### Custom Asynchronous Logging Solutions

In some scenarios, the standard frameworks may not meet the unique requirements of an application. In such cases, developing a custom asynchronous logging solution may be appropriate. Key considerations for a custom implementation include

- **Thread safety:** Ensuring that the logging operations are thread-safe to avoid data corruption or inconsistencies.
- **Memory management:** Efficiently handling memory within the logging process, particularly in managing the log message buffer, to prevent memory leaks or high memory consumption.
- **Example implementation:** A basic custom asynchronous logger might involve a producer–consumer pattern, where the main application threads (producers) enqueue log messages to a shared buffer, and a separate logging thread (consumer) dequeues and processes these messages.

### Integration with Unified JVM Logging

- **Challenges and solutions:** For JVM versions prior to JDK 17 or in complex logging scenarios, integrating external asynchronous logging frameworks with the unified JVM logging system requires careful configuration to ensure compatibility and performance. This might involve setting JVM flags and parameters to correctly route log messages to the asynchronous logging system.
- **Native support in JDK 17 and beyond:** Starting from JDK 17, the unified logging system natively supports asynchronous logging. This can be enabled using the `-Xlog:async` option, allowing the JVM's internal logging to be handled asynchronously without affecting the performance of the JVM itself.

## Best Practices and Considerations

### Managing Log Backpressure

- **Challenge:** In asynchronous logging, backpressure occurs when the rate of log message generation exceeds the capacity for processing and writing these messages. This can lead to performance degradation or loss of log data.
- **Bounded queues:** Implementing bounded queues in the logging system can help manage the accumulation of log messages. By limiting the number of unprocessed messages, these queues prevent memory overutilization.
- **Discarding policies:** Establishing policies for discarding log messages when queues reach their capacity can prevent system overload. This might involve dropping less critical log messages or summarizing message content.
- **Real-life examples:** A high-traffic web application might implement a discarding policy that prioritizes error logs over informational logs during peak load times to maintain system stability.

### Ensuring Log Reliability

- **Criticality:** The reliability of a logging system is paramount, particularly during application crashes or abrupt shutdowns. Loss of log data in such scenarios can hinder troubleshooting and impact compliance.
- **Write-ahead logs:** Implementing write-ahead logging ensures that log data is preserved even if the application fails. This technique involves writing log data to a persistent storage site before the actual logging operation concludes.
- **Graceful shutdown procedures**: Designing logging systems to handle shutdowns gracefully ensures that all buffered log data is written out before the application stops completely.
- **Strategies in adverse conditions:** Include redundancy in the logging infrastructure to capture log data even if the primary logging mechanism fails. This is particularly relevant in distributed systems where multiple components can generate logs.

### Performance Tuning of Asynchronous Logging Systems

- **Tuning for optimal performance**
  - **Thread pool sizes:** Adjusting the number of threads dedicated to asynchronous logging is crucial. More threads can handle higher log volumes, but excessive threads may cause heavy CPU overhead.
  - **Buffer capacities:** If your application experiences periodic spikes in log generation, increasing the buffer size can help absorb these spikes.
  - **Processing intervals:** The frequency at which the log is flushed from the buffer to the output can impact both performance and log timeliness. Adjusting this interval helps balance CPU usage against the immediacy of log writing.
- **JDK 17 considerations with asynchronous logging**
  - **Buffer size tuning:** The `AsyncLogBufferSize` parameter is crucial for managing the volume of log messages that the system can handle before flushing. A larger buffer can accommodate more log messages, which is beneficial during spikes in log generation. However, a larger buffer also requires more memory.
  - **Monitoring JDK 17's asynchronous logging:** With the new asynchronous logging feature, monitoring tools and techniques might need to be updated to track the utilization of the new buffer and the performance of the logging system.
- **Monitoring and metrics**
  - **Buffer utilization:** You could use a logging framework's internal metrics or a custom monitoring script to track buffer usage. If the buffer consistently has greater than 80% utilization, that's a sign you should increase its size.
  - **Queue lengths:** Monitoring tools or logging framework APIs could be used to track the length of the logging queue. Persistently long queues require more logging threads or a larger buffer.

- **Write latencies:** Measuring the time difference between log message creation and its final write operation could reveal write latencies. Optimizing file I/O operations or distributing logs across multiple files or disks could help if these latencies are consistently high.
- **JVM tools and metrics:** Tools like JVisualVM[6] and Java Mission Control (JMC)[7] can provide insights into JVM performance, including aspects that might impact logging, such as thread activity or memory usage.

## Understanding the Enhancements in JDK 11 and JDK 17

The unified logging system was first introduced in JDK 9 and has since undergone refinements and enhancements in subsequent versions of the JDK. This section discusses the significant improvements made in JDK 11 and JDK 17 with respect to unified logging.

### JDK 11

In JDK 11, one of the notable improvements to the unified logging framework was addition of the ability to dynamically configure logging at runtime. This enhancement was facilitated by the `jcmd` utility, which allows developers to send commands to a running JVM, including commands to adjust the log level. This innovation has significantly benefited developers by enabling them to increase or decrease logging, depending on the diagnostic requirements, without restarting the JVM.

JDK 11 also improved `java.util.logging` by introducing new methods for Logger and LogManager. These additions further enhanced logging control and capabilities, providing developers with more granularity and flexibility when working with Java logs.

### JDK 17

JDK 17 introduced native support for asynchronous logging improvements within the unified logging system of the JVM. In addition, JDK 17 brought broader, indirect improvements to the logging system's overall performance, security, and stability. Such improvements contribute to the effectiveness of unified logging, as a more stable and secure system is always beneficial for accurate and reliable logging.

## Conclusion

In this chapter, we explored the unified logging system, showcasing its practical usage and outlining the enhancements introduced in subsequent releases of the JDK. The unified logging system in JVM provides a single, consistent interface for all log messages. This harmony across the JVM has enhanced both readability and parseability, effectively eliminating issues with interleaved or interspersed logs. Preserving the ability to redirect logs to a file—a feature available even prior to JDK 9—was an essential part of retaining the usefulness of the earlier logging mechanisms. The logs continue to be human-readable, and timestamps, by default, reflect the uptime.

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

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

The unified logging system also offers significant value-added features. The introduction of dynamic logging in JDK 11 has allowed developers to make adjustments to logging commands during runtime. Combined with the enhanced visibility and control over the application process provided by the unified logging system, developers can now specify the depth of log information required via command-line inputs, improving the efficiency of debugging and performance testing.

The information obtained from the unified logging system can be used to tune JVM parameters for optimal performance. For example, I have my own GC parsing and plotting toolkit that is now more consolidated and streamlined to provide insights into the memory usage patterns, pause events, heap utilization, pause details, and various other patterns associated with the collector type. This information can be used to adjust the heap size, choose the appropriate garbage collector, and tune other garbage collection parameters.

Newer features of Java, such as Project Loom and Z Garbage Collector (ZGC), have significant implications for JVM performance. These features can be monitored via the unified logging system to understand their impacts on application performance. For instance, ZGC logs can provide insights into the GC's pause times and efficiency in reclaiming memory. We will learn more about these features in upcoming chapters.

In conclusion, the unified logging system in JVM is a potent tool for developers. By understanding and leveraging this system, developers can streamline their troubleshooting efforts, gaining deeper insights into the behavior and performance of their Java applications. Continuous enhancements across different JDK versions have made the unified logging system even more robust and versatile, solidifying its role as an essential tool in the Java developers' toolkit.

# Chapter 5

# End-to-End Java Performance Optimization: Engineering Techniques and Micro-benchmarking with JMH

## Introduction

Welcome to a pivotal stage in our journey through Java Virtual Machine (JVM) performance engineering. This chapter marks a significant transition, where we move from understanding the historical context and foundational structures of Java to applying this knowledge to optimize performance. This sets the stage for our upcoming in-depth exploration of performance optimization in the OpenJDK HotSpot JVM.

Together, we've traced the evolution of Java, its type system, and the JVM, witnessing Java's transition from a monolithic structure to a modular one. Now, we're poised to apply these concepts to the vital task of performance optimization.

Performance is a cornerstone of any software application's success. It directly influences the user experience, determining how swiftly and seamlessly an application responds to user interactions. The JVM, the runtime environment where Java applications are executed, is central to this performance. By learning how to adjust and optimize the JVM, you can significantly boost your Java application's speed, responsiveness, and overall performance. This could involve strategies such as adjusting memory and garbage collection patterns, using optimized code or libraries, or leveraging JVM's just-in-time (JIT) compilation capabilities and thresholds to your advantage. Mastering these JVM optimization techniques is a valuable skill for enhancing your Java applications' performance.

Building on our previous foundational investigation into the unified JVM logging interface, we now can explore performance metrics and consider how to measure and interpret them effectively. As we navigate the intricacies of JVM, we'll examine its interaction with hardware, the

role of memory barriers, and the use of volatile variables. We'll also delve into the comprehensive process of performance optimization, which encompasses monitoring, profiling, analysis, and tuning.

This chapter transcends mere theoretical discourse, embodying a distillation of over two decades of hands-on experience in performance optimization. Here, indispensable tools for Java performance measurement are introduced, alongside diagnostic and analysis methodologies that have been meticulously cultivated and refined throughout the years. This approach, involving a detailed investigation of subsystems to pinpoint potential issues, offers a crucial roadmap for software developers with a keen focus on performance optimization.

My journey has led to a deep understanding of the profound impact that underlying hardware and software stack layers have on an application's performance, hence this chapter sheds light on the interplay between hardware and software. Another dedicated section recognizes the critical importance of benchmarking, whether for assessing the impact of feature releases or analyzing performance across different system layers. Together, we will explore practical applications using the Java Micro-benchmark Harness (JMH). Additionally, we will look at the built-in profilers within JMH, such as *perfasm*, a powerful tool that can unveil the low-level performance characteristics of your code.

This chapter, therefore, is more than just a collection of theories and concepts. It's a practical guide, filled with examples and best practices, all drawn from extensive field experience. Designed to bridge the gap between theory and practice, it serves as a springboard for the detailed performance optimization discussions in the subsequent chapters of this book, ensuring readers are well-equipped to embark on the advanced performance engineering journey.

# Performance Engineering: A Central Pillar of Software Engineering

## Decoding the Layers of Software Engineering

Software engineering is a comprehensive discipline that encapsulates two distinct yet intertwined dimensions: (1) software design and development and (2) software architectural requirements.

Software design and development is the first dimension that we encounter on this journey. It involves crafting a detailed blueprint or schema for a system, which addresses the system's architecture, components, interfaces, and other distinguishing features. These design considerations build the foundation for the functional requirements—the rules that define what the system should accomplish. The functional requirements encapsulate the business rules with which the system must comply, the data manipulations it should perform, and the interactions it should facilitate.

In contrast, software architectural requirements focus on the nonfunctional or quality attributes of the system—often referred to as the "ilities." These requirements outline how the system should behave, encompassing performance, usability, security, and other crucial attributes.

Understanding and striking a balance between these two dimensions—catering to the functional needs while ensuring high-quality attributes—is a critical aspect of software

engineering. With this foundational understanding in place, let's focus on one of these essential qualities—performance.

## Performance: A Key Quality Attribute

Among the various architectural requirements, performance holds a distinctive position. It measures how effectively a system—referred to as the "system under test" (SUT)— executes a specific task or a "unit of work" (UoW). Performance is not merely a nice-to-have feature; it's a fundamental quality attribute. It influences both the efficiency of operations and the user experience, and it plays a pivotal role in determining user satisfaction.

To fully comprehend the concept of performance, we must understand the various system components and their impacts on performance. In the context of a managed runtime system like the JVM, a key component that significantly affects performance is the garbage collector (GC). I define the unit of work for GC in performance parlance as "GC work," referring to the specific tasks each GC must perform to meet its performance requirements. For example, a type of GC that I categorize as "throughput-maximizing," exemplified by the Parallel GC in OpenJDK HotSpot VM, focuses on "parallel work." In contrast, a "low-latency-optimizing" collector, as I define it, requires greater control over its tasks to guarantee sub-millisecond pauses. Hence, understanding the individual attributes and contributions of these components is critical, as together they create a holistic picture of system performance.

## Understanding and Evaluating Performance

Performance evaluation isn't a mere measure of how fast or slow a system operates. Instead, it involves understanding the interactions between the SUT and the UoW. The UoW could be a single user request, a batch of requests, or any task the system needs to perform. Performance depends on how effectively and efficiently the SUT can execute the UoW under various conditions. It encompasses factors such as the system's resource utilization, throughput, and response time, as well as how these parameters change as the workload increases.

Evaluating performance also requires engaging with the system's users, including understanding what makes them happy and what causes them dissatisfaction. Their insights can reveal hidden performance issues or suggest potential improvements. Performance isn't just about speed—it's about ensuring the system consistently meets user expectations.

## Defining Quality of Service

Quality of service (QoS) is a broad term that represents the user's perspective on the service's performance and availability. In today's dynamic software landscape, aligning QoS with service level agreements (SLAs) and incorporating modern site reliability engineering (SRE) concepts like service level objectives (SLOs) and service level indicators (SLIs) is crucial. These elements work together to ensure a shared understanding of what constitutes satisfactory performance.

SLAs define the expected level of service, typically including metrics for system performance and availability. They serve as formal agreements between service providers and users

regarding the expected performance standards, providing a benchmark against which the system's performance can be measured and evaluated.

SLOs are specific, measurable goals within SLAs that reflect the desired level of service performance. They act as targets for reliability and availability, guiding teams in maintaining optimal service levels. In contrast, SLIs are the quantifiable measures used to evaluate the performance of the service against the SLOs. They provide real-time data on various aspects of the service, such as latency, error rates, or uptime, allowing for continuous monitoring and improvement.

Defining clear, quantifiable QoS metrics, in light of SLAs, SLOs, and SLIs, ensures that all parties involved have a comprehensive and updated understanding of service performance expectations and metrics used to gauge them. This approach is increasingly considered best practice in managing application reliability and is integral to modern software development and maintenance.

In refining our understanding of QoS, I have incorporated insights from industry expert Stefano Doni, who is the co-founder and chief technology officer at Akamas.[1] Stefano's suggestion to incorporate SRE concepts like SLOs and SLIs adds significant depth and relevance to our discussion.

### Success Criteria for Performance Requirements

Successful performance isn't a "one size fits all" concept. Instead, it should be defined in terms of measurable, realistic criteria that reflect the unique requirements and constraints of each system. These criteria may include metrics such as response time, system throughput, resource utilization, and more. Regular monitoring and measurement against these criteria can help ensure the system continues to meet its performance objectives.

In the next sections, we'll explore the specific metrics for measuring Java performance, highlighting the significant role of hardware in shaping its efficiency. This will lay the groundwork for an in-depth discussion on concurrency, parallelism in hardware, and the characteristics of weakly ordered systems, leading seamlessly into the world of benchmarking.

## Metrics for Measuring Java Performance

Performance metrics are critical to understanding the behavior of your application under different conditions. These metrics provide a quantitative basis for measuring the success of your performance optimization efforts. In the context of Java performance optimization, a wide array of metrics are employed to gauge different aspects of applications' behavior and efficiency. These metrics encompass everything from latency, which is particularly insightful when viewed in the context of stimulus-response, to scalability, which assesses the application's ability to handle increasing loads. Efficiency, error rates, and resource consumption are other critical metrics that offer insights into the overall health and performance of Java applications.

[1] Akamas focuses on AI-powered autonomous performance optimization solutions; www.akamas.io/about.

Each metric provides a unique lens through which the performance of a Java application can be examined and optimized. For instance, latency metrics are crucial in real-time processing environments where timely responses are desired. Meanwhile, scalability metrics are vital not just in assessing an application's current capacity to grow and manage peak loads, but also for determining its scaling factor—a measure of how effectively the application can expand its capacity in response to increased demands. This scaling factor is integral to capacity planning because it helps predict the necessary resources and adjustments needed to sustain and support the application's growth. By understanding both the present performance and potential scaling scenarios, you can ensure that the application remains robust and efficient, adapting seamlessly to both current and future operational challenges.

A particularly insightful aspect of evaluating responsiveness and latency is understanding the call chain depth in an application. Every stage in a call chain can introduce its own delay, and these delays can accumulate and amplify as they propagate down the chain. This effect means that an issue at an early stage can significantly exacerbate the overall latency, affecting the application's end-to-end responsiveness. Such dynamics are crucial to understand, especially in complex applications with multiple interacting components.

In the Java ecosystem, these metrics are not only used to monitor and optimize applications in production environments, but they also play a pivotal role during the development and testing phases. Tools and frameworks within the Java world, such as JMH, offer specialized functionalities to measure and analyze these performance indicators.

For the purpose of this book, while acknowledging the diversity and importance of various performance metrics in Java, we will concentrate on four key areas: footprint, responsiveness, throughput, and availability. These metrics have been chosen due to their broad applicability and impact on Java applications:

## Footprint

Footprint pertains to the space and resources required to run your software. It's a crucial factor, especially in environments with limited resources. The footprint comprises several aspects:

- **Memory footprint:** The amount of memory your software consumes during execution. This includes both the Java heap for object storage and native memory used by the JVM itself. The memory footprint encompasses runtime data structures, which constitute the application system footprint. Native memory tracking (NMT) in the JVM can play a key role here, providing detailed insights into the allocation and usage of native memory. This is essential for diagnosing potential memory inefficiencies or leaks outside of the Java heap. In Java applications, optimizing memory allocation and garbage collection strategies is key to managing this footprint effectively.
- **Code footprint:** The size of the actual executable code of your software. Large codebases can require more memory and storage resources, impacting the system's overall footprint. In managed runtime environments, such as those provided by the JVM, the code footprint can have direct impact on the process, influencing overall performance and resource utilization.

- **Physical resources:** The use of resources such as storage, network bandwidth, and, especially, CPU usage. CPU footprint, or the amount of CPU resources consumed by your application, is a critical factor in performance and cost-effectiveness, especially in cloud environments. In the context of Java applications, the choice of GC algorithms can significantly affect CPU usage. Stefano notes that appropriate optimization of the GC algorithm can sometimes result in halving CPU usage, leading to considerable savings in cloud-based deployments.

These aspects of the footprint can significantly influence your system's efficiency and operational costs. For instance, a larger footprint can lead to longer start-up times, which is critical in contexts like container-based and serverless architectures. In such environments, each container includes the application and its dependencies, dictating a specific footprint. If this footprint is large, it can lead to inefficiencies, particularly in scenarios when multiple containers are running on the same host. Such situations can sometimes give rise to "noisy neighbor" issues, where one container's resource-intensive operations monopolize system resources, adversely affecting the performance of others. Hence, optimizing the footprint is not just about improving individual container efficiency; it is also about managing resource contention and ensuring more efficient resource usage on shared platforms.

In serverless computing, although resource management is automatic, an application with a large footprint may experience longer start-up times, known as "cold starts." This can impact the responsiveness of serverless functions, so footprint optimization is crucial for better performance. Similarly, for most Java applications, the memory footprint often directly correlates with the workload of the GC, which can impact performance. This is particularly true in cloud environments and with microservices architecture, where resource consumption directly affects responsiveness and operational overhead and requires careful capacity planning.

### Monitoring Non-Heap Memory with Native Memory Tracking

Building on our earlier discussion on NMT in the JVM, let's take a look at how NMT can be effectively employed to monitor non-heap memory. NMT provides valuable options for this purpose: a summary view—offering an overview of aggregate usage, and a detailed view—presenting a breakdown of usage by individual call sites:

```
-XX:NativeMemoryTracking=[off | summary | detail]
```

**NOTE** Enabling NMT, especially at the `detail` level, may introduce some performance overhead. Therefore, it should be used judiciously, particularly in production environments.

For real-time insights, `jcmd` can be used to grab the runtime `summary` or `detail`. For instance, to retrieve a `summary`, the following command is used:

```
$ jcmd <pid> VM.native_memory summary
<pid>:
Native Memory Tracking:
(Omitting categories weighting less than 1KB)
Total: reserved=12825060KB, committed=11494392KB
-                 Java Heap (reserved=10485760KB, committed=10485760KB)
                            (mmap: reserved=10485760KB, committed=10485760KB)
-                     Class (reserved=1049289KB, committed=3081KB)
                            (classes #4721)
                            (  instance classes #4406, array classes #315)
                            (malloc=713KB #12928)
                            (mmap: reserved=1048576KB, committed=2368KB)
                            (  Metadata:   )
                            (    reserved=65536KB, committed=15744KB)
                            (    used=15522KB)
                            (    waste=222KB =1.41%)
                            (  Class space:)
                            (    reserved=1048576KB, committed=2368KB)
                            (    used=2130KB)
                            (    waste=238KB =10.06%)
-                    Thread (reserved=468192KB, committed=468192KB)
                            (thread #456)
                            (stack: reserved=466944KB, committed=466944KB)
                            (malloc=716KB #2746)
                            (arena=532KB #910)
-                      Code (reserved=248759KB, committed=18299KB)
                            (malloc=1075KB #5477)
                            (mmap: reserved=247684KB, committed=17224KB)
-                        GC (reserved=464274KB, committed=464274KB)
                            (malloc=41894KB #10407)
                            (mmap: reserved=422380KB, committed=422380KB)
-                  Compiler (reserved=543KB, committed=543KB)
                            (malloc=379KB #424)
                            (arena=165KB #5)
-                  Internal (reserved=1172KB, committed=1172KB)
                            (malloc=1140KB #12338)
                            (mmap: reserved=32KB, committed=32KB)
-                     Other (reserved=818KB, committed=818KB)
                            (malloc=818KB #103)
-                    Symbol (reserved=3607KB, committed=3607KB)
                            (malloc=2959KB #79293)
                            (arena=648KB #1)
```

```
-    Native Memory Tracking (reserved=2720KB, committed=2720KB)
                            (malloc=352KB #5012)
                            (tracking overhead=2368KB)
-        Shared class space (reserved=16384KB, committed=12176KB)
                            (mmap: reserved=16384KB, committed=12176KB)
-               Arena Chunk (reserved=16382KB, committed=16382KB)
                            (malloc=16382KB)
-                   Logging (reserved=7KB, committed=7KB)
                            (malloc=7KB #287)
-                 Arguments (reserved=2KB, committed=2KB)
                            (malloc=2KB #53)
-                    Module (reserved=252KB, committed=252KB)
                            (malloc=252KB #1226)
-                 Safepoint (reserved=8KB, committed=8KB)
                            (mmap: reserved=8KB, committed=8KB)
-           Synchronization (reserved=1195KB, committed=1195KB)
                            (malloc=1195KB #19541)
-            Serviceability (reserved=1KB, committed=1KB)
                            (malloc=1KB #14)
-                 Metaspace (reserved=65693KB, committed=15901KB)
                            (malloc=157KB #238)
                            (mmap: reserved=65536KB, committed=15744KB)
-      String Deduplication (reserved=1KB, committed=1KB)
                            (malloc=1KB #8)
```

As you can see, this command generates a comprehensive report, categorizing memory usage across various JVM components such as the *Java Heap*, *Class*, *Thread*, *GC*, and *Compiler* areas, among others. The output also includes memory reserved and committed, aiding in understanding how memory is allocated and utilized by different JVM subsystems. You can also get diffs for either the `detail` level or `summary` view.

### Mitigating Footprint Issues

To address footprint-related challenges, insights gained from NMT, GC logs, JIT compilation data, and other JVM unified logging outputs can inform various mitigation strategies. Key approaches include

- **Data structure optimization:** Prioritize memory-efficient data structures by analyzing usage and selecting lighter, space-efficient alternatives tailored to your application's predictable data patterns. This approach ensures optimal memory utilization.
- **Code refactoring:** Streamline code to reduce redundant memory operations and refine algorithmic efficiency. Minimizing object creation, preferring primitives over boxed types, and optimizing string handling are key strategies to reduce memory usage.

- **Adaptive sizing and GC strategy:** Leverage modern GCs' adaptive sizing features to dynamically adjust to your application's behavior and workload. Carefully selecting the right GC algorithm—based on your footprint needs—enhances runtime efficiency. Where supported, incorporate `-XX:SoftMaxHeapSize` for applications with specific performance targets, allowing controlled heap expansion and optimizing GC pause times. This strategy, coupled with judiciously setting initial (`-Xms`) and maximum (`-Xmx`) heap sizes, forms a comprehensive approach to managing memory efficiently.

These techniques when applied cohesively, ensure a balanced approach to resource management, enhancing performance across various environments.

## Responsiveness

Latency and responsiveness, while related, illuminate different aspects of system performance. Latency measures the time elapsed from receiving a stimulus (like a user request or system event) to when a response is generated and delivered. It's a critical metric, but it captures only one dimension of performance.

Responsiveness, on the other hand, extends beyond mere speed of reaction. It evaluates how a system adapts under varied load conditions and sustains performance over time, offering a holistic view of the system's agility and operational efficiency. It encompasses the system's ability to promptly react to inputs and execute the necessary operations to produce a response.

In assessing system responsiveness, the previously discussed concepts of SLOs and SLIs become particularly relevant. By setting clear SLOs for response times, you establish the performance benchmarks your system aims to achieve. Consider the following essential questions to ask when assessing responsiveness:

- What is the targeted response time?
- Under what conditions is it acceptable for response times to exceed the targeted benchmarks, and to what extent is this deviation permissible?
- How long can the system endure delays?
- Where is response time measured: client side, server side, or end-to-end?

Establishing SLIs in response to these questions sets clear, quantifiable benchmarks to measure actual system performance relative to your SLOs. Tracking these indicators help identify any deviations and guide your optimization efforts. Tools such as JMeter, LoadRunner, and custom scripts can be used to measure responsiveness under different load conditions and perform continuous monitoring of the SLIs, providing insights into how well your system maintains the performance standards set by your SLOs.

## Throughput

Throughput is another critical metric that quantifies how many operations your system can handle per unit of time. It is vital in environments that require processing of high volumes of transactions or data. For instance, my role in performance consulting for a significant e-commerce platform highlighted the importance of maximizing throughput during peak periods like holiday sales. These periods could lead to significant spikes in traffic, so the system

needed to be able to scale resources to maintain high throughput levels and "keep the 4-9s in tail latencies" (that is, complete 99.99% of all requests within a specified time frame). By doing so, the platform could provide an enhanced user experience and ensure successful business operations.

Understanding your system's capacity to scale is crucial in these situations. Scaling strategies can involve horizontal scaling (that is, scaling out), which involves adding more systems to distribute the load, and vertical scaling (that is, scaling up), where you augment the capabilities of existing systems. For instance, in the context of the e-commerce platform during the holiday sales event, employing an effective scaling strategy is essential. Cloud environments often provide the flexibility to employ such strategies effectively, allowing for dynamic resource allocation that adapts to varying demand.

## Availability

Availability is a measure of your system's uptime—essentially, how often it is functional and accessible. Measures such as mean time between failure (MTBF) are commonly used to gauge availability. High availability is often essential for maintaining user satisfaction and meeting SLAs. Other important measures include mean time to recovery (MTTR), which indicates the average time to recover from a failure, and redundancy measures, such as the number of failover instances in a cluster. These can provide additional insights into the resilience of your system.

In my role as a performance consultant for a large-scale, distributed database company, high availability was a major focus area. The company specializes in developing scale-out architectures for data warehousing and machine learning, utilizing Hadoop for handling vast data sets. Achieving high availability requires robust fallback systems and strategies for scalability and redundancy. For example, in our Hadoop-based systems, we ensured high availability by maintaining multiple failover instances within a cluster. This setup allowed seamless transition to a backup instance in case of failure, minimizing downtime. Such redundancy is critical not only for continuous service but also for sustaining performance metrics like high throughput during peak loads, ensuring the system manages large volumes of requests or data without performance degradation.

Moreover, high availability is essential for maintaining consistent responsiveness, enabling swift recovery from failures and keeping response times stable, even under adverse conditions. In database systems, enhancing availability means increasing failover instances across clusters and diversifying them for greater redundancy. This strategy, rigorously tested, ensures smooth, automatic failover during outages, maintaining uninterrupted service and user trust.

Particularly in large-scale systems like those based on the Hadoop ecosystem—which emphasizes components like HDFS for data storage, YARN for resource management, or HBase for NoSQL database capabilities—understanding and planning for fallback systems is critical. Downtime in such systems can significantly impact business operations. These systems are designed for resilience, seamlessly taking over operations to maintain continuous service, integral to user satisfaction.

Through my experiences, I have learned that achieving high availability is a complex balancing act that requires careful planning, robust system design, and ongoing monitoring to ensure system resilience and maintain the trust of your users. Cloud deployments can significantly aid in achieving this balance. With their built-in automated failover and redundancy procedures, cloud services fortify systems against unexpected downtimes. This adaptability, along with the flexibility to scale resources on demand, is crucial during peak load periods, ensuring high availability under various conditions.

In conclusion, achieving high availability transcends technical execution; it's a commitment to building resilient systems that reliably meet and exceed user expectations. This dedication, pivotal in our digital era, demands not just advanced solutions like cloud deployments but also a holistic approach to system architecture. It involves integrating strategic design principles that prioritize seamless service continuity, reflecting a comprehensive commitment to reliability and performance excellence.

## Digging Deeper into Response Time and Availability

SLAs are essential in setting benchmarks for response time. They might include metrics such as average response time, maximum (worst-case) response time, 99th percentile response time, and other high percentiles (commonly referred to as the "4-9s" and "5-9s").

A visual representation of response times across different systems can reveal significant insights. In particular, capturing the worst-case scenario can identify potential availability issues that might go unnoticed when looking at average or best-case scenarios. As an example, consider Table 5.1, which presents recorded response times across four different systems.

Table 5.1 **Example: Monitoring Response Times**

| | Number of GC Events | Minimum (ms) | Average (ms) | 99th Percentile (ms) | Maximum (ms) |
|---|---|---|---|---|---|
| System 1 | 36,562 | 7.622 | 310.415 | 945.901 | 2932.333 |
| System 2 | 34,920 | 7.258 | 320.778 | 1006.677 | 2744.587 |
| System 3 | 36,270 | 6.432 | 321.483 | 1004.183 | 1681.305 |
| System 4 | 40,636 | 7.353 | 325.670 | 1041.272 | 18,598.557 |

The initial examination of the minimum, average, and 99th percentile response times in Table 5.1 might not raise any immediate concerns. However, the maximum response time for System 4, revealing a pause as long as ~18.6 seconds, warrants a deeper investigation. Initially, such extended pause times might suggest GC inefficiencies, but they could also stem from broader system latency issues, potentially triggering failover mechanisms in highly available distributed systems. Understanding this scenario is critical, as frequent occurrences can severely impact the MTBF and MTTR, pushing these metrics beyond acceptable thresholds.

This example underlines the importance of capturing and evaluating worst-case scenarios when assessing response times. Tools that can help profile and analyze such performance issues include system and application profilers like Intel's VTune, Oracle's Performance Analyzer, Linux *perf*, VisualVM,[2] and Java Mission Control,[3] among others. Utilizing these tools is instrumental in diagnosing systemic performance bottlenecks that compromise responsiveness, enabling targeted optimizations to enhance system availability.

## The Mechanics of Response Time with an Application Timeline

To better understand response time, consider Figure 5.1, which depicts an application timeline. The timeline illustrates two key events: the arrival of a stimulus $S_0$ at time $T_0$, and the application sending back a response $R_0$ at time $T_1$. Upon receiving the stimulus, the application continues to work ($A_{cw}$) to process this new stimulus, and then sends back a response $R_0$ at time $T_1$. The work happening before the stimulus arrival is denoted as $A_w$. Thus, Figure 5.1 illustrates the concept of response time in the context of application work.

### Understanding Response Time and Throughput in the Context of Pauses

Stop-the-world (STW) pauses refer to periods when all application threads are suspended. These pauses, which are required for garbage collection, can interrupt application work and impact response time and throughput. Considering the application timeline in the context of STW pauses can help us better appreciate how response time and throughput are affected by such events.

Let's consider a scenario where STW events interrupt the application's continuous work ($A_{cw}$). In Figure 5.2, two portions of the uninterrupted $A_{cw}$—$A_{cw_0}$ and $A_{cw_1}$—are separated by STW events. In this altered timeline, the response $R_0$ that was supposed to be sent at time $T_1$ in the uninterrupted scenario is now delayed until time $T_2$. This results in an elongated response time.

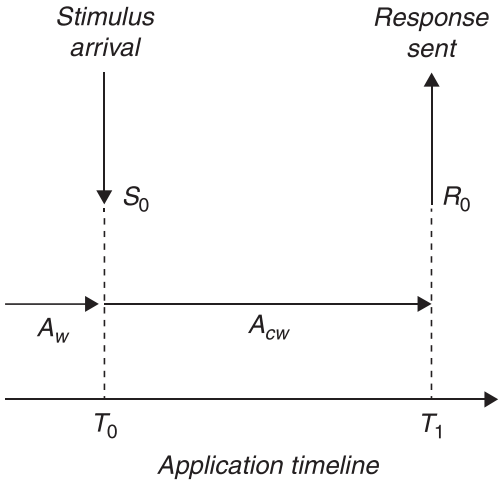


Figure 5.1 An Application Timeline Showing the Arrival of a Stimulus and the Departure of the Response

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

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

By comparing the two timelines (uninterrupted and interrupted) side by side, as shown in Figure 5.3, we can quantify the impact of these STW events. Let's focus on two key metrics: the total work done in both scenarios within the same period and the throughput.

1. Work done:
   - In the uninterrupted scenario, the total work done is $A_{cw}$.
   - In the interrupted scenario, the total work done is the sum of $A_{cw_0}$ and $A_{cw_1}$.
2. Throughput ($W_d$):
   - For the **uninterrupted timeline**, the throughput ($W_{da}$) is calculated as the total work done ($A_{cw}$) divided by the duration from stimulus arrival to response dispatch ($T_1 - T_0$).
   - In the **interrupted timeline**, the throughput ($W_{db}$) is the sum of the work done in each segment ($A_{cw_0} + A_{cw_1}$) divided by the same duration ($T_1 - T_0$).

Given that the interruptions caused by the STW pauses result in $A_{cw_0} + A_{cw_1}$ being less than $A_{cw}$, $W_{db}$ will consequently be lower than $W_{da}$. This effectively illustrates the reduction in throughput during the response window from $T_0$ to $T_1$, highlighting how STW events impact the application's efficiency in processing tasks and ultimately prolong the response time.

Through our exploration of response times, throughput, footprint and availability, it's clear that optimizing software is just one facet of enhancing system performance. As we navigate the nuances of software subsystems, we recognize that the components influencing the 'ilities' of a system extend beyond just the software layer. The hardware foundation on which the JVM and applications operate plays an equally pivotal role in shaping overall performance and reliability.

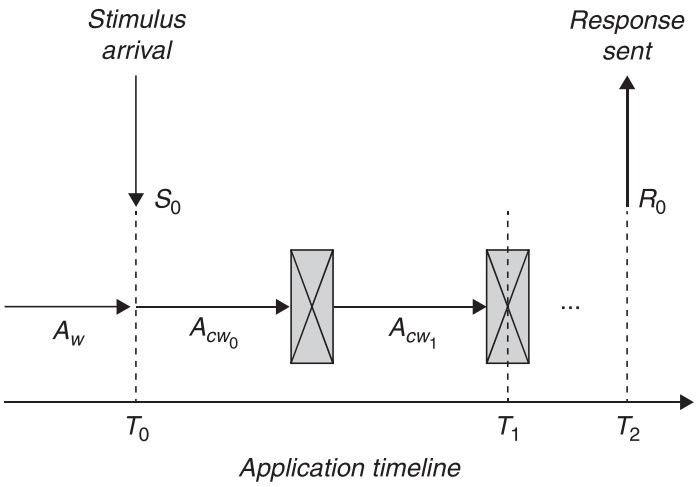


Figure 5.2 An Application Timeline with Stop-the-World Pauses

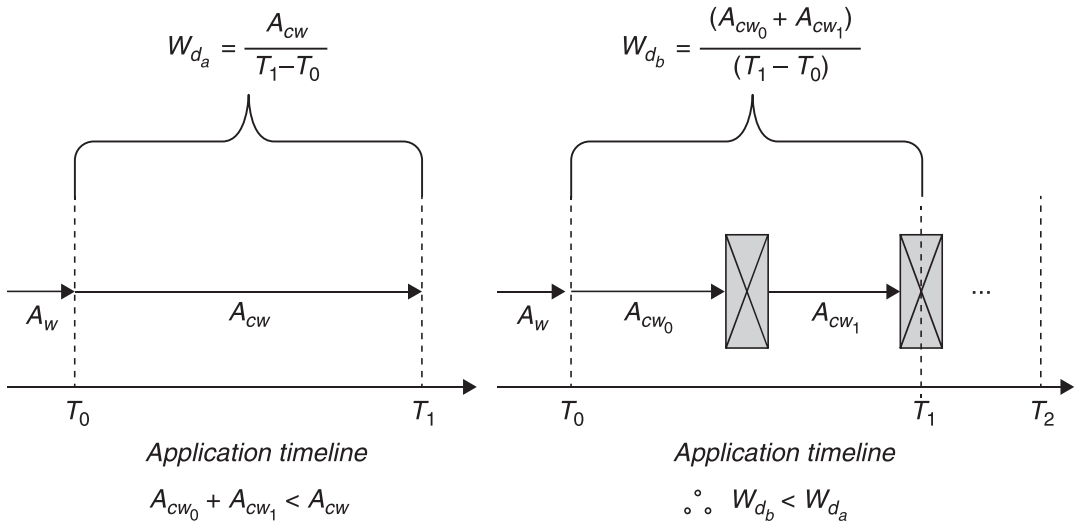


Figure 5.3 Side-by-Side Comparison to Highlight Work Done and Throughput

## The Role of Hardware in Performance

In our exploration of JVM performance, we've traversed key software metrics like responsiveness, throughput, and more. Beyond these software intricacies lies a critical dimension: the underlying hardware.

While software innovations capture much attention, hardware's role in shaping performance is equally crucial. As cloud architectures evolve, the strategic importance of hardware becomes more apparent, with vendors optimizing for both power and cost-efficiency.

Considering environments from data centers to the cloud, the impact of hardware changes—such as processor architecture adjustments or adding processing cores—is profound. How these modifications affect performance metrics or system resilience prompts a deeper investigation. Such inquiries not only enhance our understanding of hardware's influence on performance but also guide effective hardware stack modifications, underscoring the need for a balanced approach to software and hardware optimization in pursuit of technological efficiency.

Building on hardware's pivotal role in system performance, let's consider a containerized Java application scenario using Netty, renowned for its high-performance and nonblocking I/O capabilities. While Netty excels in managing concurrent connections, its efficacy extends beyond software optimization to how it integrates with cloud infrastructure and leverages the capabilities of underlying hardware within the constraints of a container environment.

Containerization introduces its own set of performance considerations, including resource isolation, overhead, and the potential for density-related issues within the host system. Deployed

on cloud platforms, such applications navigate a multilayered architecture: from hypervisors and virtual machines to container orchestration, which manages resources, scaling, and health. Each layer introduces complexities affecting data transfer, latency, and throughput.

Understanding the complexities of cloud and containerized environments is crucial for optimizing application performance, necessitating a strategic balance among software capabilities, hardware resources, and the intermediary stack. This pursuit of optimization is further shaped by global technology standards and regulations, emphasizing the importance of security, data protection, and energy efficiency. Such standards drive compliance and spur hardware-level innovations, therby enhancing virtualization stacks and overall system efficiency.

As we transition into this section, our focus expands to the symbiotic relationship between hardware and software, highlighting how hardware intricacies, from data processing mechanisms to memory management strategies, influence the performance of applications. This sets the stage for a comprehensive exploration in the subsequent sections.

## Decoding Hardware–Software Dynamics

To maximize the performance of an application, it's essential to understand the complex interplay between its code and the underlying hardware. While software provides the user interface and functionality, the hardware plays a crucial role in determining operational speed. This critical relationship, if not properly aligned, can lead to potential performance bottlenecks.

The characteristics of the hardware, such as CPU speed, memory availability, and disk I/O, directly influence how quickly and effectively an application can process tasks. It's imperative for developers and system architects to be aware of the hardware's capabilities and limitations. For example, an application optimized for multi-core processors might not perform as efficiently on a single-core setup. Additionally, the intricacies of micro-architecture, diverse subsystems, and the variables introduced by cloud environments can further complicate the scaling process.

Diving deeper into hardware–software interactions, we find complexities, especially in areas like hardware architecture and thread behavior. For instance, in an ideal world, multiple threads would seamlessly access a shared memory structure, working in harmony without any conflicts. Such a scenario, depicted in Figure 5.4 and inspired by Maranget et al.'s *A Tutorial Introduction to the ARM and POWER Relaxed Memory Models*,[4] while desirable, is often more theoretical than practical.

[4] www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf.

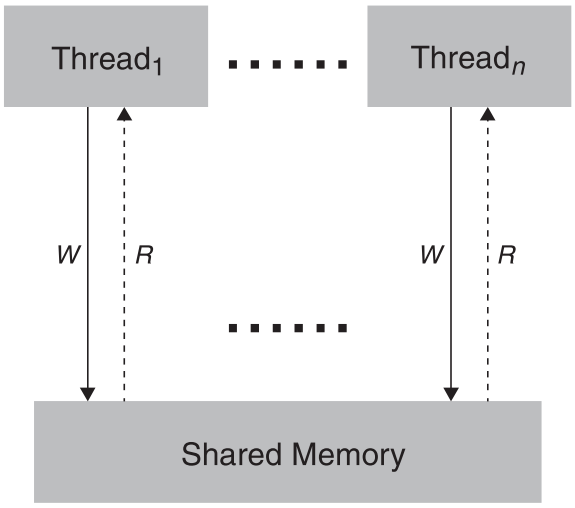


Figure 5.4 Threads 1 to *n* accessing a Shared Memory Structure Without Hindrance (Inspired by Luc Maranget, Susmit Sarkar, and Peter Sewell's *A Tutorial Introduction to the ARM and POWER Relaxed Memory Models*)

However, the real world of computing is riddled with scaling and concurrency challenges. Shared memory management often necessitates mechanisms like locks to ensure data validity. These mechanisms, while essential, can introduce complications. Drawing from Doug Lea's *Concurrent Programming in Java: Design Principles and Patterns*,[5] it's not uncommon for threads to vie for locks when accessing objects (Figure 5.5). This competition can lead to scenarios in which one thread inadvertently blocks others, resulting in execution delays and potential performance degradation.

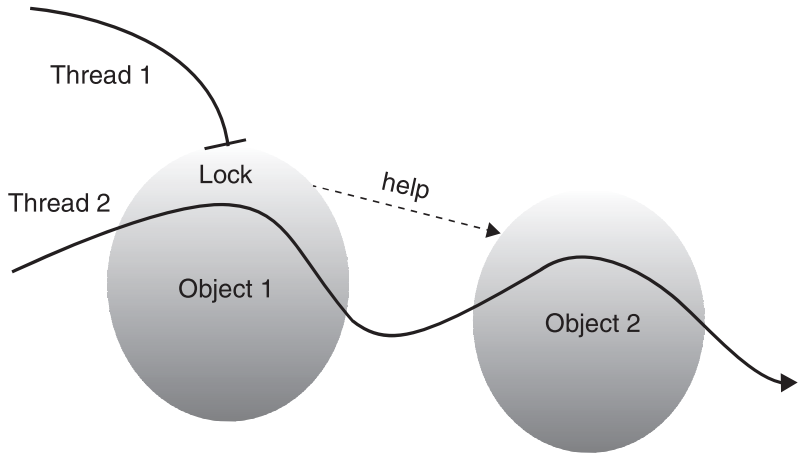


Figure 5.5 Thread 2 Locks Object 1 and Moves to the Helper in Object 2 (Inspired by Doug Lea's *Concurrent Programming in Java: Design Principles and Patterns*, 2nd ed.)

[5] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*, 2nd ed. Boston, MA: Addison-Wesley Professional, 1999.

Such disparities between the idealized vision and the practical reality emphasize the intricate interplay and inherent challenges in hardware–software interactions. Threads, shared memory structures, and the mechanisms that govern their interactions (lock or lock-free data structures and algorithms) are a testament to this complexity. It's this divergence that underscores the need for a deep understanding of how hardware influences application performance.

## Performance Symphony: Languages, Processors, and Memory Models

At its best, performance is a harmonious composition of programming languages employed, the capabilities of the processors, and the memory models they adhere to. This symphony of performance is what we aim to unravel in this section.

### Concurrency: The Core Mechanism

Concurrency refers to the execution of multiple tasks or processes simultaneously. It's a fundamental concept in modern computing, allowing for improved efficiency and responsiveness. From an application standpoint, understanding concurrency is vital for designing systems that can handle multiple users or tasks at once. For web applications, this could mean serving multiple users without delays. Misunderstanding concurrency can lead to issues like deadlocks or race conditions.

### The Symbiotic Relationship: Hardware and Software Interplay

- **Beyond hardware:** While powerful hardware can boost performance, the true potential is unlocked when the software is optimized for that hardware. This optimization is influenced by the choice of programming language, its compatibility with the processor, and adherence to a specific memory model.
- **Languages and processors:** Different programming languages have varying levels of abstraction. Whereas some offer a closer interaction with the hardware, others prioritize developer convenience. The challenge lies in ensuring that the code is optimized across this spectrum, making it performant on intended processor architecture.

### Memory Models: The Silent Catalysts

- **Balancing consistency and performance:** Memory models define how threads perceive memory operations. They strike a delicate balance between ensuring all threads have a coherent view of memory while allowing for certain reorderings for performance gains.
- **Language specifics:** Different languages come with their own memory models. For instance, Java's memory model emphasizes a specific order of operations to maintain consistency across threads, even if it means sacrificing some performance. By comparison, C++11 and newer versions offer a more fine-grained control over memory ordering than Java.

## Enhancing Performance: Optimizing the Harmony

### Hardware-Aware Programming

To truly harness the power of the hardware, we must look beyond the language's abstraction. This involves understanding cache hierarchies, optimizing data structures for cache-friendly design, and being aware of the processor's capabilities. Here's how this plays out across different levels of caching:

- **Optimizing for CPU cache efficiency:** Align data structures with cache lines to minimize cache misses and speed up access. Important strategies include managing "true sharing"—minimizing performance impacts by ensuring shared data on the same cache line is accessed efficiently—and avoiding "false sharing," where unrelated data on the same line leads to invalidations. Taking advantage of hardware prefetching, which preloads data based on anticipated use, improves spatial and temporal locality.
- **Application-level caching strategies:** Implementing caching at the software/application level, demands careful consideration of hardware scalability. The available RAM, for instance, dictates the volume of data that can be retained in memory. Decisions regarding what to cache and its retention duration hinge on a nuanced understanding of hardware capabilities, guided by principles of spatial and temporal locality.
- **Database and web caching:** Databases use memory for caching queries and results, linking performance to hardware resources. Similarly, web caching through content delivery networks (CDNs) involves geographically distributed networked servers to cache content closer to users, reducing latency. Software Defined Networks (SDN) can optimize hardware use in caching strategies for improved content delivery efficiency.

### Mastering Memory Models

A deep understanding of a language's memory model can prevent concurrency issues and unlock performance optimizations. Memory models dictate the rules governing interactions between memory and multiple threads of execution. A nuanced understanding of these models can significantly mitigate concurrency-related issues, such as visibility problems and race conditions, while also unlocking avenues for performance optimization.

- **Java's memory model:** In Java, grasping the intricacies of the *happens-before* relationship is essential. This concept ensures memory visibility and ordering of operations across different threads. By mastering these relationships, developers can write scalable thread-safe code without excessive synchronization. Understanding how to leverage volatile variables, atomic operations, and synchronized blocks effectively can lead to significant performance improvements, especially in multithreaded environments.
- **Implications for software design:** Knowledge of memory models influences decisions on data sharing between threads, use of locks, and implementation of non-blocking algorithms. It enables a more strategic approach to synchronization, helping developers choose the most appropriate technique for each scenario. For instance, in low-latency systems, avoiding heavy locks and using lock-free data structures can be critical.

Having explored the nuances of hardware and memory models that can help us craft highly efficient and resilient concurrent software, our next step is to delve into the intricacies of memory model specifics.

## Memory Models: Deciphering Thread Dynamics and Performance Impacts

Memory models outline the rules governing thread interactions through shared memory and dictate how values propagate between threads. These rules have a significant influence on the system's performance. In an ideal computing world, we would envision the memory model as a sequentially consistent shared memory system. Such a system would ensure the operations are executed in the exact sequence as they appear in the original program order, known as the single, global execution order, resulting in a sequentially consistent machine.

For example, consider a scenario where the program order dictated that Thread 2 would always get the lock on Object 1 first, effectively blocking Thread 1 every time as it moves to the helper in Object 2. This can be visualized with the aid of two complementary diagrammatic representations: a sequence diagram and a Gantt chart.

The sequence diagram shown in Figure 5.6 provides a detailed view of the interactions between threads and objects. It shows Thread 1 initiating its process with a pre-lock operation, immediately followed by Thread 2 acquiring a lock on Object 1. Thread 2's subsequent work on Object 2 and the eventual release of the lock on Object 1 are depicted in a clear, sequential manner. Concurrently, Thread 1 enters a waiting state, demonstrating its dependency on the lock held by Thread 2. Once Thread 2 releases the lock, Thread 1 acquires it, proceeds with its operation on Object 2, and then releases the lock, completing its sequence of operations.

Complementing the sequence diagram, the Gantt chart, shown in Figure 5.7, illustrates the global execution order over a timeline. It presents the start and end points of each thread's interaction with the objects, capturing the essence of sequential consistency. The Gantt chart marks the waiting period of Thread 1, which overlaps with Thread 2's exclusive access period, and then shows the continuation of Thread 1's actions after the lock's release.

By joining the global execution timeline with the program order timeline, we can observe a straightforward, sequential flow of events. This visualization aligns with the principles of a sequentially consistent machine, where despite the concurrent nature of thread operations, the system behaves as if there is a single, global sequence of well-ordered events. The diagrams thus serve to enhance our understanding of sequential consistency.

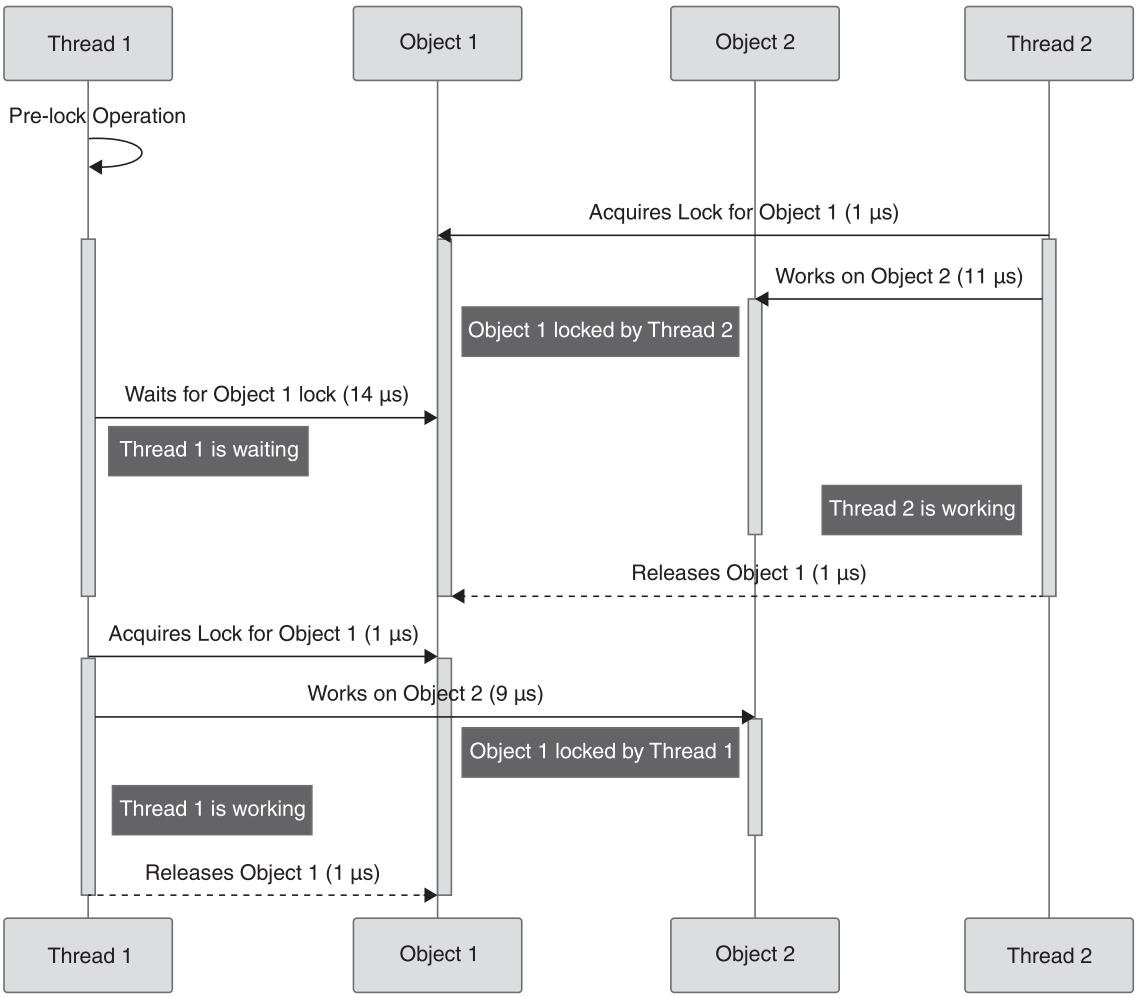


Figure 5.6 Sequence of Thread Operation in a Sequentially Consistent Shared Memory Model

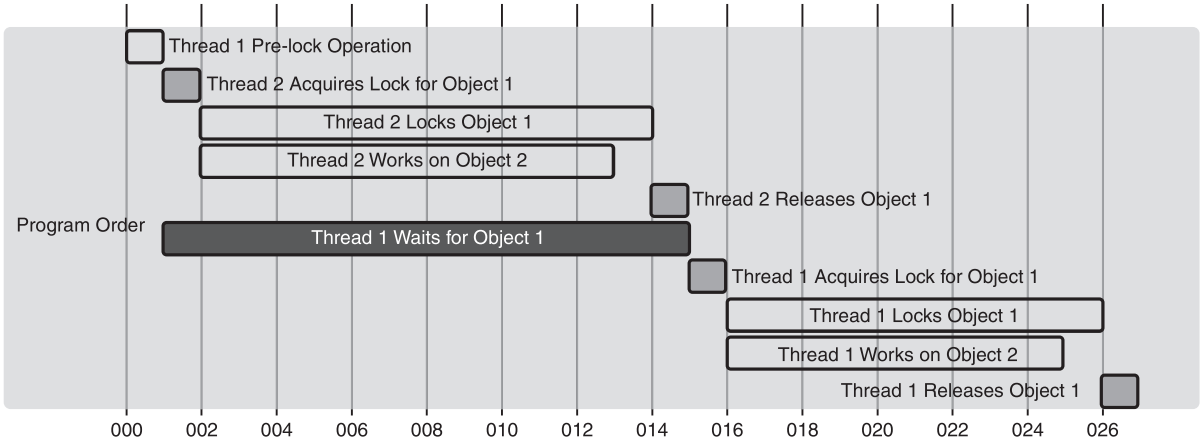


Figure 5.7 Global Execution Timeline of Thread Interactions