adapting to higher allocation rates or increased long-lived data pressures.
- **Real-time predictions:** The ability of the GC to make predictions about future memory usage based on current trends and adjust its behavior accordingly. Most latency-geared GCs have this prediction capability.
- **Fast recovery**: The ability of the GC to quickly recover from situations where it is unable to keep up with the rate of memory allocation or promotion, such as by initiating an evacuation failure or a full garbage collection cycle.

Containerized Environments

Containers have become a ubiquitous part of modern application architecture, significantly influencing how applications are deployed and managed. In a containerized environment, JVM ergonomics takes on a new dimension of complexity. The orchestration strategies utilized, particularly in systems like Kubernetes, demand a nuanced approach to JVM resource allocation, aligning it with the performance objectives of our OLTP system. Here, we delve into the following aspects of containerized environments:

- **Container resource allocation and GC optimization:** We build upon the container orchestration strategies to ensure that the JVM's resource allocation is aligned with the OLTP system's performance objectives. This includes manual configurations to override JVM ergonomics that might not be optimal in a container setup.
- **Network-intensive JVM tuning:** Given the network-intensive nature of OLTP systems, we evaluate how container networking impacts JVM performance and optimize network settings accordingly.
- **Optimizing JVM's I/O operations:** We scrutinize the effects of I/O operations, particularly file handling by the JVM, to ensure efficient data access and transfer in our OLTP system.

OS Considerations

In this section, we analyze how the OS can influence the OLTP system's performance. We explore the nuances of process scheduling, memory management, and I/O operations, and discuss how these factors interact with our Java applications, especially in high-volume transaction environments. We'll employ a range of diagnostic tools provided by the OS to gather data at distinct levels and group the information based on the categories identified earlier.

For instance, when working with Linux systems, we have an arsenal of effective tools at our disposal:

- **vmstat:** This utility is instrumental in monitoring CPU and memory utilization, supplying a plethora of critical statistics:
  - **Virtual memory management:** A memory management strategy that abstracts the physical storage resources on a machine, creating an illusion of extensive main memory for the user. This technique significantly enhances the system's efficiency and flexibility by optimally managing and allocating memory resources according to needs.
  - **CPU usage analysis:** The proportion of time the CPU is operational, compared to when it is idle. A high CPU usage rate typically indicates that the CPU is frequently engaged in transaction processing tasks, whereas a low value may suggest underutilization or an idle state, often pointing toward other potential bottlenecks.

- **Sysstat tools**
  - **mpstat:** Specifically designed for thread- and CPU-level monitoring, *mpstat* is adept at gathering information on lock contention, context switches, wait time, steal time, and other vital CPU/thread metrics relevant to high-concurrency OLTP scenarios.
  - **iostat:** Focuses on input/output device performance, providing detailed metrics on disk read/write activities, crucial for diagnosing IO bottlenecks in OLTP systems.

These diagnostic tools not only provide a snapshot of the current state but also guide us in fine-tuning the OS to enhance the performance of our OLTP system. They help us understand the interaction between the OS and our Java applications at a deeper level, particularly in managing resources efficiently under heavy transaction loads.

Host OS and Hypervisor Dynamics

The host OS and hypervisor subsystem is the foundational layer of virtualization. The host OS layer is crucial for managing the physical resources of the machine. In OLTP systems, its efficient management of CPU, memory, and storage directly impacts overall system performance. Monitoring tools at this level help us understand how well the host OS is allocating resources to different VMs and containers.

The way the hypervisor handles CPU scheduling, memory partitioning, and network traffic can significantly affect transaction processing efficiency. Hence, to optimize OLTP performance, we analyze the hypervisor's settings and its interaction with both the host OS and the guest OS. This includes examining how virtual CPUs are allocated, understanding the memory overcommitment strategies, and assessing network virtualization configurations.

Hardware Subsystem

At the lowest level, we consider the actual physical hardware on which our software runs. This examination builds on our foundational knowledge of different CPU architectures and the intricate memory subsystems. We also look at the storage and network components that contribute to the OLTP system's performance. In this context, the concepts of concurrent programming and memory models we previously discussed become even more relevant. By applying effective monitoring techniques, we aim to understand resource constraints or identify opportunities to improve performance. This provides us with a complete picture of the hardware–software interactions in our system.

Thus, a thorough assessment of the hardware subsystem—covering CPU behavior, memory performance, and the efficiency of storage and network infrastructure—is essential. This approach helps by not only pinpointing performance bottlenecks but also illuminating the intricate relationships between the software and hardware layers. Such insights are crucial for implementing targeted optimizations that can significantly elevate the performance of our Java-centric OLTP database system.

To deepen our understanding and facilitate this optimization, we explore some advanced tools that bridge the gap between hardware assessment and actionable data: system profilers and performance monitoring units. These tools play a pivotal role in translating raw hardware metrics into meaningful insights that can guide our optimization strategies.

System Profilers and Performance Monitoring Units

Within the realm of performance engineering, system profilers—such as `perf`, OProfile, Intel's VTune Profiler, and other architecture-geared profilers—provide invaluable insights into hardware's functioning that can help developers optimize both system and application performance. They interact with the hardware subsystem to provide a comprehensive view that is unattainable through standard monitoring tools alone.

A key source of data for these profilers is the performance monitoring units (PMUs) within the CPU. PMUs, including both fixed-function and programmable PMUs, provide a wealth of low-level performance information that can be used to identify bottlenecks and optimize system performance.

System profilers can collect data from PMUs to generate a detailed performance profile. For example, the `perf` command can be used to sample CPU stack traces across the entire system, creating a "flat profile" that summarizes the time spent in each function across the entire system. This kind of performance profile can be used to identify functions that are consuming a disproportionate amount of CPU time. By providing both inclusive and exclusive times for each function, a flat profile offers a detailed look at where the system spends its time.

However, system profilers are not limited to CPU performance; they can also monitor other system components, such as memory, disk I/O, and network I/O. For example, tools like *iostat* and *vmstat* can provide detailed information about disk I/O and memory usage, respectively.

Once we have identified problematic patterns, bottlenecks, or opportunities for improvement through monitoring stats, we can employ these advanced profiling tools for deeper analysis.

Java Application Profilers

Java application profilers like *async-profiler*[12] (a low-overhead sampling profiler) and VisualVM (a visual tool that integrates several command-line JDK tools and provides lightweight profiling capabilities) provide a wealth of profiling data on JVM performance. They can perform different types of profiling, such as CPU profiling, memory profiling, and thread profiling. When used in conjunction with other system profilers, these tools can provide a more complete picture of application performance, from the JVM level to the system level.

By collecting data from PMUs, profilers like *async-profiler* can provide low-level performance information that helps developers identify bottlenecks. They allow developers to get stack and generated code-level details for the workload, enabling them to compare the time spent in the generated code and compiler optimizations for different architectures like Intel and Arm.

To use *async-profiler* effectively, we need *hsdis*,[13] a disassembler plug-in for OpenJDK HotSpot VM. This plug-in is used to disassemble the machine code generated by the JIT compiler, offering a deeper understanding of how the Java code is being executed at the hardware level.

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

[13] https://blogs.oracle.com/javamagazine/post/java-hotspot-hsdis-disassembler

Key Takeaways

In this section, we applied a top-down approach to understand and enhance the performance of our Java-centric OLTP system. We started at the highest system level, extending through the complex ecosystem of middleware, APIs, and storage engines. We then traversed the layers of the JVM and its runtime environment, crucial for efficient transaction processing. Our analysis incorporated the nuances of containerized environments. Understanding the JVM's behavior within these containers and how it interacts with orchestration platforms like Kubernetes is pivotal in aligning JVM resource allocation with the OLTP system's performance goals.

As we moved deeper, the role of the OS came into focus, especially in terms of process scheduling, memory management, and I/O operations. Here, we explored diagnostic tools to gain insights into how the OS influences the Java application, particularly in a high-transaction OLTP environment. Finally, we reached the foundational hardware subsystem, where we assessed the CPU, memory, storage, and network components. By employing system profilers and PMUs, we can gain a comprehensive view of the hardware's performance and its interaction with our software layers.

Performance engineering is a critical aspect of software engineering that addresses the efficiency and effectiveness of a system. It requires a deep understanding of each layer of the modern stack—from application down to hardware—and employs various methodologies and tools to identify and resolve performance bottlenecks. By continuously monitoring, judiciously profiling, expertly analyzing, and meticulously tuning the system, performance engineers can enhance the system's performance and scalability, thereby improving the overall user experience—that is, how effectively and efficiently the end user can interact with the system. Among other factors, the overall user experience includes how quickly the system responds to user requests, how well it handles high-demand periods, and how reliable it is in delivering the expected output. An improved user experience often leads to higher user satisfaction, increased system usage, and better fulfillment of the system's overall purpose.

But, of course, the journey doesn't end here. To truly validate and quantify the results of our performance engineering efforts, we need a robust mechanism that offers empirical evidence of our system's capabilities. This is where performance benchmarking comes into play.

The Importance of Performance Benchmarking

Benchmarking is an indispensable component of performance engineering, a field that measures and evaluates software performance. The process helps us ensure that the software consistently meets user expectations and performs optimally under various conditions. Performance benchmarking extends beyond the standard software development life cycle, helping us gauge and comprehend how the software operates, particularly in terms of its "ilities"—such as scalability, reliability, and similar nonfunctional requirements. Its primary function is to provide data-driven assurance about the performance capabilities of a system (i.e., the system under test [SUT]). Aligning with the methodologies we discussed earlier, the empirical approach provides us with assurance and validates theoretical underpinnings of system design with comprehensive testing and analysis. Performance benchmarking offers a reliable measure of how effectively and efficiently the system can handle varied conditions, ranging from standard operational loads to peak demand scenarios.

To accomplish this, we conduct a series of experiments designed to reveal the performance characteristics of the SUT under different conditions. This process is similar to how a unit of work (UoW) operates in performance engineering. For benchmarking purposes, a UoW could be a single user request, a batch of requests, or any task the system is expected to perform.

Key Performance Metrics

When embarking on a benchmarking journey, it's essential to extract specific metrics to gain a comprehensive understanding of the software's performance landscape:

- **Response time:** This metric captures the duration of each specific usage pattern. It's a direct indicator of the system's responsiveness to user requests.
- **Throughput:** This metric represents the system's capacity, indicating the number of operations processed within a set time frame. A higher throughput often signifies efficient code execution.
- **Java heap utilization:** This metric takes a deep dive into memory usage patterns. Monitoring the Java heap can reveal potential memory bottlenecks and areas for optimization.
- **Thread behavior:** Threads are the backbone of concurrent execution in Java. Analyzing counts, interactions, and potential blocking scenarios reveals insights into concurrency issues, including, starvation, over-and-under-provisioning, and potential deadlocks.
- **Hardware and OS pressures:** Beyond the JVM, it's essential to monitor system-level metrics. Keep an eye on CPU usage, memory allocation, network I/O, and other vital components to gauge the overall health of the system.

Although these foundational metrics provide a solid starting point, specific benchmarks often require additional JVM or application-level statistics tailored to the application's unique characteristics. By adhering to the performance engineering process and effectively leveraging both the top-down and bottom-up methodologies, we can better contextualize these metrics. This structured approach not only aids in targeted optimizations within our UoW, but also contributes to the broader understanding and improvement of the SUT's overall performance landscape. This dual perspective ensures a comprehensive optimization strategy, addressing specific, targeted tasks while simultaneously providing insights into the overall functioning of the system.

The Performance Benchmark Regime: From Planning to Analysis

Benchmarking goes beyond mere speed or efficiency measurements. It's a comprehensive process that delves into system behavior, responsiveness, and adaptability under different workloads and conditions. The benchmarking process involves several interconnected activities.

The Performance Benchmarking Process

The benchmarking journey begins with the identification of a need or a question and culminates in a comprehensive analysis. It involves the following steps (illustrated in Figure 5.12):

- **Requirements gathering:** We start by identifying the performance needs or asks guiding the benchmarking process. This phase often involves consulting the product's design to derive a performance test plan tailored to the application's unique requirements.
- **Test planning and development:** The comprehensive plan is designed to track requirements, measure specific implementations, and ensure their validation. The plan distinctly outlines the benchmarking strategy in regard to how to benchmark the unit of test (UoT), focusing on its limits. It also characterizes the workload/benchmark while defining acceptance criteria for the results. It's not uncommon for several phases of this process to be in progress simultaneously, reflecting the dynamic and evolving nature of performance engineering. Following the planning phase, we move on to the test development phase, keeping the implementation details in mind.
- **Performance Validation:** In the validation phase, every benchmark undergoes rigorous assessment to ensure its accuracy and relevance. Robust processes are put in place to enhance the repeatability and stability of results across multiple runs. It's crucial to understand both the UoT and the test harness's performance characteristics. Also, continuous validation becomes a necessary investment to ensure the benchmark remains true to its UoW and reflective of the real-world scenarios.

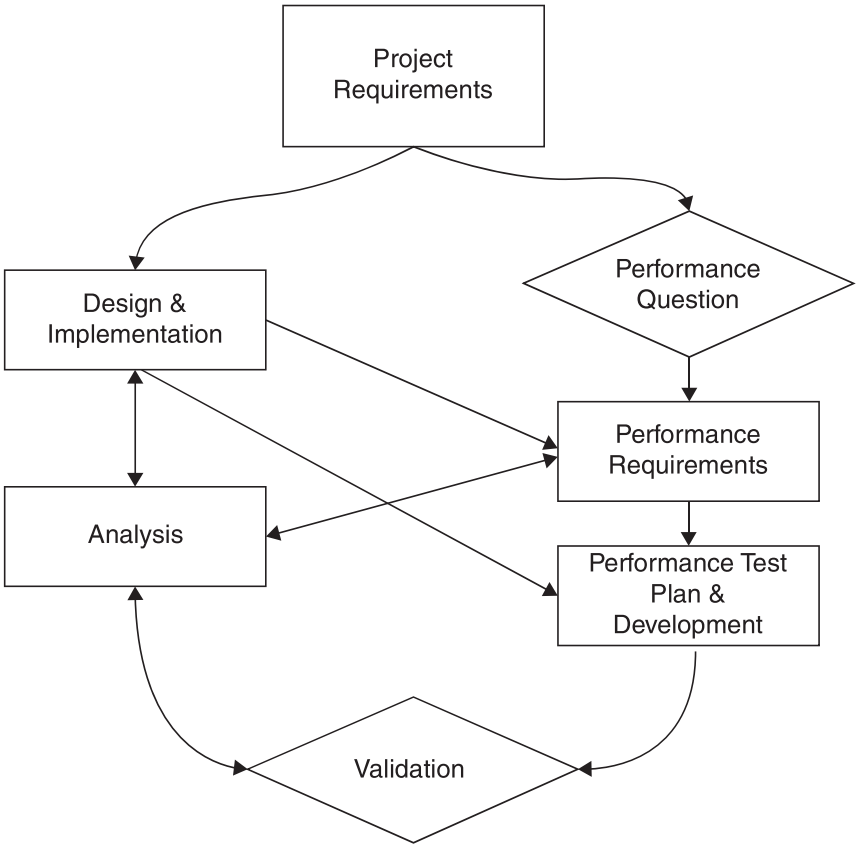

Figure 5.12 Benchmarking Process

- **Analysis:** In the analysis phase, we ensure that the test component (that is, the UoT) is stressed enough for performance engineers and developers to make informed decisions. This mirrors how performance engineers consider the interaction between the SUT and the UoW in performance evaluation. The accuracy of results is paramount, as the alternative is having irrelevant results drive our decision making. For instance, high variance in measurements indicates interference, necessitating an investigation into its source. Histograms of the measures, especially in high-variance scenarios, can be insightful. Performance testing inherently focuses on performance, presupposing that the UoT is functionally stable. Any performance test resulting in functional failures should be deemed a failed run, and its results discarded.

Iterative Approach to Benchmarking

Just as product design is continually revised and updated based on new insights and changing requirements, the performance plan is a living document. It evolves in response to the continuous insights gained through the performance analysis and design phases. This iterative process allows us to revisit, rethink, and redo any of the previous phases to take corrective action, ensuring that our benchmarks remain relevant and accurate. It enables us to capture system-level bottlenecks, measure resource consumption, understand execution details (for example, data structure locations on the heap), cache friendliness, and more. Iterative benchmarking also allows us to evaluate whether a new feature meets or exceeds the performance requirements without causing any regression.

When comparing two UoTs, it's essential to validate, characterize, and tune them independently, ensuring a fair comparison. Ensuring the accuracy and relevance of benchmarks is pivotal to making informed decisions in the realm of software performance.

Benchmarking JVM Memory Management: A Comprehensive Guide

In the realm of JVM performance, memory management stands out as a cornerstone. With the evolution of groundbreaking projects like Panama and Valhalla, and the advent of GCs such as G1, ZGC, and Shenandoah, the intricacies of JVM memory management have become a big concern. This section takes an in-depth look into benchmarking JVM memory management, a process that equips developers with the knowledge to optimize their applications effectively.

Setting Clear Objectives

- **Performance milestones:** Clearly define your goals. Whether it's minimizing latency, boosting throughput, or achieving optimal memory utilization, a well-defined objective will serve as the compass for your benchmarking endeavors.
- **Benchmarking environment:** Always benchmark within an environment that replicates your production setting. This ensures that the insights you gather are directly applicable to real-world scenarios.

Key Metrics to Benchmark

- **Allocation rates:** Monitor how swiftly your application allocates memory. Elevated rates can be indicative of inefficiencies in your application's memory usage patterns. With the OpenJDK GCs, high allocation rates and other stressors can combine forces and send the system into "graceful degradation" mode. In this mode, the GC may not meet its optimal pause time goals or throughput targets but will continue to function, ensuring that the application remains operational. Graceful degradation is a testament to the resilience and adaptability of these GCs, emphasizing the importance of monitoring and optimizing allocation rates for consistent application performance.
- **Garbage collection metrics:** It's essential to closely monitor GC efficiency metrics such as GC pause times, which reflect the durations for which application threads are halted to facilitate memory reclamation. Extended or frequent pauses can adversely affect application responsiveness. The frequency of GC events can indicate the JVM's memory pressure levels, with higher frequencies potentially pointing to memory challenges. Understanding the various GC phases, especially in advanced collectors, can offer granular insights into the GC process. Additionally, be vigilant of GC-induced throttling effects, which represent the performance impacts on the application due to GC activities, especially with the newer concurrent collectors.
- **Memory footprint:** Vigilantly track the application's total memory utilization within the JVM. This includes the heap memory, the segmented code cache, and the Metaspace. Each area has its role in memory management, and understanding their usage patterns can reveal optimization opportunities. Additionally, keep an eye on off-heap memory, which encompasses memory allocations outside the standard JVM heap and the JVM's own operational space, such as allocations by native libraries, thread stacks, and direct buffers. Monitoring this memory is vital, as unchecked off-heap usage can lead to unexpected out-of-memory or other suboptimal, resource-intensive access patterns, even if the heap seems to be comfortably underutilized.
- **Concurrency dynamics:** Delve into the interplay of concurrent threads and memory management, particularly with GCs like G1, ZGC, and Shenandoah. Concurrent data structures, which are vital for multithreaded memory management, introduce overheads in ensuring thread safety and data consistency. Maintenance barriers, which are crucial for memory ordering, can subtly affect both read and write access patterns. For instance, write barriers might slow data updates, whereas read barriers can introduce retrieval latencies. Grasping these barriers' intricacies is essential, as they can introduce complexities not immediately apparent in standard GC pause metrics.
- **Native memory interactions (for Project Panama):** Embracing the advancements brought by Project Panama represents an area of significant opportunity for JVM developers. By leveraging the Foreign Function and Memory (FFM) API, developers can tap into a new era of seamless and efficient Java-native interactions. Beyond just efficiency, Panama offers a safer interface, reducing the risks associated with traditional JNI methods. This transformative approach not only promises to mitigate commonly experienced issues such as buffer overflows, but also ensures type safety. As with any opportunity, it's crucial to benchmark and validate these interactions in real-world

scenarios, ensuring that you fully capitalize on the potential benefits and guarantees Panama brings to the table.

- **Generational activity and regionalized work:** Modern GCs like G1, Shenandoah, and ZGC employ a regionalized approach, partitioning the heap into multiple regions. This regionalization directly impacts the granularity of maintenance structures, such as remembered sets. Objects that exceed region size thresholds, termed "humongous," can span multiple regions and receive special handling to avert fragmentation. Additionally, the JVM organizes memory into distinct generations, each with its own collection dynamics. Maintenance barriers are employed across both generations and regions to ensure data integrity during concurrent operations. A thorough understanding of these intricacies is pivotal for advanced memory optimization, especially when analyzing region occupancies and generational activities.

Choosing the Right Tools

- **Java Micro-benchmarking Harness (JMH):** A potent tool for micro-benchmarks, JMH offers precise measurements for small code snippets.
- **Java Flight Recorder and JVisualVM:** These tools provide granular insights into JVM internals, including detailed GC metrics.
- **GC logs:** Analyzing GC logs can spotlight potential inefficiencies and bottlenecks. By adding visualizers, you can get a graphical representation of the GC process, making it easier to spot patterns, anomalies, or areas of concern.

Set Up Your Benchmarking Environment

- **Isolation:** Ensure your benchmarking environment is devoid of external interferences, preventing any distortions in results.
- **Consistency:** Uphold uniform configurations across all benchmarking sessions, ensuring results are comparable.
- **Real-world scenarios:** Emulate real-world workloads for the most accurate insights, incorporating realistic data and request frequencies.

Benchmark Execution Protocol

- **Warm-up ritual:** Allow the JVM to reach its optimal state, ensuring JIT compilation and class loading have stabilized before collecting metrics.
- **Iterative approach:** Execute benchmarks repeatedly, with results analyzed using geometric means and confidence intervals to ensure accuracy and neutralize outliers and anomalies.
- **Monitoring external factors:** Keep an eye on external influences such as OS-induced scheduling disruptions, I/O interruptions, network latency, and CPU contention. Be particularly attentive to the 'noisy neighbor' effect in shared environments, as these can significantly skew benchmark results.

Analyze Results

- **Identify bottlenecks:** Pinpoint areas where memory management might be hampering performance.
- **Compare with baselines:** If you've made performance-enhancing changes, contrast the new results with previous benchmarks to measure improvements.
- **Visual representation:** Utilize graphs for data trend visualization.

The Refinement Cycle

- **Refine code:** Modify your code or configurations based on benchmark insights.
- **Re-benchmark:** After adjustments, re-execute benchmarks to gauge the impact of those changes.
- **Continuous benchmarking:** Integrate benchmarking into your development routine for ongoing performance monitoring.

Benchmarking JVM memory management is an ongoing endeavor of measurement, analysis, and refinement. With the right strategies and tools in place, developers can ensure their JVM-based applications are primed for efficient memory management. Whether you're navigating the intricacies of projects like Panama and Valhalla or trying to understand modern GCs like G1, ZGC, and Shenandoah, a methodical benchmarking strategy is the cornerstone of peak JVM performance. Mastering JVM memory management benchmarking equips developers with the tools and insights needed to ensure efficient and effective application performance.

As we've delved into the intricacies of benchmarking and JVM memory management, you might have wondered about the tools and frameworks that can facilitate this process. How can you ensure that your benchmarks are accurate, consistent, and reflective of real-world scenarios? This leads us to the next pivotal topic.

Why Do We Need a Benchmarking Harness?

Throughout this chapter, we have emphasized the symbiotic relationship between hardware and software, including how their interplay profoundly influences their performance. When benchmarking our software to identify performance bottlenecks, we must acknowledge this dynamic synergy. Benchmarking should be viewed as an exploration of the entire system, factoring in the innate optimizations across different levels of the Java application stack.

Recall our previous discussions of the JVM, garbage collection, and concepts such as JIT compilation, tiered compilation thresholds, and code cache sizes. All of these elements play a part in defining the performance landscape of our software. For instance, as highlighted in the hardware–software interaction section, the processor cache hierarchies and processor memory model can significantly influence performance. Thus, a benchmark designed to gauge performance across different cloud architectures (for example) should be sensitive to these factors across the stack, ensuring that we are not merely measuring and operating in silos, but rather are taking full advantage of the holistic system and its built-in optimizations.

This is where a benchmarking harness comes into the picture. A benchmarking harness provides a standardized framework that not only helps us to circumvent common benchmarking pitfalls, but also streamlines the build, run, and timing processes for benchmarks, thereby ensuring that we obtain precise measurements and insightful performance evaluations. Recognizing this need, the Java community—most notably, Oracle engineers—has provided its own open-source benchmarking harness, encompassing a suite of micro-benchmarks, as referenced in JDK Enhancement Proposal (JEP) 230: *Microbenchmark Suite*.[14] Next, we will examine the nuances of this benchmarking harness and explore its importance in achieving reliable performance measurements.

The Role of the Java Micro-Benchmark Suite in Performance Optimization

Performance optimization plays a critical role in the release cycle, especially the introduction of features intended to boost performance. For instance, JEP 143: *Improve Contended Locking*[15] employed 22 benchmarks to gauge performance improvements related to object monitors. A core component of performance testing is regression analysis, which compares benchmark results between different releases to identify nontrivial performance regressions.

The Java Microbenchmark Harness (JMH) brings stability to the rigorous benchmarking process that accompanies each JDK release. JMH, which was first added to the JDK tree with JDK 12, provides scripts and benchmarks that compile and run for both current and previous releases, making regression studies easier.

Key Features of JMH for Performance Optimization

- **Benchmark modes:** JMH supports various benchmarking modes, such as Average Time, Sample Time, Single Shot Time, Throughput, and All. Each of these modes provides a different perspective on the performance characteristics of the benchmark.
- **Multithreaded benchmarks:** JMH supports both single-threaded and multithreaded benchmarks, allowing performance testing under various levels of concurrency.
- **Compiler control:** JMH offers mechanisms to ensure the JIT compiler doesn't interfere with the benchmarking process by over-optimizing or eliminating the benchmarking code.
- **Profilers:** JMH comes with a simple profiler interface plus several default profilers, including the GC profiler, Compiler profiler, and Classloader profiler. These profilers can provide more in-depth information about how the benchmark is running and where potential bottlenecks are.

JMH is run as a stand-alone project, with dependencies on user benchmark JAR files. For those readers who are unfamiliar with Maven templates, we'll begin with an overview of Maven and the information required to set up the benchmarking project.

[14] https://openjdk.org/jeps/230

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

Getting Started with Maven

Maven is a tool for facilitating the configuration, building, and management of Java-based projects. It standardizes the build system across all projects through a project object model (POM), bringing consistency and structure to the development process. For more information about Maven, refer to the Apache Maven Project website.[16]

After installing Maven, the next step is setting up the JMH project.

Setting Up JMH and Understanding Maven Archetype

The Maven Archetype is a project templating toolkit used to create a new project based on the JMH project Archetype. Its setup involves assigning a "GroupId" and an "ArtifactId" to your local project. The JMH setup guide provides recommendations for the Archetype and your local project. In our example, we will use the following code to generate the project in the `mybenchmarks` directory:

```
$ mvn archetype:generate \
   -DarchetypeGroupId=org.openjdk.jmh \
   -DarchetypeArtifactId=jmh-java-benchmark-archetype \
   -DarchetypeVersion=1.36 \
   -DgroupId=org.jmh.suite \
   -DartifactId=mybenchmarks \
   -DinteractiveMode=false \
   -Dversion=1.0
```

After running the commands, you should see a "BUILD SUCCESS" message.

Writing, Building, and Running Your First Micro-benchmark in JMH

The first step in creating a micro-benchmark is to write the benchmark class. This class should be located in the `mybenchmarks` directory tree. To indicate that a method is a benchmark method, annotate it with `@Benchmark`.

Let's write a toy benchmark to measure the performance of two key operations in an educational context: registering students and adding courses. The `OnlineLearningPlatform` class serves as the core of this operation, providing methods to register students and add courses. Importantly, the `Student` and `Course` entities are defined as separate classes, each encapsulating their respective data. The benchmark methods `registerStudentsBenchmark` and `addCoursesBenchmark` in the `OnlineLearningPlatformBenchmark` class aim to measure the performance of these operations. The simple setup offers insights into the scalability and performance of the `OnlineLearningPlatform`, making it an excellent starting point for beginners in JMH.

```
package org.jmh.suite;

import org.openjdk.jmh.annotations.Benchmark;
```

[16] https://maven.apache.org/

```
import org.openjdk.jmh.annotations.State;
import org.openjdk.jmh.annotations.Scope;

// JMH Benchmark class for OnlineLearningPlatform
@State(Scope.Benchmark)
public class OnlineLearningPlatformBenchmark {

    // Instance of OnlineLearningPlatform to be used in the benchmarks
    private final OnlineLearningPlatform platform = new OnlineLearningPlatform();

    // Benchmark for registering students
    @Benchmark
    public void registerStudentsBenchmark() {
        Student student1 = new Student("Annika Beckwith");
        Student student2 = new Student("Bodin Beckwith");
        platform.registerStudent(student1);
        platform.registerStudent(student2);
    }

    // Benchmark for adding courses
    @Benchmark
    public void addCoursesBenchmark() {
        Course course1 = new Course("Marine Biology");
        Course course2 = new Course("Astrophysics");
        platform.addCourse(course1);
        platform.addCourse(course2);
    }

}
```

The directory structure of the project would look like this:

```
mybenchmarks
└──src
    └──main
        └──java
            └──org
                └──jmh
                    └──suite
                        └──OnlineLearningPlatformBenchmark.java
                        └──OnlineLearningPlatform.java
                        └──Student.java
                        └──Course.java
target
└──benchmarks.jar
```

Once the benchmark class is written, it's time to build the benchmark. The process remains the same even when more benchmarks are added to the `mybenchmarks` directory. To build and install the project, use the Maven command `mvn clean install`. After successful execution, you'll see the "BUILD SUCCESS" message. This indicates that your project has been successfully built and the benchmark is ready to be run.

Finally, to run the benchmark, enter the following at the command prompt:

```
$ java –jar target/benchmarks.jar org.jmh.suite.OnlineLearningPlatformBenchmark
```

The execution will display the benchmark information and results, including details about the JMH and JDK versions, warm-up iterations and time, measurement iterations and time, and others.

The JMH project then presents the aggregate information after completion:

```
# Run complete. Total time: 00:16:45

Benchmark                                                  Mode   Cnt Score          Error        Units
OnlineLearningPlatformBenchmark.registerStudentsBenchmark thrpt  25  50913040.845 ± 1078858.366  ops/s
OnlineLearningPlatformBenchmark.addCoursesBenchmark       thrpt  25  50742536.478  ± 1039294.551 ops/s
```

That's it! We've completed a run and found our methods executing a certain number of operations per second (default = Throughput mode). Now you're ready to refine and add more complex calculations to your JMH project.

Benchmark Phases: Warm-Up and Measurement

To ensure developers get reliable benchmark results, JMH provides `@Warmup` and `@Measurement` annotations that help specify the warm-up and measurement phases of the benchmark. This is significant because the JVM's JIT compiler often optimizes code during the first few runs.

```
import org.openjdk.jmh.annotations.*;

@Warmup(iterations = 5)
@Measurement(iterations = 5)
public class MyBenchmark {

    @Benchmark
    public void testMethod() {
        // your benchmark code here...
    }
}
```

In this example, JMH first performs five warm-up iterations, running the benchmark method but not recording the results. This warm-up phase is analogous to the ramp-up phase in the application's life cycle, where the JVM is gradually improving the application's performance until it reaches peak performance. The warm-up phase in JMH is designed to reach this steady-state before the actual measurements are taken.

After the warm-up, JMH does five measurement iterations, where it does record the results. This corresponds to the steady-state phase of the application's life cycle, where the application executes its main workload at peak performance.

In combination, these features make JMH a powerful tool for performance optimization, enabling developers to better understand the behavior and performance characteristics of their Java code. For detailed learning, it's recommended to refer to the official JMH documentation and sample codes provided by the OpenJDK project.[17]

Loop Optimizations and @OperationsPerInvocation

When dealing with a loop in a benchmark method, you must be aware of potential loop optimizations. The JVM's JIT compiler can recognize when the result of a loop doesn't affect the program's outcome and may remove the loop entirely during optimization. To ensure that the loop isn't removed, you can use the `@OperationsPerInvocation` annotation. This annotation tells JMH how many operations your benchmark method is performing, preventing the JIT compiler from skipping the loop entirely.

The `Blackhole` class is used to "consume" values that you don't need but want to prevent from being optimized away. This outcome is achieved by passing the value to the `blackhole.consume()` method, which ensures the JIT compiler won't optimize away the code that produced the value. Here's an example:

```
@Benchmark
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@OperationsPerInvocation(10000)
public void measureLoop() {
    for (int i = 0; i < 10000; i++) {
        blackhole.consume(i);
    }
}
```

In this example, the `@OperationsPerInvocation(10000)` annotation informs JMH that the `measureLoop()` method performs 10,000 operations. The `blackhole.consume(i)` call ensures that the loop isn't removed during JIT optimizations, leading to accurate benchmarking of the loop's performance.

[17] https://github.com/openjdk/jmh

Benchmarking Modes in JMH

JMH provides several benchmarking modes, and the choice of mode depends on which aspect of performance you're interested in.

- **Throughput:** Measures the benchmark method's throughput, focusing on how many times the method can be called within a given time unit. This mode is useful when you want to measure the raw speed, such as operations in a data stream.
- **Average Time:** Measures the average time taken per benchmark method invocation, offering insights into how long it typically takes to execute a single method call. This mode is useful for understanding the performance of discrete operations, such as processing a request to a web server.
- **Sample Time:** Measures the time it takes for each benchmark method invocation and calculates the distribution of invocation times. This mode is useful when you want to understand the distribution of times, not just the average, such as latency of a network call.
- **Single Shot Time:** Measures the time it takes for a single invocation of the benchmark method. This mode is useful for benchmarks that take a significant amount of time to execute, such as a complex algorithm or a large database query.
- **All:** Runs the benchmark in all modes and reports the results for each mode. This mode is useful when you want a comprehensive view of your method's performance.

To specify the benchmark mode, use the `@BenchmarkMode` annotation on your benchmark method. For instance, to benchmark a method in throughput mode, you would use the following code:

```
@Benchmark
@BenchmarkMode(Mode.Throughput)
public void myMethod() {
    // ...
}
```

Understanding the Profilers in JMH

In addition to supporting the creation, building, and execution of benchmarks, JMH provides inbuilt profilers. Profilers can help you gain a better understanding of how your code behaves under different circumstances and configurations. In the case of JMH, these profilers provide insights into various aspects of the JVM, such as garbage collection, method compilation, class loading, and code generation.

You can specify a profiler using the `-prof` command-line option when running the benchmarks. For instance, `java -jar target/benchmarks.jar -prof gc` will run your benchmarks with the GC profiler enabled. This profiler reports the amount of time spent in GC

and the total amount of data processed by the GC during the benchmarking run. Other profilers can provide similar insights for different aspects of the JVM.

Key Annotations in JMH

JMH provides a variety of annotations to guide the execution of your benchmarks:

- **`@Benchmark`:** This annotation denotes a method as a benchmark target. The method with this annotation will be the one where the performance metrics are collected.
- **`@BenchmarkMode`:** This annotation specifies the benchmarking mode (for example, Throughput, Average Time). It determines how JMH runs the benchmark and calculates the results.
- **`@OutputTimeUnit`:** This annotation specifies the unit of time for the benchmark's output. It allows you to control the time granularity in the benchmark results.
- **`@Fork`:** This annotation specifies the number of JVM forks for each benchmark. It allows you to isolate each benchmark in its own process and avoid interactions between benchmarks.
- **`@Warmup`:** This annotation specifies the number of warm-up iterations, and optionally the amount of time that each warm-up iteration should last. Warm-up iterations are used to get the JVM up to full speed before measurements are taken.
- **`@Measurement`:** This annotation specifies the number of measurement iterations, and optionally the amount of time that each measurement iteration should last. Measurement iterations are the actual benchmark runs whose results are aggregated and reported.
- **`@State`:** This annotation is used to denote a class containing benchmark state. The state instance's life-cycle[18] is managed by JMH. It allows you to share common setup code, variables and define thread-local or shared scope to control state sharing between benchmark methods.
- **`@Setup`:** This annotation marks a method that should be executed to set up a benchmark or state object. It's a life-cycle method that is called before the benchmark method is executed.
- **`@TearDown`:** This annotation marks a method that should be executed to tear down a benchmark or state object. It's a life-cycle method that is called after the benchmark method is executed.
- **`@OperationsPerInvocation`:** This annotation indicates the number of operations a benchmark method performs in each invocation. It's used to inform JMH about the number of operations in a benchmark method, which is useful to avoid distortions from loop optimizations.

[18] 'Lifecycle' refers to the sequence of stages (setup, execution, teardown) a benchmark goes through.

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

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

JVM Benchmarking with JMH

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Profiling JMH Benchmarks with perfasm

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

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

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

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

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

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

Conclusion

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

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

This page intentionally left blank