While Figures 5.6 and 5.7 and the discussion outline an idealized, sequential consistent system, real-world contemporary computing systems often follow more relaxed memory models such as total store order (TSO), partial store order (PSO), or release consistency. These models, when combined with various hardware optimization strategies such as store/write buffering and out-of-order execution, allow for performance gains. However, they also introduce the possibility of observing operations out of their programmed order, potentially leading to inconsistencies.

Consider the concept of store buffering as an example, where threads may observe their own writes before they become visible to other threads, which can disrupt the perceived order of operations. To illustrate, let's explore this with a store buffering example with two threads (or processes), `p0` and `p1`. Initially, we have two shared variables `X` and `Y` set to `0` in memory. Additionally, we have `foo` and `bar`, which are local variables stored in registers of threads `p0` and `p1`, respectively:

```
p0                  p1
a: X = 1;           c: Y = 1;
b: foo = Y;         d: bar = X;
```

In an ideal, sequentially consistent environment, the values for `foo` and `bar` are determined by combined order of operations that could include sequences like `{a,b,c,d}`, `{c,d,a,b}`, `{a,c,b,d}`, `{c,a,b,d}`, and so on. In such a scenario, it's impossible for both `foo` and `bar` to be zero, as the operations within each thread are guaranteed to be observed in the order they were issued.

Yet, with store buffering, threads might perceive their own writes before others. This can lead to situations in which both `foo` and `bar` are zero, because `p0`'s write to `X` (`a: X = 1`) might be recognized by `p0` before `p1` notices it, and similarly for `p1`'s write to `Y` (`c: Y = 1`).

Although such behaviors can lead to inconsistencies that deviate from the expected sequential order, these optimizations are fundamental for enhancing system performance, paving the way for faster processing times and greater efficiency. Additionally, cache coherence protocols like MESI and MOESI[6] play a crucial role in maintaining consistency across processor caches, further complicating the memory model landscape.

In conclusion, our exploration of memory consistency models has laid the groundwork for understanding the intricacies of concurrent operations within the JVM. For those of us working with Java and JVM performance tuning, it is essential to get a good handle on the practical realities and the underlying principles of these models because they influence the smooth and efficient execution of our concurrent programs.

As we move forward, we will continue to explore the world of hardware. In particular, we'll discuss multiprocessor threads or cores with tiered memory structures alongside the Java Memory Model's (JMM) constructs. We will cover the use of volatile variables - a key JMM feature for crafting thread-safe applications - and explore how concepts like memory barriers and fences contribute to visibility and ordering in concurrent Java environments.

Furthermore, the concept of Non-Uniform Memory Access (NUMA) becomes increasingly relevant as we consider the performance of Java applications on systems with variable memory

[6] https://redis.com/glossary/cache-coherence/

access times. Understanding NUMA's implications is critical, as it affects garbage collection, thread scheduling, and memory allocation strategies within the JVM, influencing the overall performance of Java applications.

Concurrent Hardware: Navigating the Labyrinth

Within the landscape of concurrent hardware systems, we must navigate the complex layers of multiple-processor cores, each of which has its own hierarchical memory structure.

Simultaneous Multithreading and Core Utilization

Multiprocessor cores may implement simultaneous multithreading (SMT), a technique designed to improve the efficiency of CPUs by allowing multiple hardware threads to execute simultaneously on a single core. This concurrency within a single core is designed to better utilize CPU resources, as idle CPU cores can be used by another hardware thread while one thread waits for I/O operations or other long-latency instructions.

Contrasting this with the utilization of full cores, where each thread is run on a separate physical core with dedicated computational resources, we encounter a different set of performance considerations for the JVM. Full cores afford each thread complete command over a core's resources, reducing contention and potentially heightening performance for compute-intensive tasks. In such an environment, JVM's garbage collection and thread scheduling can operate with the assurance of unimpeded access to these resources, tailoring strategies that maximize the efficiency of each core.

However, in the realms where hyper-threading (SMT) is employed, the JVM is presented with a more intricate scenario. It must recognize that threads may not have exclusive access to all the resources of a core. This shared environment can elevate throughput for multithreaded applications but may also introduce a higher likelihood of resource contention. The JVM's thread scheduler and GC must adapt to this reality, balancing workload distribution and GC processes to accommodate the subtleties of SMT.

The choice between leveraging full cores or hyper-threading aligns closely with the nature of the Java application in question. For I/O-bound or multithreaded workloads that can capitalize on idle CPU cycles, hyper-threading may offer a throughput advantage. Conversely, for CPU-bound processes that demand continuous and intensive computation, full cores might provide the best performance outcomes.

As we delve deeper into the implications of these hardware characteristics for JVM performance, we must recognize the importance of aligning our software strategies with the underlying hardware capabilities. Whether optimizing for the dedicated resources of full cores or the concurrent execution environment of SMT, the ultimate goal remains the same: to enhance the performance and responsiveness of Java applications within these concurrent hardware systems.

Advanced Processor Techniques and Memory Hierarchy

The complexity of modern processors extends beyond the realm of SMT and full core utilization. Within these advanced systems, cores have their own unique cache hierarchies to

maintain cache coherence across multiple processing units. Each core typically possesses a Level 1 instruction cache (L1I$), a Level 1 data cache (L1D$), and a private Level 2 cache (L2). Cache coherence is the mechanism that ensures all cores have a consistent view of memory. In essence, when one core updates a value in its cache, other cores are made aware of this change, preventing them from using outdated or inconsistent data.

Included in this setup is a communal last-level cache (LLC) that is typically accessible by all cores, facilitating efficient data sharing among them. Additionally, some high-performance systems incorporate a system-level cache (SLC) alongside or in place of the LLC to further optimize data access times or to serve specific computational demands.

To maximize performance, these processors employ advanced techniques. For example, out-of-order execution allows the processor to execute instructions as soon as the data becomes available, rather than strictly adhering to the original program order. This maximizes resource utilization, improving processor efficiency.

Moreover, as discussed earlier, the processors employ load and store buffers to manage data being transferred to and from memory. These buffers temporarily hold data for memory operations, such as loads and stores, allowing the CPU to continue executing other instructions without waiting for the memory operations to complete. This mechanism smooths out the flow of data between the fast CPU registers and the relatively slower memory, effectively bridging the speed gap and optimizing the processor's performance.

At the heart of the memory hierarchy is a memory controller that oversees the double data rate (DDR) memory banks within the system, orchestrating the data flow to and from physical memory. Figure 5.8 illustrates a comprehensive hardware configuration containing all of the aforementioned components.

Memory consistency, or the manner in which one processor's memory changes are perceived by others, varies across hardware architectures. At one end of the spectrum are architectures with strong memory consistency models like x86-64 and SPARC, which utilize TSO.[7] Conversely, the Arm v8-A architecture has a more relaxed model that allows for more flexible reordering of memory operations. This flexibility can lead to higher performance but requires programmers to be more diligent in using synchronization primitives to avoid unexpected behavior in multi-threaded applications.[8]

To illustrate the challenges, consider the store buffer example discussed earlier with two variables `X` and `Y`, both initially set to `0`. In TSO architectures like x86-64, the utilization of store buffers could lead to both `foo` and `bar` being `0` if thread `P1` reads `Y` before `P2`'s write to `Y` becomes visible, and simultaneously `P2` reads `X` before `P1`'s write to `X` is seen by `P2`. In Arm v8, the hardware could also reorder the operations, potentially leading to even more unexpected outcomes beyond those permissible in TSO.

[7] https://docs.oracle.com/cd/E26502_01/html/E29051/hwovr-15.html

[8] https://community.arm.com/arm-community-blogs/b/infrastructure-solutions-blog/posts/synchronization-overview-and-case-study-on-arm-architecture-563085493

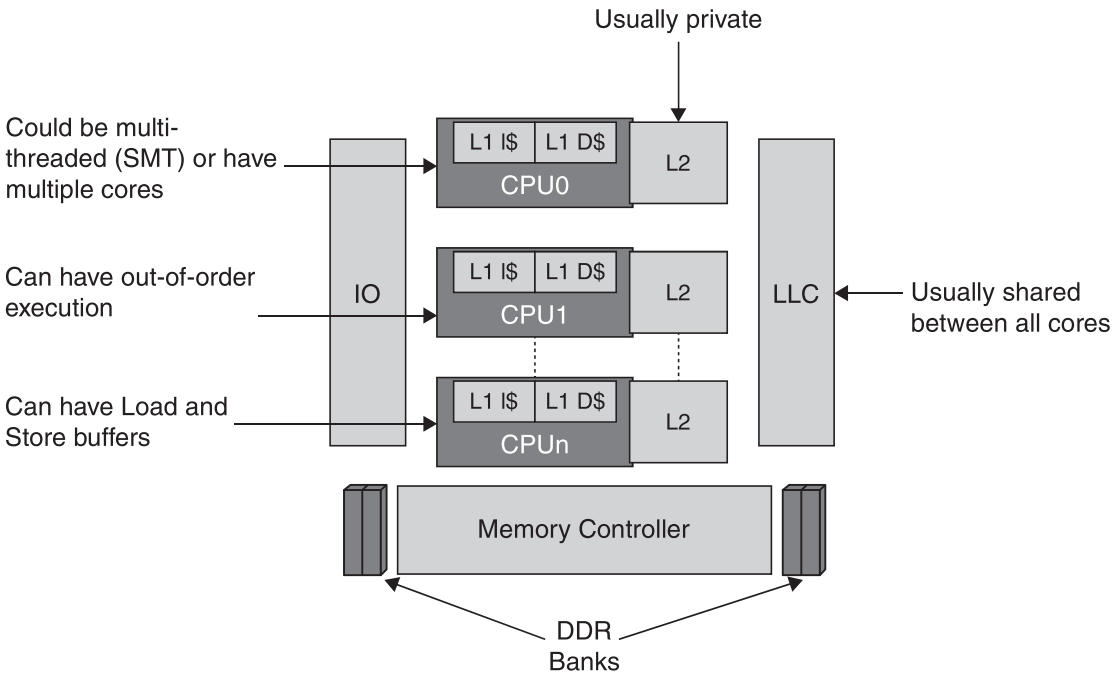

Figure 5.8 A Modern Hardware Configuration

In navigating the labyrinth of advanced processor technologies and memory hierarchies, we have actually uncovered only half the story. The true efficacy of these hardware optimizations is realized when they are effectively synchronized with the software that runs on them. As developers and system architects, our challenge is ensure that our software not only harnesses but also complements these sophisticated hardware capabilities.

To this end, software mechanisms like barriers and fences can both complement and harness these hardware features. These instructions enforce ordering constraints and ensure that operations within a multithreaded environment are executed in a manner that respects the hardware's memory model. This understanding is pivotal for Java performance engineering because it enables us to craft software strategies that optimize the performance of Java applications in complex computing environments. Building on our understanding of Java's memory model and concurrency challenges, let's take a closer look at these software strategies, exploring how they work in tandem with hardware to optimize the efficiency and reliability of Java-based systems.

Order Mechanisms in Concurrent Computing: Barriers, Fences, and Volatiles

In the complex world of concurrent computing, barriers, fences, and volatiles serve as crucial tools to preserve order and ensure data consistency. These mechanisms function by preventing certain types of reordering that could potentially lead to unintended consequences or unexpected outcomes.

Barriers

A barrier is a type of synchronization mechanism used in concurrent programming to block certain kinds of reordering. Essentially, a barrier acts like a checkpoint or synchronization point in your program. When a thread hits a barrier, it cannot pass until all other threads have also reached this barrier. In the context of memory operations, a memory barrier prevents the reordering of read and write operations (loads and stores) across the barrier. This helps in maintaining the consistency of shared data in a multithreaded environment.

Fences

Whereas barriers provide a broad synchronization point, fences are more tailored to memory operations. They enforce ordering constraints, ensuring that certain operations complete before others commence. For instance, a store-load fence ensures that all store operations (writes) that appear before the fence in the program order are visible to the other threads before any load operations (reads) that appear after the fence.

Volatiles

In languages like Java, the `volatile` keyword provides a way to ensure visibility and ordering of variables across threads. With a volatile variable, a write to the variable is visible to all threads that subsequently read from the variable, providing a *happens-before* guarantee. Nonvolatile variables, in contrast, don't have the benefit of the interaction guaranteed by the `volatile` keyword. Hence, the compiler can use a cached value of the nonvolatile variable.

Atomicity in Depth: Java Memory Model and *Happens-Before* Relationship

Atomicity ensures that operations either fully execute or don't execute at all, thereby preserving data integrity. The JMM is instrumental in understanding this concept in a multithreaded context. It provides visibility and ordering guarantees, ensuring atomicity and establishing the *happens-before* relationship for predictable execution. When one action *happens-before* another, the first action is not only visible to the second action but also ordered before it. This relationship ensures that Java programs executed in a multithreaded environment behave predictably and consistently.

Let's consider a practical example that illustrates these concepts in action. The `AtomicLong` class in Java is a prime example of how atomic operations are implemented and utilized in a multithreaded context to maintain data integrity and predictability. `AtomicLong` ensures atomic operations on `long` values through low-level synchronization techniques, acting as single, indivisible units of work. This characteristic protects them from the potential disruption or interference of concurrent operations. Such atomic operations form the bedrock of synchronization, ensuring system-wide coherence even in the face of intense multithreading.

```
AtomicLong counter = new AtomicLong();

void incrementCounter() {
```

```
    counter.incrementAndGet();
}
```

In this code snippet, the `incrementAndGet()` method in the `AtomicLong` class demonstrates atomic operation. It combines the increment (`counter++`) and retrieval (`get`) of the counter value into a single operation. This atomicity is crucial in a multithreaded environment because it prevents other threads from observing or interfering with the `counter` in an inconsistent state during the operation.

The `AtomicLong` class harnesses a low-level technique to ensure this order. It uses compare-and-set loops, rooted in CPU instructions, that effectively act as memory barriers. This atomicity, guaranteed by barriers, is an essential aspect of concurrent programming. As we progress, we'll dig deeper into how these concepts bolster performance engineering and optimization.

Let's consider another practical example to illustrate these concepts, this time involving a hypothetical `DriverLicense` class. This example demonstrates how the `volatile` keyword is used in Java to ensure visibility and ordering of variables across threads:

```
class DriverLicense {
    private volatile boolean validLicense = false;

    void renewLicense() {
        this.validLicense = true;
    }

    boolean isLicenseValid() {
        return this.validLicense;
    }
}
```

In this example, the `DriverLicense` class serves as a model for managing a driver's license status. When the license is renewed (by calling the `renewLicense()` method), the `validLicense` field is set to `true`. The current status can be verified with `isLicenseValid()` method.

The `validLicense` field is declared as `volatile`, which guarantees that once the license is renewed in one thread (for example, after an online renewal), the new status of the license will immediately be visible to all other threads (for example, police officers checking the license status). Without the `volatile` keyword, there might be a delay before other threads could see the updated license status due to various caching or reordering optimizations done by the JVM or the system.

In the context of real-world applications, misunderstanding or improperly implementing these mechanisms can lead to severe consequences. For instance, an e-commerce platform that doesn't handle concurrent transactions properly might double-charge a customer or fail to update inventory correctly. For developers, understanding these mechanisms is essential when designing applications that rely on multithreading. Their proper use can ensure that tasks are completed in the correct order, preventing potential data inconsistencies or errors.

As we advance, our exploration will examine runtime performance through effective thread management and synchronization techniques. Specifically, Chapter 7, "Runtime Performance Optimizations: A Focus on Strings, Locks, and Beyond" will not only introduce advanced synchronization and locking mechanisms but also guide us through the concurrency utilities. By leveraging the strengths of these concurrency frameworks, alongside a foundational understanding of the underlying hardware behavior, we equip ourselves with a comprehensive toolkit for mastering performance optimization in concurrent systems.

Concurrent Memory Access and Coherency in Multiprocessor Systems

In our exploration of memory models and their profound influence on thread dynamics, we've unearthed the intricate challenges that arise when dealing with memory access in multiprocessor systems. These challenges are magnified when multiple cores or threads concurrently access and modify shared data. In such scenarios, it is very important to ensure coherency and efficient memory access.

In the realm of multiprocessor systems, threads often interact with different memory controllers. Although concurrent memory access is a hallmark of these systems, built-in coherency mechanisms ensure that data inconsistencies are kept at bay. For instance, even if one core updates a memory location, coherency protocols will ensure that all cores have a consistent view of the data. This eliminates scenarios where a core might access outdated values. The presence of such robust mechanisms highlights the sophisticated design of modern systems, which prioritize both performance and data consistency.

In multiprocessor systems, ensuring data consistency across threads is a prime concern. Tools like barriers, fences, and volatiles have been foundational in guaranteeing that operations adhere to a specific sequence, maintaining data consistency across threads. These mechanisms are essential in ensuring that memory operations are executed in the correct order, preventing potential *data races* and inconsistencies.

> **NOTE** Data races occur when two or more threads access shared data simultaneously and at least one access is a write operation.

However, with the rise of systems featuring multiple processors and diverse memory banks, another layer of complexity is added—namely, the physical distance and access times between processors and memory. This is where Non-Uniform Memory Access (NUMA) architectures come into the picture. NUMA optimizes memory access based on the system's physical layout. Together, these mechanisms and architectures work in tandem to ensure both data consistency and efficient memory access in modern multiprocessor systems.

NUMA Deep Dive: My Experiences at AMD, Sun Microsystems, and Arm

Building on our understanding of concurrent hardware and software mechanisms, it's time to journey further into the world of NUMA, a journey informed by my firsthand experiences at AMD, Sun Microsystems, and Arm. NUMA is a crucial component in multiprocessing environments, and understanding its nuances can significantly enhance system performance.

Within a NUMA system, the memory controller isn't a mere facilitator—it's the nerve center. At AMD, working on the groundbreaking Opteron processor, I observed how the controller's role was a pivotal function. Such a NUMA-aware, intelligent entity capable of judiciously allocating memory based on the architecture's nuances and the individual processors' needs optimized memory allocation based on proximity to the processors, minimizing cross-node traffic, reducing latency, and enhancing system performance.

Consider this: When a processor requests data, the memory controller decides where the data is stored. If it's local, the controller enables swift, direct access. If not, the controller must engage the interconnection network, thus introducing latency. In my work at AMD, the role of a NUMA-aware memory controller was to strike an optimal balance between these decisions to maximize performance.

Buffers in NUMA systems serve as intermediaries, temporarily storing data during transfers, helping to balance memory access across nodes. This balancing act prevents a single node from being inundated with an excessive number of requests, ensuring a smooth operation—a principle we used to optimize the scheduler for Sun Microsystems' high-performance V40z servers and SPARC T4s.

Attention must also be directed to the transport system—the artery of a NUMA architecture. It enables data mobility across the system. A well-optimized transport system minimizes latency, thereby contributing to the system's overall performance—a key consideration, reinforced during my tenure at Arm.

Mastering NUMA isn't a superficial exercise: It's about understanding the interplay of components such as the memory controllers, buffers, and the transport system within this intricate architecture. This deep understanding has been instrumental in my efforts to optimize performance in concurrent systems.

Figure 5.9 shows a simplified NUMA architecture with buffers and interconnects.

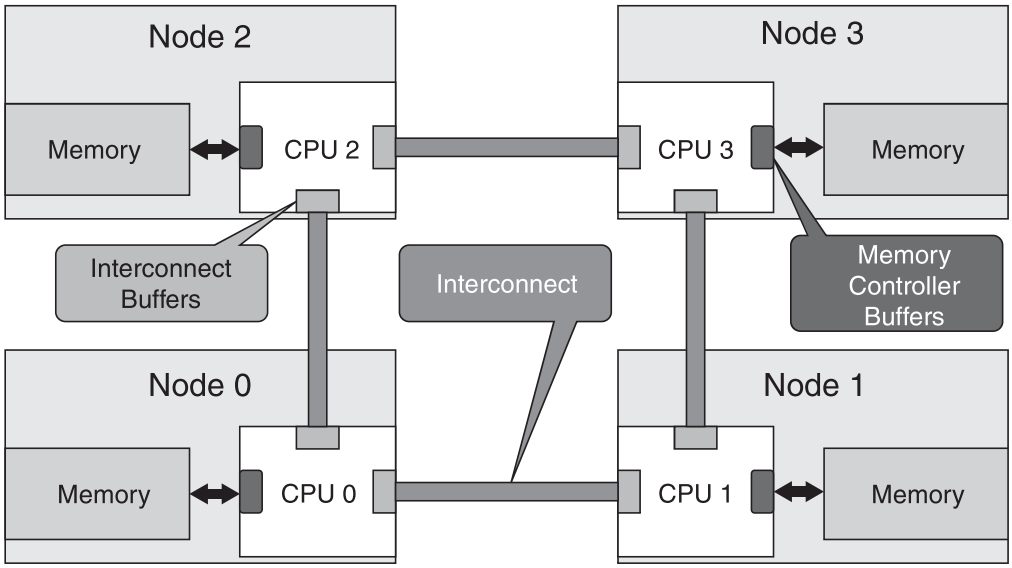

Figure 5.9 NUMA Nodes in Modern Systems

Venturing further into the NUMA landscape, we encounter various critical traffic patterns, including cross-traffic, local traffic, and interleaved memory traffic patterns. These patterns significantly influence memory access latency and, consequently, data processing speed (Figure 5.10).

Cross-traffic refers to instances where a processor accesses data in another processor's memory bank. This cross-node movement introduces latency but is an inevitable reality of multiprocessing systems. A key consideration in such cases is efficient handling through intelligent memory allocation, facilitated by the NUMA-aware memory controller and observing paths with the least latency, without crowding the buffers.

Local traffic, by contrast, occurs when a processor retrieves data from its local memory bank—a process faster than cross-node traffic due to the elimination of the interconnect network's demands. Optimizing a system to increase local traffic while minimizing cross-traffic latencies is a cornerstone of efficient NUMA systems.

Interleaved memory adds another layer of complexity to NUMA systems. Here, successive memory addresses scatter across various memory banks. This scattering balances the load across nodes, reducing bottlenecks and enhancing performance. However, it can inadvertently increase cross-node traffic—a possibility that calls for careful management of the memory controllers.

NUMA architecture addresses the scalability limitations inherent in shared memory models, wherein all processors share a single memory space. As the number of processors increases, the bandwidth to the shared memory becomes a bottleneck, limiting the scalability of the system. However, programming for NUMA systems can be challenging, as developers need to consider memory locality when designing their algorithms and data structures. To assist developers modern operating systems offer NUMA APIs for allocating memory on specific nodes. Additionally, memory management systems, such as databases and GCs, are becoming increasingly NUMA-aware. This awareness is crucial for maximizing system performance and was a key focus in my work at AMD, Sun Microsystems, and Arm.

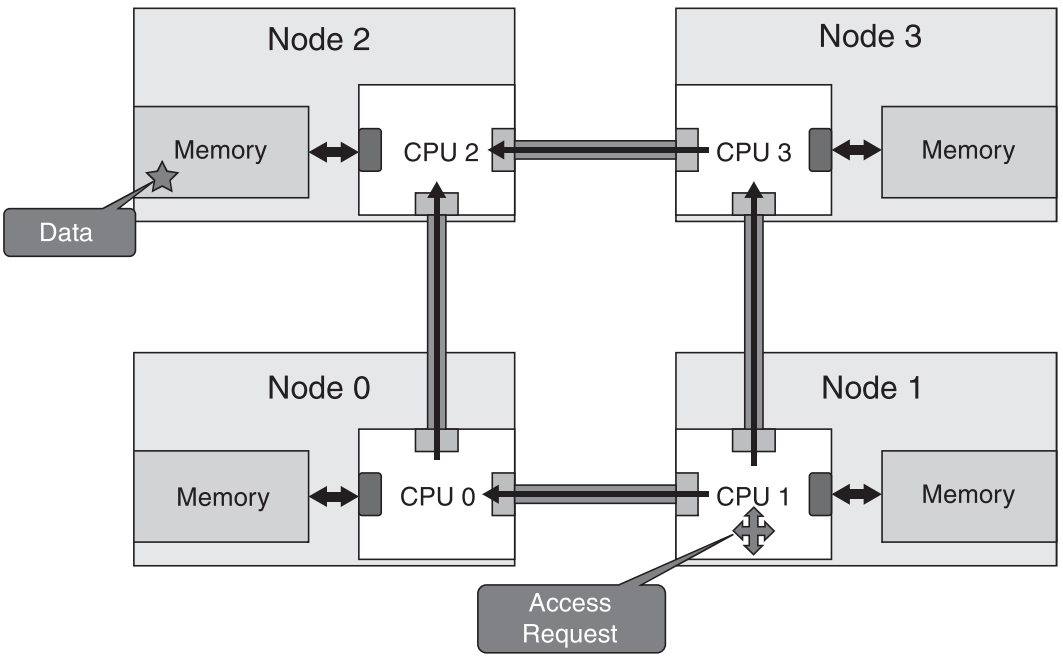

Figure 5.10 An Illustration of NUMA Access

Other related concepts include Cache Coherent Non-Uniform Memory Access (ccNUMA), in which a system maintains a consistent image of memory across all processors; by comparison, NUMA refers to the overarching design philosophy of building systems with non-uniform memory. Understanding these concepts, in addition to barriers, fences, and volatiles, is critical to mastering performance optimization in concurrent systems.

Advancements in NUMA: Fabric Architectures in Modern Processors

The NUMA architecture has undergone many advancements with the evolution of processor designs, which are particularly noticeable in high-end server processors. These innovations are readily evident in the work done by industry giants like Intel, AMD, and Arm, each of which has made unique contributions that are particularly relevant in server environments and cloud computing scenarios.

- **Intel's mesh fabric:** The Intel Xeon Scalable processors have embraced a mesh architecture, which replaces the company's traditional ring interconnect.[9] This mesh topology effectively connects cores, caches, memory, and I/O components in a more direct manner. This allows for higher bandwidth and lower latency, both of which are crucial for data center and high-performance computing applications. This shift significantly bolsters data-processing capabilities and system responsiveness in complex computational environments.
- **AMD's infinity fabric:** AMD's EPYC processors feature the Infinity Fabric architecture.[10] This technology interconnects multiple processor dies within the same package, enabling efficient data transfer and scalability. It's particularly beneficial in multiple-die CPUs, addressing challenges related to exceptional scaling and performance in multi-core environments. AMD's approach dramatically improves data handling in large-scale servers, which is critical to meet today's growing computational demands.
- **Arm's NUMA advances:** Arm has also made strides in NUMA technology, particularly in terms of its server-grade CPUs. Arm architectures like the Neoverse N1 platform[11] embody a scalable architecture for high-throughput, parallel-processing tasks with an emphasis on an energy-efficient NUMA design. The intelligent interconnect architecture optimizes memory accesses across different cores, balancing performance with power efficiency, and thereby improving performance per watts for densely populated servers.

From Pioneering Foundations to Modern Innovations: NUMA, JVM, and Beyond

Around 20 years ago, during my tenure at AMD, our team embarked on a groundbreaking journey. We ventured beyond the conventional, seeking to bridge the intricate world of NUMA with the dynamic realm of JVM performance. This early collaboration culminated in the inception of the NUMA-aware GC, a significant innovation to strategically optimize memory management in such a multi-node system. This was a testament to the foresight and creativity

[9] www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html

[10] www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/AMD-Optimizes-EPYC-Memory-With-NUMA.pdf

[11] www.arm.com/-/media/global/solutions/infrastructure/arm-neoverse-n1-platform.pdf

of that era. The insights we garnered then weren't just momentary flashes of brilliance; they laid a robust foundation for the JVM landscape we're familiar with today. Today, we're building upon that legacy, enhancing various GCs with the wisdom and insights from our foundational work. In Chapter 6, "Advanced Memory Management and Garbage Collection in OpenJDK," you'll witness how these early innovations continue to shape and influence modern JVM performance.

Bridging Theory and Practice: Concurrency, Libraries, and Advanced Tooling

As we shift our focus to practical application of these foundational theories, the insights gained from understanding the architectural nuances and concurrency paradigms will help us build tailor-made frameworks and libraries. These can take advantage of the SMT or per-core scaling and prime the software cache to complement hardware cache/memory hierarchies. Armed with this knowledge, we should be able to harmonize our systems with the underlying architectural model, using it to our advantage with respect to thread pools or to harness concurrent, lock-free algorithms. This approach, a vital aspect of cloud computing, is a perfect example of what it means to be cloud-native in application and libraries development.

The JVM ecosystem comes with advanced tools for monitoring and profiling, which can be instrumental in translating these theoretical concepts into actionable insights. Tools like Java Mission Control and VisualVM allow developers to peek under the hood of the JVM, offering a detailed understanding of how applications interact with the JVM and underlying hardware. These insights are crucial for identifying bottlenecks and optimizing performance.

Performance Engineering Methodology: A Dynamic and Detailed Approach

When we examine the previous sections through the lens of application development, two common threads emerge: optimization and efficiency. While these concepts might initially seem abstract or overly technical, they are central to determining how well an application performs, its scalability, and its reliability. Developers focused solely on the application layer might inadvertently underestimate the significance of these foundational concepts. However, a holistic understanding of both the application itself and the underlying system infrastructure can ensure that the resulting software is robust, efficient, and capable of meeting user demands.

Performance engineering—a multidimensional, cyclical discipline—necessitates a deep comprehension of not just the application, JVM, and OS, but also the underlying hardware. In today's diverse technological landscape, this domain extends to include modern deployment environments such as containers and virtual machines managed by hypervisors. These elements, which are integral to cloud computing and virtualized infrastructures, add layers of complexity and opportunity to the performance engineering process.

Within this expanded domain, we can employ diverse methodologies, such as top-down and bottom-up approaches, to uncover, diagnose, and address performance issues entrenched in systems, whether they are running on traditional setups or modern, distributed architectures. Alongside these approaches, the technique of experimental design—although not a traditional methodology—plays a pivotal role, offering a systematic procedure for conducting experiments and evaluating system performance from baremetal servers to containerized microservices. These approaches and techniques are universally applicable, ensuring that performance optimization remains achievable regardless of the deployment scenarios.

Experimental Design

Experimental design is a systematic approach used to test hypotheses about system performance. It emphasizes data collection and promotes evidence-based decision making. This method is employed in both top-down and bottom-up methodologies to establish baseline and peak or stress performance measurements and also to precisely evaluate the impact of incremental or specific enhancements. The baseline performance represents the system's capabilities under normal or expected loads, whereas the peak or stress performance highlights its resilience under extreme loads. Experimental design aids in distinguishing variable changes, assessing performance impacts, and gathering crucial monitoring and profiling data.

Inherent to experimental design are the control and treatment conditions. The control condition represents the baseline scenario, often corresponding to the system's performance under a typical load, such as with the current software version. In contrast, the treatment condition introduces experimental variables, such as a different software version or load, to observe their impact. The comparison between the control and the treatment conditions helps gauge the impact of the experimental variables on system performance.

One specific type of experimental design is A/B testing, in which two versions of a product or system are compared to determine which performs better in terms of a specific metric. While all A/B tests are a form of experimental design, not all experimental designs are A/B tests. The broader field of experimental design can involve multiple conditions and is used across various domains, from marketing to medicine.

A crucial aspect of setting up these conditions involves understanding and controlling the state of the system. For instance, when measuring the impact of a new JVM feature, it's essential to ensure that the JVM's state is consistent across all trials. This could include the state of the GC, the state of JVM-managed threads, the interaction with the OS, and more. This consistency ensures reliable results and accurate interpretations.

Bottom-Up Methodology

The bottom-up methodology begins at the granularity of low-level system components and gradually works its way up to provide detailed insights into system performance. Throughout my experiences at AMD, Sun, Oracle, Arm, and now at Microsoft as a JVM system and GC performance engineer, the bottom-up methodology has been a crucial tool in my repertoire. Its utility has been evident in a multitude of scenarios, from introducing novel features at the JVM/runtime level and evaluating the implications of Metaspace, to fine-tuning thresholds

for the G1 GC's mixed collections. As we'll see in the upcoming JMH benchmarking section, the bottom-up approach was also critical in understanding how a new Arm processor's atomic instruction set could influence the system's scalability and performance.

Like experimental design, the bottom-up approach requires a thorough understanding of how state is managed across different levels of the system, extending from hardware to high-level application frameworks:

- **Hardware nuances:** It's essential to comprehend how processor architectures, memory hierarchies, and I/O subsystems impact the JVM behavior and performance. For example, understanding how a processor's core and cache architecture influences garbage collection and thread scheduling in the JVM can lead to more informed decisions in system tuning.
- **JVM's GC:** Developers must recognize how the JVM manages heap memory, particularly how garbage collection strategies like G1 and low-latency collectors like Shenandoah and ZGC are optimized for different hardware, OS, and workload patterns. For instance, ZGC enhances performance by utilizing large memory pages and employing multi-mapping techniques, with tailored optimizations that adapt to the specific memory management characteristics and architectural nuances of different operating systems and hardware platforms.
- **OS threads management:** Developers should explore how the OS manages threads in conjunction with the JVM's multithreading capabilities. This includes the nuances of thread scheduling and the effects of context switching on JVM performance.
- **Containerized environments:**
  - **Resource allocation and limits:** Analyzing container orchestration platforms like Kubernetes for their resource allocation strategies—from managing CPU and memory requests and limits to their interaction with the underlying hardware resources. These factors influence crucial JVM configurations and impact their performance within containers.
  - **Network and I/O operations:** Evaluating the impact of container networking on JVM network-intensive applications and the effects of I/O operations on JVM's file handling.
- **JVM tuning in containers:** We should also consider the specifics of JVM tuning in container environments, such as adjusting heap size with respect to the container's memory limits. In a resource-restricted environment, ensuring that the JVM is properly tuned within the container is crucial.
- **Application frameworks:** Developers must understand how frameworks like Netty, used for high-performance network applications, manage asynchronous tasks and resource utilization. In a bottom-up analysis, we would scrutinize Netty's thread management, buffer allocation strategies, and how these interact with JVM's non-blocking I/O operations and GC behavior.

The efficacy of the bottom-up methodology becomes particularly evident when it's integrated with benchmarks. This combination enables precise measurement of performance and scalability across the entire diverse technology stack—from the hardware and the OS, up through the JVM, container orchestration platforms, and cloud infrastructure, and finally to

the benchmark itself. This comprehensive approach provides an in-depth understanding of the performance landscape, particularly highlighting the crucial interactions of the JVM with the container within the cloud OS, the memory policies, and the hardware specifics. These insights, in turn, are instrumental in tuning the system for optimal performance. Moreover, the bottom-up approach has played an instrumental role in enhancing the out-of-box user experience for JVM and GCs, reinforcing its importance in the realm of performance engineering. We will explore this pragmatic methodology more thoroughly in Chapter 7, when we discuss performance improvements related to contented locks.

In contrast, the top-down methodology, our primary focus in the next sections, is particularly valuable when operating under a definitive statement of work (SoW). A SoW is a carefully crafted document that delineates the responsibilities of a vendor or service provider. It outlines the expected work activities, specific deliverables, and timeline for execution, thereby serving as a guide/roadmap for the performance engineering endeavor. In the context of performance engineering, the SoW provides a detailed layout for achieving the system's quality driven SLAs. This methodology, therefore, enables system-specific optimizations to be aligned coherently with the overarching goals as outlined in the SoW.

Taken together, these methodologies and techniques equip a performance engineer with a comprehensive toolbox, facilitating the effective enhancement of system throughput and the extension of its capabilities.

Top-Down Methodology

The top-down methodology is an overarching approach that begins with a comprehensive understanding of the system's high-level objectives and works downward, leveraging our in-depth understanding of the holistic stack. The method thrives when we have a wide spectrum of knowledge—from the high-level application architecture and user interaction patterns to the finer details of hardware behavior and system configuration.

The approach benefits from its focus on "known-knowns"—the familiar forces at play, including operational workflows, access patterns, data layouts, application pressures, and tunables. These "known-knowns" represent aspects or details about the system that we already understand well. They might include the structure of the system, how its various components interact, and the general behavior of the system under normal conditions. The strength of the top-down methodology lies in its ability to reveal the "known-unknowns"—that is, aspects or details about the system that we know exist, but we don't understand well yet. They might include potential performance bottlenecks, unpredictable system behaviors under certain conditions, or the impact of specific system changes on performance.

> **NOTE** The investigative process embedded in the top-down methodology aims to transform the "known-unknowns" into "known-knowns," thereby providing the knowledge necessary for continual system optimization. While our existing domain knowledge of the subsystems provides a foundational understanding, effectively identifying and addressing various bottlenecks requires more detailed and specific information that goes beyond this initial knowledge to help us achieve our performance and scalability goals.

Applying the top-down methodology demands a firm grasp of how state is managed across different system components, from the higher-level application state down to the finer details of memory management by the JVM and the OS. These factors can significantly influence the system's overall behavior. For instance, when bringing up a new hardware system, the top-down approach starts by defining the performance objectives and requirements of the hardware, followed by investigating the "known-unknowns," such as scalability, and optimal deployment scenarios. Similarly, transitioning from an on-premises setup to a cloud environment begins with the overarching goals of the migration, such as cost-efficiency and low overhead, and then addresses how these objectives impact application deployment, JVM tuning, and resource allocation. Even when you're introducing a new library into your application ecosystem, you should employ the top-down methodology, beginning with the desired outcomes of the integration and then examining the library's interactions with existing components and their effects on application behavior.

Building a Statement of Work

As we transition to constructing a statement of work, we will see how the top-down methodology shapes our approach to optimizing complex systems like large-scale online transactional processing (OLTP) database systems. Generally, such systems, which are characterized by their extensive, distributed data footprints, require careful optimization to ensure efficient performance. Key to this optimization, especially in the case of our specific OLTP system, is the effective cache utilization and multithreading capabilities, which can significantly reduce the system's *call chain traversal time*.

Call chain traversal time is a key metric in OLTP systems, representing the cumulative time required to complete a series of function calls and returns with in a program. This metric is calculated by summing the duration of all individual function calls in the chain. In environments where rapid processing of transactions is essential, minimizing this time can significantly enhance the system's performance.

Creating a SoW for such a complex system involves laying out clear performance and capacity objectives, identifying possible bottlenecks, and specifying the timeline and deliverables. In our OLTP system, it also requires understanding how state is managed within the system, especially when the OLTP system handles a multitude of transactions and needs to maintain consistent stateful interactions. How this state is managed across various components—from the application level to the JVM, OS, and even at the I/O process level—has a significant bearing on the system's performance. The SoW acts as a strategic roadmap, guiding our use of the top-down methodology in investigating and resolving performance issues.

The OLTP system establishes object data relationships for rapid retrievals. The goal is not only to enhance transactional latency but also to balance this with the need to reduce *tail latencies* and increase system utilization during peak loads. Tail latency refers to the delay experienced at the worst percentile of a distribution (for example, the 99th percentile and above). Reducing tail latency means making the system's worst-case performance better, which can greatly improve the overall user experience.

Broadly speaking, OLTP systems often aim to improve responsiveness and efficiency simultaneously. They strive to ensure that each transaction is processed swiftly (enhancing responsiveness) and that the system can handle an increasing number of these transactions, especially during periods of high demand (ensuring efficiency). This dual focus is crucial for the system's overall operational effectiveness, which in turn makes it an essential aspect of the performance engineering process.

Balancing responsiveness with system efficiency is essential. If a system is responsive but not scalable, it might process individual tasks quickly, but it could become overwhelmed when the number of tasks increases. Conversely, a system that is scalable but not performant could handle a large number of tasks, but if each task takes too long to process, the overall system efficiency will still be compromised. Therefore, the ideal system excels in both aspects.

In the context of real-world Java workloads, these considerations are particularly relevant. Java-based systems often deal with high transaction volumes and require careful tuning to ensure optimal performance and capacity. The focus on reducing tail latencies, enhancing transactional latency, and increasing system utilization during peak loads are all critical aspects of managing Java workloads effectively.

Having established the clear performance and scalability goals in our SoW, we now turn to the process that will enable us to meet these objectives. In the next section, we take a detailed look at the performance engineering process, systematically exploring the various facets and subsystems at play.

The Performance Engineering Process: A Top-Down Approach

In this section, we will embark on our exploration of performance engineering with a systematic, top-down approach. This methodology, chosen for its comprehensive coverage of system layers, begins at the highest system level and subsequently drills down into more specific subsystems and components. The cyclic nature of this methodology involves four crucial phases:

1. **Performance monitoring:** Tracking the application's performance metrics over its lifespan to identify potential areas for improvement.
2. **Profiling:** Carrying out targeted profiling for each pattern of interest as identified during monitoring. Patterns may need to be repeated to gauge their impact accurately.
3. **Analysis:** Interpreting the results from the profiling phase and formulating a plan of action. The analysis phase often goes hand in hand with profiling.
4. **Apply tunings:** Applying the tunings suggested by the analysis phase.

The iterative nature of this process ensures that we refine our understanding of the system, continuously improve performance, and meet our scalability goals. Here, it's important to remember that hardware monitoring can play a critical role. Data collected on CPU utilization, memory usage, network I/O, disk I/O, and other metrics can provide valuable insights into the system's performance. These findings can guide the profiling, analysis, and tuning stages, helping us understand whether a performance issue is related to the software or due to hardware limitations.

Building on the Statement of Work: Subsystems Under Investigation

In this section, we apply our top-down performance methodology to each layer of the modern application stack, represented in Figure 5.11. This systematic approach allows us to enhance the call chain traversal times, mitigate tail latencies, and increase system utilization during peak loads.

Our journey through the stack begins with the *Application* layer, where we assess its functional performance and high-level characteristics. We then explore the *Application Ecosystem*, focusing on the libraries and frameworks that support the application's operations. Next, we focus on the *JVM's* execution capabilities and the *JRE*, responsible for providing the necessary libraries and resources for execution. These components could be encapsulated within the *Container*, ensuring that the application runs quickly and reliably from one computing environment to another.

Beneath the container lies the *(Guest) OS*, tailored to the containerized setting. It is followed by the *Hypervisor* along with the *Host OS* for virtualized infrastructures, which manage the containers and provide them access to the physical *Hardware* resources.

We evaluate the overall system performance by leveraging various tools and techniques specific to each layer to determine where potential bottlenecks may lie and look for opportunities to make broad-stroke improvements. Our analysis encompasses all elements, from hardware to software, providing us with a holistic view of the system. Each layer's analysis, guided by the goals outlined in our SoW, provides insights into potential performance bottlenecks and optimization opportunities. By systematically exploring each layer, we aim to enhance the overall performance and scalability of the system, aligning with the top-down methodology's objective to transform our system-level understanding into targeted, actionable insights.

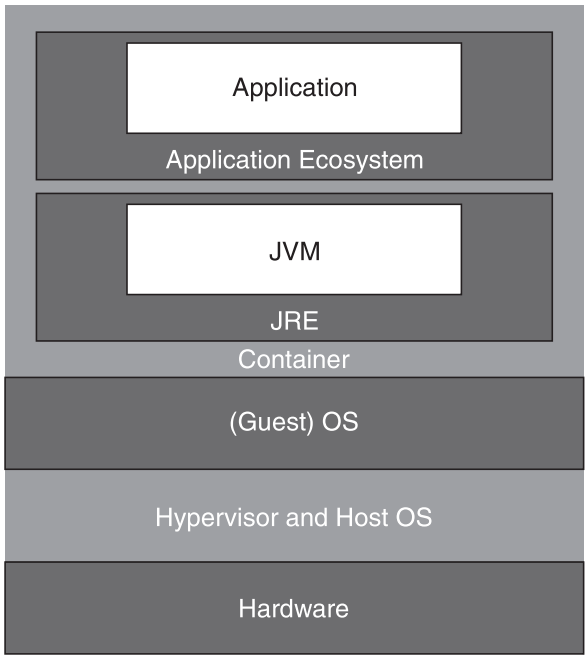

Figure 5.11 A Simplified Illustration of Different Layers of a Typical System Stack

The Application and Its Ecosystem: A System-Wide Outlook

In line with our SoW for the Java-centric OLTP database system, we'll now focus on the application and its ecosystem for a thorough system-wide examination. Our analysis encompasses the following key components:

- **Core database services and functions:** These are the fundamental operations that a database system performs, such as data storage, retrieval, update, and delete operations.
- **Commit log handling:** This refers to the management of commit logs, which record all transactions that modify the data in a database.
- **Data caches:** These hardware or software components store data so future requests for that data can be served faster, which can significantly enhance the system's performance.
- **Connection pools:** These caches of database connections are maintained and optimized for reuse. This strategy promotes efficient utilization of resources, as it mitigates the overhead of establishing a new connection every time database access is required.
- **Middleware/APIs:** Middleware serves as a software bridge between an OS and the applications that run on it, facilitating seamless communications and interactions. In tandem, application programming interfaces (APIs) provide a structured set of rules and protocols for constructing and interfacing with software applications. Collectively, middleware and APIs bolster the system's interoperability, ensuring effective and efficient interaction among its various components.
- **Storage engine:** This is the underlying software component that handles data storage and retrieval in the database system. Optimizing the storage engine can lead to significant performance improvements.
- **Schema:** This is the organization or structure for a database, defined in a formal language supported by the database system. An efficient schema design can improve data retrieval times and overall system performance.

System Infrastructure and Interactions

In a software system, libraries are collections of precompiled code that developers can reuse to perform common tasks. Libraries and their interactions play a crucial role in the performance of a system. Interactions between libraries refer to how these different sets of precompiled code work together to execute more complex tasks. Examining these interactions can provide important insights into system performance, as problems or inefficiencies in these interactions could potentially slow down the system or cause it to behave in unexpected ways.

Now, let's link these libraries and their interactions to the broader context of the operational categories of a database system. Each of the components in the database system contributes to the following:

1. **Data preprocessing tasks:** Operations such as standardization and validation, which are crucial for maintaining data integrity and consistency in OLTP systems. For these tasks, we aim to identify the most resource-intensive transactions and map their paths. We'll also try to understand the libraries involved and how they interact with one another.