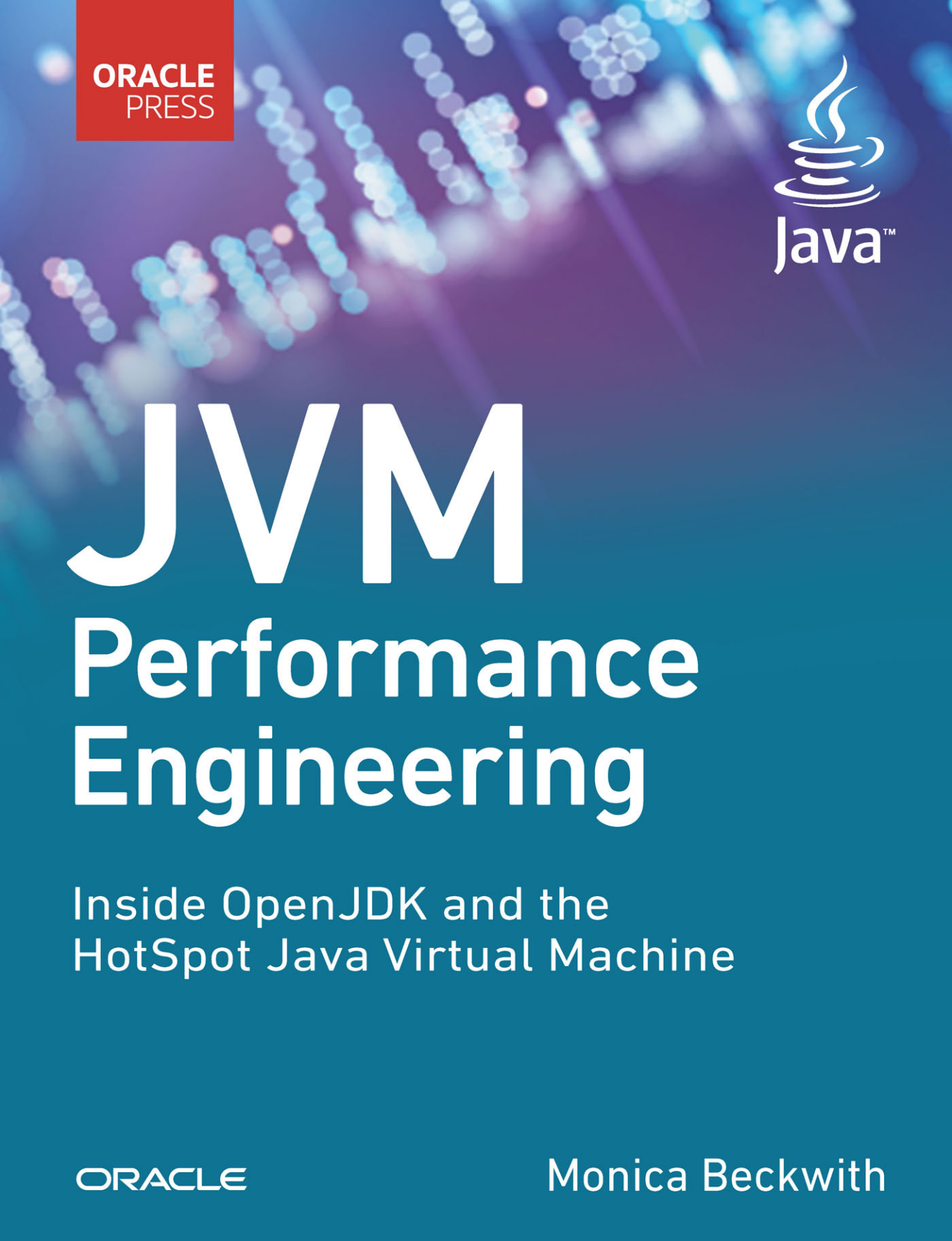
ORACLE
PRESS
Java™
JVM
Performance
Engineering
Inside OpenJDK and the
HotSpot Java Virtual Machine
ORACLE
Monica Beckwith

JVM Performance Engineering

This page intentionally left blank

JVM Performance Engineering

Inside OpenJDK and the HotSpot Java Virtual Machine

Monica Beckwith

Addison-Wesley
Hoboken, New Jersey

Cover image: Amiak / Shutterstock
Figures 7.8–7.18, 7.22–7.29: The Apache Software Foundation

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2024930211

ISBN-13: 978-0-13-465987-9
ISBN-10: 0-13-465987-2

$PrintCode

To my cherished companions, who have provided endless inspiration and comfort throughout the journey of writing this book:

In loving memory of Perl, Sherekhan, Cami, Mr. Spots, and Ruby. Their memories continue to guide and brighten my days with their lasting legacy of love and warmth.

And to Delphi, Calypso, Ivy, Selene, and little Bash, who continue to fill my life with joy, curiosity, and playful adventures. Their presence brings daily reminders of the beauty and wonder in the world around us.

This book is a tribute to all of them—those who have passed and those who are still by my side—celebrating the unconditional love and irreplaceable companionship they have graciously shared with me.

This page intentionally left blank

Contents

Preface

Welcome to my guide to JVM performance engineering, distilled from more than 20 years of expertise as a Java Champion and performance engineer. Within these pages lies a journey through the evolution of the JVM—a narrative that unfolds Java's robust capabilities and architectural prowess. This book meticulously navigates the intricacies of JVM internals and the art and science of performance engineering, examining everything from the inner workings of the HotSpot VM to the strategic adoption of modular programming. By asserting Java's pivotal role in modern computing—from server environments to the integration with exotic hardware—it stands as a beacon for practitioners and enthusiasts alike, heralding the next frontier in JVM performance engineering.

Intended Audience

This book is primarily written for Java developers and software engineers who are keen to enhance their understanding of JVM internals and performance tuning. It will also greatly benefit system architects and designers, providing them with insights into JVM's impact on system performance. Performance engineers and JVM tuners will find advanced techniques for optimizing JVM performance. Additionally, computer science and engineering students and educators will gain a comprehensive understanding of JVM's complexities and advanced features.

With the hope of furthering education in performance engineering, particularly with a focus on the JVM, this text also aligns with advanced courses on programming languages, algorithms, systems, computer architectures, and software engineering. I am passionate about fostering a deeper understanding of these concepts and excited about contributing to coursework that integrates the principles of JVM performance engineering and prepares the next generation of engineers with the knowledge and skills to excel in this critical area of technology.

Focusing on the intricacies and strengths of the language and runtime, this book offers a thorough dissection of Java's capabilities in concurrency, its strengths in multithreading, and the sophisticated memory management mechanisms that drive peak performance across varied environments.

Book Organization

Chapter 1, "The Performance Evolution of Java: The Language and the Virtual Machine," expertly traces Java's journey from its inception in the mid-1990s to the sophisticated advancements in Java 17. Highlighting Java's groundbreaking runtime environment, complete with the JVM, expansive class libraries, and a formidable set of tools, the chapter sets the stage for Java's innovative advancements, underlying technical excellence, continuous progress, and flexibility.

Key highlights include an examination of the OpenJDK HotSpot VM's transformative garbage collectors (GCs) and streamlined Java bytecode. This section illustrates Java's dedication to

performance, showcasing advanced JIT compilation and avant-garde optimization techniques. Additionally, the chapter explores the synergistic relationship between the HotSpot VM's client and server compilers, and their dynamic optimization capabilities, demonstrating Java's continuous pursuit of agility and efficiency.

Another focal point is the exploration of OpenJDK's memory management with the HotSpot GCs, particularly highlighting the adoption of the "weak generational hypothesis." This concept underpins the efficiency of collectors in HotSpot, employing parallel and concurrent GC threads as needed, ensuring peak memory optimization and application responsiveness.

The chapter maintains a balance between technical depth and accessibility, making it suitable for both seasoned Java developers and those new to the language. Practical examples and code snippets are interspersed to provide a hands-on understanding of the concepts discussed.

Chapter 2, "Performance Implications of Java's Type System Evolution," seamlessly continues from the performance focus of Chapter 1, delving into the heart of Java: its evolving type system. The chapter explores Java's foundational elements—primitive and reference types, interfaces, classes, and arrays—that anchored Java programming prior to Java SE 5.0.

The narrative continues with the transformative enhancements from Java SE 5.0 onward, such as the introduction of generics, annotations, and VarHandle type reference—all further enriching the language. The chapter spotlights recent additions such as switch expressions, sealed classes, and the much-anticipated records.

Special attention is given to Project Valhalla's ongoing work, examining the performance nuances of the existing type system and the potential of future value classes. The section offers insights into Project Valhalla's ongoing endeavors, from refined generics to the conceptualization of classes for basic primitives.

Java's type system is more than just a set of types—it's a reflection of Java's commitment to versatility, efficiency, and innovation. The goal of this chapter is to illuminate the type system's past, present, and promising future, fostering a profound understanding of its intricacies.

Chapter 3, "From Monolithic to Modular Java: A Retrospective and Ongoing Evolution," provides extensive coverage of the Java Platform Module System (JPMS) and its breakthrough impact on modular programming. This chapter marks Java's bold transition into the modular era, beginning with a fundamental exploration of modules. It offers hands-on guidance through the creation, compilation, and execution of modules, making it accessible even to newcomers in this domain.

Highlighting Java's transition from a monolithic JDK to a modular framework, the chapter reflects Java's adaptability to evolving needs and its commitment to innovation. A standout section of this chapter is the practical implementation of modular services using JDK 17, which navigates the intricacies of module interactions, from service providers to consumers, enriched by working examples. The chapter addresses key concepts like encapsulation of implementation details and the challenges of JAR hell, illustrating how Jigsaw layers offer elegant solutions in the modular landscape.

Further enriching this exploration, the chapter draws insightful comparisons with OSGi, spotlighting the parallels and distinctions, to give readers a comprehensive understanding of Java's

modular systems. The introduction of essential tools such as *jdeps*, *jlink*, *jdeprscan*, and *jmod*, integral to the modular ecosystem, is accompanied by thorough explanations and practical examples. This approach empowers readers to effectively utilize these tools in their developmental work.

Concluding with a reflection on the performance nuances of JPMS, the chapter looks forward to the future of Java's modular evolution, inviting readers to contemplate its potential impacts and developments.

Chapter 4, "The Unified Java Virtual Machine Logging Interface," delves into the vital yet often underappreciated world of logs in software development. It begins by underscoring the necessity of a unified logging system in Java, addressing the challenges posed by disparate logging systems and the myriad benefits of a cohesive approach. The chapter not only highlights the unification and infrastructure of the logging system but also emphasizes its role in monitoring performance and optimization.

The narrative explores the vast array of log tags and their specific roles, emphasizing the importance of creating comprehensive and insightful logs. In tackling the challenges of discerning any missing information, the chapter provides a lucid understanding of log levels, outputs, and decorators. The intricacies of these features are meticulously examined, with practical examples illuminating their application in tangible scenarios.

A key aspect of this chapter is the exploration of asynchronous logging, a critical feature for enhancing log performance with minimal impact on application efficiency. This feature is essential for developers seeking to balance comprehensive logging with system performance.

Concluding the chapter, the importance of logs as a diagnostic tool is emphasized, showcasing their role in both proactive system monitoring and reactive problem-solving. Chapter 4 not only highlights the power of effective logging in Java, but also underscores its significance in building and maintaining robust applications. This chapter reinforces the theme of Java's ongoing evolution, showcasing how advancements in logging contribute significantly to the language's capability and versatility in application development.

Chapter 5, "End-to-End Java Performance Optimization: Engineering Techniques and Microbenchmarking with JMH," focuses on the essence of performance engineering within the Java ecosystem. Emphasizing that performance transcends mere speed, this chapter highlights its critical role in crafting an unparalleled user experience. It commences with a formative exploration of performance engineering's pivotal role within the broader software development realm, highlighting its status as a fundamental quality attribute and unraveling its multifaceted layers.

With precision, the chapter delineates the metrics pivotal to gauging Java's performance, encompassing aspects from footprint to the nuances of availability, ensuring readers grasp the full spectrum of performance dynamics. Stepping in further. It explores the intricacies of response time and its symbiotic relationship with availability. This inspection provides insights into the mechanics of application timelines, intricately weaving the narrative of response time, throughput, and the inevitable pauses that punctuate them.

Yet, the performance narrative is only complete by acknowledging the profound influence of hardware. This chapter decodes the symbiotic relationship between hardware and software,

emphasizing the harmonious symphony that arises from the confluence of languages, processors, and memory models. From the subtleties of memory models and their bearing on thread dynamics to the foundational principles of Java Memory Model, this chapter journeys through the maze of concurrent hardware, shedding light on the order mechanisms pivotal to concurrent computing.

Moving beyond theoretical discussions, this chapter draws on over two decades of hands-on experience in performance optimization. It introduces a systematic approach to performance diagnostics and analysis, offering insights into methodologies and a detailed investigation of subsystems and approaches to identifying potential performance issues. The methodologies are not only vital for software developers focused on performance optimization but also provide valuable insights into the intricate relationship between underlying hardware, software stacks, and application performance.

The chapter emphasizes the importance of a structured benchmarking regime, encompassing everything from memory management to the assessment of feature releases and system layers. This sets the stage for the Java Micro-Benchmark Suite (JMH), the *pièce de résistance* of JVM benchmarking. From its foundational setup to the intricacies of its myriad features, the journey encompasses the genesis of writing benchmarks, to their execution, enriched with insights into benchmarking modes, profilers, and JMH's pivotal annotations.

Chapter 5 thus serves as a comprehensive guide to end-to-end Java performance optimization and as a launchpad for further chapters. It inspires a fervor for relentless optimization and arms readers with the knowledge and tools required to unlock Java's unparalleled performance potential.

Memory management is the silent guardian of Java applications, often operating behind the scenes but crucial to their success. **Chapter 6**, "Advanced Memory Management and Garbage Collection in OpenJDK," marks a deep dive into specialized JVM improvements, showcasing advanced performance tools and techniques. This chapter offers a leap into the world of garbage collection, unraveling the techniques and innovations that ensure Java applications run efficiently and effectively.

The chapter commences with a foundational overview of garbage collection in Java, setting the stage for the detailed exploration of Thread-Local Allocation Buffers (TLABs) and Promotion Local Allocation Buffers (PLABs), and elucidating their pivotal roles in memory management. As we progress, the chapter sheds light on optimizing memory access, emphasizing the significance of the NUMA-aware garbage collection and its impact on performance.

The highlight of this chapter lies in its exploration of advanced garbage collection techniques. The narrative reviews the G1 Garbage Collector (G1 GC), unraveling its revolutionary approach to heap management. From grasping the advantages of a regionalized heap to optimizing G1 GC parameters for peak performance, this section promises a holistic cognizance of one of Java's most advanced garbage collectors. Additionally, the Z Garbage Collector (ZGC) is presented as a technological marvel with its adaptive optimization techniques, and the advancements that make it a game-changer in real-time applications.

This chapter also offers insights into the emerging trends in garbage collection, setting the stage for what lies ahead. Practicality remains at the forefront, with a dedicated section offering invaluable tips for evaluating GC performance. From sympathizing with various workloads, such as Online Analytical Processing (OLAP) to Online Transaction Processing (OLTP) and Hybrid Transactional/Analytical Processing (HTAP), to synthesizing live data set pressure and data lifespan patterns, the chapter equips readers with the apparatus and knowledge to optimize memory management effectively. This chapter is an accessible guide to advanced garbage collection techniques that Java professionals need to navigate the topography of memory management.

Chapter 7, "Runtime Performance Optimizations: A Focus on Strings, Locks, and Beyond," is dedicated to exploring the critical facets of Java's runtime performance, particularly in the realms of string handling and lock synchronization—two areas essential for efficient application performance.

The chapter excels at taking a comprehensive approach to demystifying these JVM optimizations through detailed under-the-hood analysis—utilizing a range of profiling techniques, from bytecode analysis to memory and sample-based profiling to gathering call stack views of profiled methods—to enrich the reader's understanding. Additionally, the chapter leverages JMH benchmarking to highlight the tangible improvements such optimizations bring. The practical use of *async-profiler* for method-level insights and NetBeans memory profiler further enhances the reader's granular understanding of the JVM enhancements. This chapter aims to test and illuminate the optimizations, equipping readers with a comprehensive approach to using these tools effectively, thereby building on the performance engineering methodologies and processes discussed in Chapter 5.

The journey continues with an extensive review of the string optimizations in Java, highlighting major advancements across various Java versions, and then shifts focus onto enhanced multithreading performance, highlighting Java's thread synchronization mechanisms.

Further, the chapter helps navigate the world of concurrency, with discussion of the transition from the thread-per-task model to the scalable thread-per-request model. The examination of Java's Executor Service, ThreadPools, ForkJoinPool framework, and CompletableFuture ensures a robust comprehension of Java's concurrency mechanisms.

The chapter concludes with a glimpse into the future of concurrency in Java with virtual threads. From understanding virtual threads and their carriers to discussing parallelism and integration with existing APIs, this chapter is a practical guide to advanced concurrency mechanisms and string optimizations in Java.

Chapter 8, "Accelerating Time to Steady State with OpenJDK HotSpot VM," is dedicated to optimizing start-up to steady-state performance, crucial for transient applications such as containerized environments, serverless architectures, and microservices. The chapter emphasizes the importance of minimizing JVM start-up and warm-up time to enhance efficient execution, incorporating a pivotal exploration into GraalVM's revolutionary role in this domain.

The narrative dissects the phases of JVM start-up and the journey to an application's steady-state, highlighting the significance of managing state during these phases across various

architectures. An in-depth look at Class Data Sharing (CDS) sheds light on shared archive files and memory mapping, underscoring the advantages in multi-instance setups. The narrative then shifts to ahead-of-time (AOT) compilation, contrasting it with just-in-time (JIT) compilation and detailing the transformative impact of HotSpot VM's Project Leyden and its forecasted ability to manage states via CDS and AOT. This sets the stage for GraalVM and its revolutionary impact on Java's performance landscape. By harnessing advanced optimization techniques, including static images and dynamic compilation, GraalVM enhances performance for a wide array of applications. The exploration of cutting-edge technologies like GraalVM alongside a holistic survey of OpenJDK projects such as CRIU and CraC, which introduce groundbreaking checkpoint/restore functionality, adds depth to the discussion. This comprehensive coverage provides insights into the evolving strategies for optimizing Java applications, making this chapter an invaluable resource for developers looking to navigate today's cloud native environments.

The final chapter, **Chapter 9**, "Harnessing Exotic Hardware: The Future of JVM Performance Engineering," focuses on the fascinating intersection of exotic hardware and the JVM, illuminating its galvanizing impact on performance engineering. This chapter begins with an introduction to the increasingly prominent world of exotic hardware, particularly within cloud environments. It explores the integration of this hardware with the JVM, underscoring the pivotal role of language design and toolchains in this process.

Through a series of carefully detailed case studies, the chapter showcases the real-world applications and challenges of integrating such hardware accelerators. From the Lightweight Java Game Library (LWJGL), to the innovative Aparapi, which bridges Java and OpenCL, each study offers valuable insights into the complexities and triumphs of these integrations. The chapter also examines Project Sumatra's significant contributions to this realm and introduces TornadoVM, a specialized JVM tailored for hardware accelerators.

Through these case studies, the symbiotic potential of integrating exotic hardware with the JVM becomes increasingly evident, leading up to an overview of Project Panama, heralding a new horizon in JVM performance engineering. At the heart of Project Panama lies the Vector API, a symbol of innovation designed for vector computations. This API is not just about computations—it's about ensuring they are efficiently vectorized and tailored for hardware that thrives on vector operations. This ensures that developers have the tools to express parallel computations optimized for diverse hardware architectures. But Panama isn't just about vectors. The Foreign Function and Memory API emerges as a pivotal tool, a bridge that allows Java to converse seamlessly with native libraries. This is Java's answer to the age-old challenge of interoperability, ensuring Java applications can interface effortlessly with native code, breaking language barriers.

Yet, the integration is no walk in the park. From managing intricate memory access patterns to deciphering hardware-specific behaviors, the path to optimization is laden with complexities. But these challenges drive innovation, pushing the boundaries of what's possible. Looking to the future, the chapter showcases my vision of Project Panama as the gold standard for JVM interoperability. The horizon looks promising, with Panama poised to redefine performance and efficiency for Java applications.

This isn't just about the present or the imminent future. The world of JVM performance engineering is on the cusp of a revolution. Innovations are knocking at our door, waiting to be embraced—with Tornado VM's Hybrid APIs, and with HAT toolkit and Project Babylon on the horizon.

How to Use This Book

1. *Sequential Reading for Comprehensive Understanding:* This book is designed to be read from beginning to end, as each chapter builds upon the knowledge of the previous ones. This approach is especially recommended for readers new to JVM performance engineering.
2. *Modular Approach for Specific Topics:* Experienced readers may prefer to jump directly to chapters that address their specific interests or challenges. The table of contents and index can guide you to relevant sections.
3. *Practical Examples and Code:* Throughout the book, practical examples and code snippets are provided to illustrate key concepts. To get the most out of these examples, readers are encouraged to build on and run the code themselves. (See item 5.)
4. *Visual Aids for Enhanced Understanding:* In addition to written explanations, this book employs a variety of textual and visual aids to deepen your understanding.
    a. *Case Studies:* Real-world scenarios that demonstrate the application of JVM performance techniques.
    b. *Screenshots:* Visual outputs depicting profiling results as well as various GC plots, which are essential for understanding the GC process and phases.
    c. *Use-Case Diagrams:* Visual representations that map out the system's functional requirements, showing how different entities interact with each other.
    d. *Block Diagrams:* Illustrations that outline the architecture of a particular JVM or system component, highlighting performance features.
    e. *Class Diagrams:* Detailed object-oriented designs of various code examples, showing relationships and hierarchies.
    f. *Process Flowcharts:* Step-by-step diagrams that walk you through various performance optimization processes and components.
    g. *Timelines:* Visual representations of the different phases or state changes in an activity and the sequence of actions that are taken.
5. *Utilizing the Companion GitHub Repository:* A significant portion of the book's value lies in its practical application. To facilitate this, I have created JVM Performance Engineering GitHub Repository (https://github.com/mo-beck/JVM-Performance-Engineering). Here, you will find
    a. *Complete Code Listings:* All the code snippets and scripts mentioned in the book are available. This allows you to see the code and experiment with it. Use it as a launchpad for your projects and fork and improve it.
    b. *Additional Resources and Updates:* The field of JVM Performance Engineering is ever evolving. The repository will be periodically updated with new scripts, resources, and information to keep you abreast of the latest developments.
    c. *Interactive Learning:* Engage with the material by cloning the repository, running the GC scripts against your GC log files, and modifying them to see how outcomes better suit your GC learning and understanding journey.

6. *Engage with the Community:* I encourage readers to engage with the wider community. Use the GitHub repository to contribute your ideas, ask questions, and share your insights. This collaborative approach enriches the learning experience for everyone involved.
7. *Feedback and Suggestions:* Your feedback is invaluable. If you have suggestions, corrections, or insights, I warmly invite you to share them. You can provide feedback via the GitHub repository, via email (jvmbook@codekaram.com), or via social media platforms (https://www.linkedin.com/in/monicabeckwith/ or https://twitter.com/JVMPerfEngineer).

In Java's vast realm, my tale takes wing,
A narrative so vivid, of wonders I sing.
Distributed systems, both near and afar,
With JVM shining—the brightest star!

Its rise through the ages, a saga profound,
With each chronicle, inquiries resound.
"Where lies the wisdom, the legends so grand?"
They ask with a fervor, eager to understand.

This book is a beacon for all who pursue,
A tapestry of insights, both aged and new.
In chapters that flow, like streams to the seas,
I share my heart's journey, my tech odyssey.

—Monica Beckwith

Acknowledgments

Reflecting on the journey of creating this book, my heart is full of gratitude for the many individuals whose support, expertise, and encouragement have been the wind beneath my wings.

At the forefront of my gratitude is my family—the unwavering pillars of support. To my husband, Ben: Your understanding and belief in my work, coupled with your boundless love and care, have been the bedrock of my perseverance.

To my children, Annika and Bodin: Your patience and resilience have been my inspiration. Balancing the demands of a teen's life with the years it took to bring this book to fruition, you have shown a maturity and understanding well beyond your years. Your support, whether it be a kind word at just the right moment or understanding my need for quiet as I wrestled with complex ideas, has meant more to me than words can express. Your unwavering faith, even when my work required sacrifices from us all, has been a source of strength and motivation. I am incredibly proud of the kind and supportive individuals you are becoming, and I hope this book reflects the values we cherish as a family.

Editorial Guidance

A special word of thanks goes to my executive editor at Pearson, Greg Doench, whose patience has been nothing short of saintly. Over the years, through health challenges, the dynamic nature of JVM release cycles and project developments, and the unprecedented times of COVID, Greg has been a beacon of encouragement. His unwavering support and absence of frustration in the face of my tardiness have been nothing less than extraordinary. Greg, your steadfast presence and guidance have not only helped shape this manuscript but have also been a personal comfort.

Chapter Contributions

The richness of this book's content is a culmination of my extensive work, research, and insights in the field, enriched further by the invaluable contributions of various experts, colleagues, collaborators, and friends. Their collective knowledge, feedback, and support have been instrumental in adding depth and clarity to each topic discussed, reflecting years of dedicated expertise in this domain.

- In Chapter 3, Nikita Lipski's deep experience in Java modularity added compelling depth, particularly on the topics of the JAR hell versioning issues, layers, and his remarkable insights on OSGi.
- Stefano Doni's enriched field expertise in Quality of Service (QoS), performance stack, and theoretical expertise in operational laws and queueing, significantly enhanced Chapter 5, bringing a blend of theoretical and practical perspectives.
- The insights and collaborative interactions with Per Liden and Stefan Karlsson were crucial in refining my exploration of the Z Garbage Collector (ZGC) in Chapter 6. Per's

numerous talks and blog posts have also been instrumental in helping the community understand the intricacies of ZGC in greater detail.

- Chapter 7 benefitted from the combined insights of Alan Bateman and Heinz Kabutz. Alan was instrumental in helping me refine this chapter's coverage of Java's locking mechanisms and virtual threads. His insights helped clarify complex concepts, added depth to the discussion of monitor locks, and provided valuable perspective on the evolution of Java's concurrency model. Heinz's thorough review ensured the relevance and accuracy of the content.
- For Chapter 8, Ludovic Henry's insistence on clarity with respect to the various terminologies and persistence to include advanced topics and Alina Yurenko's insights into GraalVM and its future developments provided depth and foresight and reshaped the chapter to its glorious state today.

  Alina has also influenced me to track the developments in GraalVM—especially the introduction of layered native images, which promises to reduce build times and enable sharing of base images.
- Last but not the least, for Chapter 9, I am grateful to Gary Frost for his thorough review of Aparapi, Project Sumatra, and insights on leveraging the latest JDK (early access) version for developing projects like Project Panama. Dr. Juan Fumero's leadership in the development of TornadoVM and insights into parallel programming challenges have been instrumental in providing relevant insights for my chapter, deepening its clarity, and enhancing its narrative.

  It was a revelation to see our visions converge and witness these industry stalwarts drive the enhancements in the integration of Java with modern hardware accelerators.

Mentors, Influencers, and Friends

Several mentors, leaders, and friends have significantly influenced my broader understanding of technology:

- Charlie Hunt's guidance in my GC performance engineering journey has been foundational. His groundbreaking work on String density has inspired many of my own approaches with methodologies and process. His seminal work *Java Performance* is an essential resource for all performance enthusiasts and is highly recommended for its depth and insight.
- Gil Tene's work on the C4 Garbage Collector and his educational contributions have deeply influenced my perspective on low pause collectors and their interactive nature. I value our check-ins, which I took as mentorship opportunities to learn from one of the brightest minds.
- Thomas Schatzl's generous insights on the G1 Garbage Collector have added depth and context to this area of study, enriching my understanding following my earlier work on G1 GC. Thomas is a GC performance expert whose work, including on Parallel GC, continues to inspire me.

- Vladimir Kozlov's leadership and work in various aspects of the HotSpot JVM have been crucial in pushing the boundaries of Java's performance capabilities. I cherish our work together on prefetching, tiered thresholds, various code generation, and JVM optimizations, and I appreciate his dedication to HotSpot VM.
- Kirk Pepperdine, for our ongoing collaborations that span from the early days of developing G1 GC parser scripts for our joint hands-on lab sessions at JavaOne, to our recent methodologies, processes, and benchmarking endeavors at Microsoft, continuously pushes the envelope in performance engineering.
- Sergey Kuksenko and Alexey Shipilev, along with my fellow JVM performance engineering experts, have been my comrades in relentless pursuit of Java performance optimizations.
- Erik Österlund's development of generational ZGC represents an exciting and forward-looking aspect of garbage collection technology.
- John Rose, for his unparalleled expertise in JVM internals and his pivotal role in the evolution of Java as a language and platform. His vision and deep technical knowledge have not only propelled the field forward but also provided me with invaluable insights throughout my career.

Each of these individuals has not only contributed to the technical depth and richness of this book but also played a vital role in my personal and professional growth. Their collective wisdom, expertise, and support have been instrumental in shaping both the content and the journey of this book, reflecting the collaborative spirit of the Java community.

This page intentionally left blank

About the Author

Monica Beckwith is a leading figure in Java Virtual Machine (JVM) performance tuning and optimizations. With a strong Electrical and Computer Engineering academic foundation, Monica has carved out an illustrious, impactful, and inspiring professional journey.

At Advanced Micro Devices (AMD), Monica refined her expertise in Java, JVM, and systems performance engineering. Her work brought critical insights to NUMA's architectural enhancements, improving both hardware and JVM performance through optimized code generation, improved footprint and advanced JVM techniques, and memory management. She continued her professional growth at Sun Microsystems, contributing significantly to JVM performance enhancements across Sun SPARC, Solaris, and Linux, aiding in the evolution of a scalable Java ecosystem.

Monica's role as a Java Champion and coauthor of *Java Performance Companion*, as well as authoring this current book, highlight her steadfast commitment to the Java community. Notably, her work in the optimization of G1 Garbage Collector went beyond optimization; she delved into diagnosing pain points, fine-tuning processes, and identifying critical areas for enhancement, thereby setting new precedents in JVM performance. Her expertise not only elevated the efficiency of the G1 GC but also showcased her intricate knowledge of JVM's complexities. At Arm, as a managed runtimes performance architect, Monica played a key role in shaping a unified strategy for the Arm ecosystem, fostering a competitive edge for performance on Arm-based servers.

Monica's significant contributions and thought leadership have enriched the broader tech community. Monica serves on the program committee for various prestigious conferences and hosts JVM and performance-themed tracks, further emphasizing her commitment to knowledge sharing and community building.

At Microsoft, Monica's expertise shines brightly as she optimizes JVM-based workloads, applications, and key services, across a diverse range of deployment scenarios, from bare metal to sophisticated Azure VMs. Her deep-seated understanding of hardware and software engineering, combined with her adeptness in systems engineering and benchmarking principles, uniquely positions her at the critical juncture of the hardware and software. This position enables her to significantly contribute to the performance, scalability and power efficiency characterization, evaluation, and analysis of both current and emerging hardware systems within the Azure Compute infrastructure.

Beyond her technical prowess, Monica embodies values that resonate deeply with those around her. She is a beacon of integrity, authenticity, and continuous learning. Her belief in the transformative power of actions, the sanctity of reflection, and the profound impact of empathy defines her interactions and approach. A passionate speaker, Monica's commitment to lifelong learning is evident in her zeal for delivering talks and disseminating knowledge.

Outside the confines of the tech world, Monica's dedication extends to nurturing young minds as a First Lego League coach. This multifaceted persona, combined with her roles as a Java Champion, author, and performance engineer at Microsoft, cements her reputation as a respected figure in the tech community and a source of inspiration for many.

This page intentionally left blank

Chapter 1

The Performance Evolution of Java: The Language and the Virtual Machine

More than three decades ago, the programming languages landscape was largely defined by C and its object-oriented extension, C++. In this period, the world of computing was undergoing a significant shift from large, cumbersome mainframes to smaller, more efficient minicomputers. C, with its suitability for Unix systems, and C++, with its innovative introduction of classes for object-oriented design, were at the forefront of this technological evolution.

However, as the industry started to shift toward more specialized and cost-effective systems, such as microcontrollers and microcomputers, a new set of challenges emerged. Applications were ballooning in terms of lines of code, and the need to "port" software to various platforms became an increasingly pressing concern. This often necessitated rewriting or heavily modifying the application for each specific target, a labor-intensive and error-prone process. Developers also faced the complexities of managing numerous static library dependencies and the demand for lightweight software on embedded systems—areas where C++ fell short.

It was against this backdrop that Java emerged in the mid-1990s. Its creators aimed to fill this niche by offering a "write once, run anywhere" solution. But Java was more than just a programming language. It introduced its own runtime environment, complete with a virtual machine (Java Virtual Machine [JVM]), class libraries, and a comprehensive set of tools. This all-encompassing ecosystem, known as the Java Development Kit (JDK), was designed to tackle the challenges of the era and set the stage for the future of programming. Today, more than a quarter of a century later, Java's influence in the world of programming languages remains strong, a testament to its adaptability and the robustness of its design.

The performance of applications emerged as a critical factor during this time, especially with the rise of large-scale, data-intensive applications. The evolution of Java's runtime system has played a pivotal role in addressing these performance challenges. Thanks to the optimization in generics, autoboxing and unboxing, and enhancements to the concurrency utilities, Java applications have seen significant improvements in both performance and scalability. Moreover, the changes have had far-reaching implications for the performance of the JVM itself. In

particular, the JVM has had to adapt and optimize its execution strategies to efficiently handle these new language features. As you read this book, bear in mind the historical context and the driving forces that led to Java's inception. The evolution of Java and its virtual machine have profoundly influenced the way developers write and optimize software for various platforms.

In this chapter, we will thoroughly examine the history of Java and JVM, highlighting the technological advancements and key milestones that have significantly shaped its development. From its early days as a solution for platform independence, through the introduction of new language features, to the ongoing improvements to the JVM, Java has evolved into a powerful and versatile tool in the arsenal of modern software development.

A New Ecosystem Is Born

In the 1990s, the internet was emerging, and web pages became more interactive with the introduction of Java applets. Java applets were small applications that ran within web browsers, providing a "real-time" experience for end users.

Applets were not only platform independent but also "secure," in the sense that the user needed to trust the applet writer. When discussing security in the context of the JVM, it's essential to understand that direct access to memory should be forbidden. As a result, Java introduced its own memory management system, called the garbage collector (GC).

> **NOTE** In this book, the acronym *GC* is used to refer to both *garbage collection,* the process of automatic memory management, and *garbage collector*, the module within the JVM that performs this process. The specific meaning will be clear based on the context in which *GC* is used.

Additionally, an abstraction layer, known as Java bytecode, was added to any executable. Java applets quickly gained popularity because their bytecode, residing on the web server, would be transferred and executed as its own process during web page rendering. Although the Java bytecode is platform independent, it is interpreted and compiled into native code specific to the underlying platform.

A Few Pages from History

The JDK included tools such as a Java compiler that translated Java code into Java bytecode. Java bytecode is the executable handled by the Java Runtime Environment (JRE). Thus, for different environments, only the runtime needed to be updated. As long as a JVM for a specific environment existed, the bytecode could be executed. The JVM and the GC served as the execution engines. For Java versions 1.0 and 1.1, the bytecode was interpreted to the native machine code, and there was no dynamic compilation.

Soon after the release of Java versions 1.0 and 1.1, it became apparent that Java needed to be more performant. Consequently, a just-in-time (JIT) compiler was introduced in Java 1.2. When

combined with the JVM, it provided dynamic compilation based on hot methods and loop-back branch counts. This new VM was called the Java HotSpot VM.

Understanding Java HotSpot VM and Its Compilation Strategies

The Java HotSpot VM plays a critical role in executing Java programs efficiently. It includes JIT compilation, tiered compilation, and adaptive optimization to improve the performance of Java applications.

The Evolution of the HotSpot Execution Engine

The HotSpot VM performs *mixed-mode* execution, which means that the VM starts in interpreted mode, with the bytecode being converted into native code based on a description table. The table has a template of native code corresponding to each bytecode instruction known as the *TemplateTable*; it is just a simple lookup table. The execution code is stored in a code cache (known as *CodeCache*). *CodeCache* stores native code and is also a useful cache for storing JIT-ted code.

> **NOTE** HotSpot VM also provides an interpreter that doesn't need a template, called the C++ interpreter. Some OpenJDK ports[1] choose this route to simplify porting of the VM to non-x86 platforms.

Performance-Critical Methods and Their Optimization

Performance engineering is a critical aspect of software development, and a key part of this process involves identifying and optimizing performance-critical methods. These methods are frequently executed or contain performance-sensitive code, and they stand to gain the most from JIT compilation. Optimizing performance-critical methods is not just about choosing appropriate data structures and algorithms; it also involves identifying and optimizing the methods based on their frequency of invocation, size and complexity, and available system resources.

Consider the following `BookProgress` class as an example:

```
import java.util.*;

public class BookProgress {
    private String title;
    private Map<String, Integer> chapterPages;
    private Map<String, Integer> chapterPagesWritten;

    public BookProgress(String title) {
        this.title = title;
```

[1] https://wiki.openjdk.org/pages/viewpage.action?pageId=13729802

```
        this.chapterPages = new HashMap<>();
        this.chapterPagesWritten = new HashMap<>();
    }

    public void addChapter(String chapter, int totalPages) {
        this.chapterPages.put(chapter, totalPages);
        this.chapterPagesWritten.put(chapter, 0);
    }

    public void updateProgress(String chapter, int pagesWritten) {
        this.chapterPagesWritten.put(chapter, pagesWritten);
    }

    public double getProgress(String chapter) {
        return ((double) chapterPagesWritten.get(chapter) / chapterPages.get(chapter)) * 100;
    }

    public double getTotalProgress() {
        int totalWritten = chapterPagesWritten.values().stream().mapToInt(Integer::intValue).sum();
        int total = chapterPages.values().stream().mapToInt(Integer::intValue).sum();
        return ((double) totalWritten / total) * 100;
    }
}

public class Main {
    public static void main(String[] args) {
        BookProgress book = new BookProgress("JVM Performance Engineering");
        String[] chapters = {
            "Performance Evolution",
            "Performance and Type System",
            "Monolithic to Modular",
            "Unified Logging System",
            "End-to-End Performance Optimization",
            "Advanced Memory Management",
            "Runtime Performance Optimization",
            "Accelerating Startup",
            "Harnessing Exotic Hardware"
        };
        for (String chapter : chapters) {
            book.addChapter(chapter, 100);
        }
        for (int i = 0; i < 50; i++) {
            for (String chapter : chapters) {
                int currentPagesWritten = book.chapterPagesWritten.get(chapter);
                if (currentPagesWritten < 100) {
                    book.updateProgress(chapter, currentPagesWritten + 2);
                    double progress = book.getProgress(chapter);
```

```
                    System.out.println("Progress for chapter " + chapter + ": " + progress + "%");
                }
            }
        }
        System.out.println("Total book progress: " + book.getTotalProgress() + "%");
    }
}
```

In this code, we've defined a `BookProgress` class to track the progress of writing a book, which is divided into chapters. Each chapter has a total number of pages and a current count of pages written. The class provides methods to add chapters, update progress, and calculate the progress of each chapter and the overall book.

The `Main` class creates a `BookProgress` object for a book titled "JVM Performance Engineering." It adds nine chapters, each with 100 pages, and simulates writing the book by updating the progress of each chapter in a round-robin fashion, writing two pages at a time. After each update, it calculates and prints the progress of the current chapter and, once all pages are written, the overall progress of the book.

The `getProgress(String chapter)` and `updateProgress(String chapter, int pagesWritten)` methods are identified as performance-critical methods. Their frequent invocation makes them prime candidates for optimization by the HotSpot VM, illustrating how certain methods in a program may require more attention for performance optimization due to their high frequency of use.

Interpreter and JIT Compilation

The HotSpot VM provides an interpreter that converts bytecode into native code based on the *TemplateTable*. Interpretation is the first step in adaptive optimization offered by this VM and is considered the slowest form of bytecode execution. To make the execution faster, the HotSpot VM utilizes adaptive JIT compilation. The JIT-optimized code replaces the template code for methods that are identified as performance critical.

As mentioned in the previous section, the HotSpot VM monitors executed code for performance-critical methods based on two key metrics—method entry counts and loop-back branch counts. The VM assigns call counters to individual methods in the Java application. When the entry count exceeds a preestablished value, the method or its callee is chosen for asynchronous JIT compilation. Similarly, there is a counter for each loop in the code. Once the HotSpot VM determines that the loop-back branches (also known as loop-back edges) have crossed their threshold, the JIT optimizes that particular loop. This optimization is called on-stack replacement (OSR). With OSR, only the loop for which the loop-back branch counter overflowed will be compiled and replaced asynchronously on the execution stack.

Print Compilation

A very handy command-line option that can help us better understand adaptive optimization in the HotSpot VM is `-XX:+PrintCompilation`. This option also returns information on different optimized compilation levels, which are provided by an adaptive optimization called tiered compilation (discussed in the next subsection).

The output of the `-XX:+PrintCompilation` option is a log of the HotSpot VM's compilation tasks. Each line of the log represents a single compilation task and includes several pieces of information:

- The timestamp in milliseconds since the JVM started and this compilation task was logged.
- The unique identifier for this compilation task.
- Flags indicating certain properties of the method being compiled, such as whether it's an OSR method (%), whether it's synchronized (s), whether it has an exception handler (!), whether it's blocking (b), or whether it's native (n).
- The tiered compilation level, indicating the level of optimization applied to this method.
- The fully qualified name of the method being compiled.
- For OSR methods, the bytecode index where the compilation started. This is usually the start of a loop.
- The size of the method in the bytecode, in bytes.

Here are a few examples of the output of the `-XX:+PrintCompilation` option:

```
567  693 % !   3    org.h2.command.dml.Insert::insertRows @ 76 (513 bytes)
656  797     n 0    java.lang.Object::clone (native)
779  835  s    4    java.lang.StringBuffer::append (13 bytes)
```

These logs provide valuable insights into the behavior of the HotSpot VM's adaptive optimization, helping us understand how our Java applications are optimized at runtime.

Tiered Compilation

Tiered compilation, which was introduced in Java 7, provides multiple levels of optimized compilations, ranging from T0 to T4:

1. **T0:** Interpreted code, devoid of compilation. This is where the code starts and then moves on to the T1, T2, or T3 level.
2. **T1–T3:** Client-compiled mode. T1 is the first step where the method invocation counters and loop-back branch counters are used. At T2, the client compiler includes profiling information, referred to as profile-guided optimization; it may be familiar to readers who are conversant in static compiler optimizations. At the T3 compilation level, completely profiled code can be generated.
3. **T4:** The highest level of optimization provided by the HotSpot VM's server compiler.

Prior to tiered compilation, the server compiler would employ the interpreter to collect such profiling information. With the introduction of tiered compilation, the code reaches client compilation levels faster, and now the profiling information is generated by client-compiled methods themselves, providing better start-up times.

NOTE Tiered compilation has been enabled by default since Java 8.

Client and Server Compilers

The HotSpot VM provides two flavors of compilers: the fast client compiler (also known as the C1 compiler) and the server compiler (also known as the C2 compiler).

1. **Client compiler (C1):** Aims for fast start-up times in a client setup. The JIT invocation thresholds are lower for a client compiler than for a server compiler. This compiler is designed to compile code quickly, providing a fast start-up time, but the code it generates is less optimized.
2. **Server compiler (C2):** Offers many more adaptive optimizations and better thresholds geared toward higher performance. The counters that determine when a method/loop needs to be compiled are still the same, but the invocation thresholds are different (much lower) for a client compiler than for a server compiler. The server compiler takes longer to compile methods but produces highly optimized code that is beneficial for long-running applications. Some of the optimizations performed by the C2 compiler include *inlining* (replacing method invocations with the method's body), loop unrolling (increasing the loop body size to decrease the overhead of loop checks and to potentially apply other optimizations such as loop vectorization), dead code elimination (removing code that does not affect the program results), and range-check elimination (removing checks for index out-of-bounds errors if it can be assured that the array index never crosses its bounds). These optimizations help to improve the execution speed of the code and reduce the overhead of certain operations.[2]

Segmented Code Cache

As we delve deeper into the intricacies of the HotSpot VM, it's important to revisit the concept of the code cache. Recall that the code cache is a storage area for native code generated by the JIT compiler or the interpreter. With the introduction of tiered compilation, the code cache also becomes a repository for profiling information gathered at different levels of tiered compilation. Interestingly, even the *TemplateTable*, which the interpreter uses to look up the native code sequence for each bytecode, is stored in the code cache.

The size of the code cache is fixed at start-up but can be modified on the command line by passing the desired maximum value to `-XX:ReservedCodeCacheSize`. Prior to Java 7, the default value for this size was 48 MB. Once the code cache was filled up, all compilation would cease. This posed a significant problem when tiered compilation was enabled, as the code cache would contain not only JIT-compiled code (represented as *nmethod* in the HotSpot VM) but also profiled code. The *nmethod* refers to the internal representation of a Java method that has been compiled into machine code by the JIT compiler. In contrast, the profiled code is the code that has been analyzed and optimized based on its runtime behavior. The code cache needs

[2] "What the JIT!? Anatomy of the OpenJDK HotSpot VM." infoq.com.

to manage both of these types of code, leading to increased complexity and potential performance issues.

To address these problems, the default value for `ReservedCodeCacheSize` was increased to 240 MB in JDK 7 update 40. Furthermore, when the code cache occupancy crosses a preset `CodeCacheMinimumFreeSpace` threshold, the JIT compilation halts and the JVM runs a *sweeper*. The *nmethod* sweeper reclaims space by evacuating older compilations. However, sweeping the entire code cache data structure can be time-consuming, especially when the code cache is large and nearly full.

Java 9 introduced a significant change to the code cache: It was segmented into different regions based on the type of code. This not only reduced the sweeping time but also minimized fragmentation of the long-lived code by shorter-lived code. Co-locating code of the same type also reduced hardware-level instruction cache misses.

The current implementation of the segmented code cache includes the following regions:

- **Non-method code heap region:** This region is reserved for VM internal data structures that are not related to Java methods. For example, the *TemplateTable*, which is a VM internal data structure, resides here. This region doesn't contain compiled Java methods.
- **Non-profiled *nmethod* code heap:** This region contains Java methods that have been compiled by the JIT compiler without profiling information. These methods are fully optimized and are expected to be long-lived, meaning they won't be recompiled frequently and may need to be reclaimed only infrequently by the sweeper.
- **Profiled *nmethod* code heap:** This region contains Java methods that have been compiled with profiling information. These methods are not as optimized as those in the non-profiled region. They are considered transient because they can be recompiled into more optimized versions and moved to the non-profiled region as more profiling information becomes available. They can also be reclaimed by the sweeper as often as needed.

Each of these regions has a fixed size that can be set by their respective command-line options:

Heap Region Type	Size Command-Line Option
Non-method code heap	`-XX:NonMethodCodeHeapSize`
Non-profiled *nmethod* code heap	`-XX:NonProfiledCodeHeapSize`
Profiled *nmethod* code heap	`-XX:ProfiledCodeHeapSize`

Going forward, the hope is that the segmented code caches can accommodate additional code regions for heterogeneous code such as ahead-of-time (AOT)–compiled code and code for hardware accelerators.[3] There's also the expectation that the fixed sizing thresholds can be upgraded to utilize adaptive resizing, thereby avoiding wastage of memory.

[3] JEP 197: *Segmented Code Cache.* https://openjdk.org/jeps/197.

Adaptive Optimization and Deoptimization

Adaptive optimization allows the HotSpot VM runtime to optimize the interpreted code into compiled code or insert an optimized loop on the stack (so we could have something like an "interpreted to compiled, and back to interpreted" code execution sequence). There is another major advantage of adaptive optimization, however—in deoptimization of code. That means the compiled code could go back to being interpreted, or a higher-optimized code sequence could be rolled back into a less-optimized sequence.

Dynamic deoptimization helps Java reclaim code that may no longer be relevant. A few example use cases are when checking interdependencies during dynamic class loading, when dealing with polymorphic call sites, and when reclaiming less-optimized code. Deoptimization will first make the code "not entrant" and eventually reclaim it after marking it as "zombie" code.[4]

Deoptimization Scenarios

Deoptimization can occur in several scenarios when working with Java applications. In this section, we'll explore two of these scenarios.

Class Loading and Unloading

Consider an application containing two classes, `Car` and `DriverLicense`. The `Car` class requires a `DriverLicense` to enable drive mode. The JIT compiler optimizes the interaction between these two classes. However, if a new version of the `DriverLicense` class is loaded due to changes in driving regulations, the previously compiled code may no longer be valid. This necessitates deoptimization to revert to the interpreted mode or a less-optimized state. This allows the application to employ the new version of the `DriverLicense` class.

Here's an example code snippet:

```
class Car {
    private DriverLicense driverLicense;

    public Car(DriverLicense driverLicense) {
        this.driverLicense = driverLicense;
    }

    public void enableDriveMode() {
        if (driverLicense.isAdult()) {
            System.out.println("Drive mode enabled!");
        } else if (driverLicense.isTeenDriver()) {
            if (driverLicense.isLearner()) {
                System.out.println("You cannot drive without a licensed adult's supervision.");
            } else {
                System.out.println("Drive mode enabled!");
            }
        } else {
```

[4] https://www.infoq.com/articles/OpenJDK-HotSpot-What-the-JIT/

```
            System.out.println("You don't have a valid driver's license.");
        }
    }
}

class DriverLicense {
    private boolean isTeenDriver;
    private boolean isAdult;
    private boolean isLearner;

    public DriverLicense(boolean isTeenDriver, boolean isAdult, boolean isLearner) {
        this.isTeenDriver = isTeenDriver;
        this.isAdult = isAdult;
        this.isLearner = isLearner;
    }

    public boolean isTeenDriver() {
        return isTeenDriver;
    }

    public boolean isAdult() {
        return isAdult;
    }

    public boolean isLearner() {
        return isLearner;
    }
}

public class Main {
    public static void main(String[] args) {
        DriverLicense driverLicense = new DriverLicense(false, true, false);
        Car myCar = new Car(driverLicense);
        myCar.enableDriveMode();
    }
}
```

In this example, the `Car` class requires a `DriverLicense` to enable drive mode. The driver's license can be for an adult, a teen driver with a learner's permit, or a teen driver with a full license. The `enableDriveMode()` method checks the driver's license using the `isAdult()`, `isTeenDriver()`, and `isLearner()` methods, and prints the appropriate message to the console.

If a new version of the `DriverLicense` class is loaded, the previously optimized code may no longer be valid, triggering deoptimization. This allows the application to use the new version of the `DriverLicense` class without any issues.

Polymorphic Call Sites

Deoptimization can also occur when working with polymorphic call sites, where the actual method to be invoked is determined at runtime. Let's look at an example using the `DriverLicense` class:

```
abstract class DriverLicense {
    public abstract void drive();
}

class AdultLicense extends DriverLicense {
    public void drive() {
        System.out.println("Thanks for driving responsibly as an adult");
    }
}

class TeenPermit extends DriverLicense {
    public void drive() {
        System.out.println("Thanks for learning to drive responsibly as a teen");
    }
}

class SeniorLicense extends DriverLicense {
    public void drive() {
        System.out.println("Thanks for being a valued senior citizen");
    }
}

public class Main {
    public static void main(String[] args) {
        DriverLicense license = new AdultLicense();
        license.drive(); // monomorphic call site

        // Changing the call site to bimorphic
        if (Math.random() < 0.5) {
            license = new AdultLicense();
        } else {
            license = new TeenPermit();
        }
        license.drive(); // bimorphic call site

        // Changing the call site to megamorphic
        for (int i = 0; i < 100; i++) {
            if (Math.random() < 0.33) {
                license = new AdultLicense();
```

```
            } else if (Math.random() < 0.66) {
                license = new TeenPermit();
            } else {
                license = new SeniorLicense();
            }
            license.drive(); // megamorphic call site
        }
    }
}
```

In this example, the abstract `DriverLicense` class has three subclasses: `AdultLicense`, `TeenPermit`, and `SeniorLicense`. The `drive()` method is overridden in each subclass with different implementations.

First, when we assign an `AdultLicense` object to a `DriverLicense` variable and call `drive()`, the HotSpot VM optimizes the call site to a monomorphic call site and caches the target method address in an *inline cache* (a structure to track the call site's type profile).

Next, we change the call site to a bimorphic call site by randomly assigning an `AdultLicense` or `TeenPermit` object to the `DriverLicense` variable and calling `drive()`. Because there are two possible types, the VM can no longer use the monomorphic dispatch mechanism, so it switches to the bimorphic dispatch mechanism. This change does not require deoptimization—and still provides a performance boost by reducing the number of virtual method dispatches needed at the call site.

Finally, we change the call site to a megamorphic call site by randomly assigning an `AdultLicense`, `TeenPermit`, or `SeniorLicense` object to the `DriverLicense` variable and calling `drive()` 100 times. As there are now three possible types, the VM cannot use the bimorphic dispatch mechanism and must switch to the megamorphic dispatch mechanism. This change also does not require deoptimization.

However, if we were to introduce a new subclass `InternationalLicense` and change the call site to include it, the VM could potentially deoptimize the call site and switch to a megamorphic or polymorphic call site to handle the new type. This change is necessary because the VM's type profiling information for the call site would be outdated, and the previously optimized code would no longer be valid.

Here's the code snippet for the new subclass and the updated call site:

```
class InternationalLicense extends DriverLicense {
    public void drive() {
        System.out.println("Thanks for driving responsibly as an international driver");
    }
}

// Updated call site
for (int i = 0; i < 100; i++) {
```

```
    if (Math.random() < 0.25) {
        license = new AdultLicense();
    } else if (Math.random() < 0.5) {
        license = new TeenPermit();
    } else if (Math.random() < 0.75) {
        license = new SeniorLicense();
    } else {
        license = new InternationalLicense();
    }
    license.drive(); // megamorphic call site with a new type
}
```

HotSpot Garbage Collector: Memory Management Unit

A crucial component of the HotSpot execution engine is its memory management unit, commonly known as the garbage collector (GC). HotSpot provides multiple garbage collection algorithms that cater to a trifecta of performance aspects: application responsiveness, throughput, and overall footprint. *Responsiveness* refers to the time taken to receive a response from the system after sending a stimulus. *Throughput* measures the number of operations that can be performed per second on a given system. *Footprint* can be defined in two ways: as optimizing the amount of data or objects that can fit into the available space and as removing redundant information to save space.

Generational Garbage Collection, Stop-the-World, and Concurrent Algorithms

OpenJDK offers a variety of generational GCs that utilize different strategies to manage memory, with the common goal of improving application performance. These collectors are designed based on the principle that "most objects die young," meaning that most newly allocated objects on the Java heap are short-lived. By taking advantage of this observation, generational GCs aim to optimize memory management and significantly reduce the negative impact of garbage collection on the performance of the application.

Heap collection in GC terms involves identifying live objects, reclaiming space occupied by garbage objects, and, in some cases, compacting the heap to reduce fragmentation. Fragmentation can occur in two ways: (1) internal fragmentation, where allocated memory blocks are larger than necessary, leaving wasted space within the blocks; and (2) external fragmentation, where memory is allocated and deallocated in such a way that free memory is divided into noncontiguous blocks. External fragmentation can lead to inefficient memory use and potential allocation failures. Compaction is a technique used by some GCs to combat external fragmentation; it involves moving objects in memory to consolidate free memory into a single contiguous block. However, compaction can be a costly operation in terms of CPU usage and can cause lengthy pause times if it's done as a stop-the-world operation.

The OpenJDK GCs employ several different GC algorithms:

- **Stop-the-world (STW) algorithms:** STW algorithms pause application threads for the entire duration of the garbage collection work. Serial, Parallel, (Mostly) Concurrent Mark and Sweep (CMS), and Garbage First (G1) GCs use STW algorithms in specific phases of their collection cycles. The STW approach can result in longer pause times when the heap fills up and runs out of allocation space, especially in nongenerational heaps, which treat the heap as a single continuous space without segregating it into generations.
- **Concurrent algorithms:** These algorithms aim to minimize pause times by performing most of their work concurrently with the application threads. CMS is an example of a collector using concurrent algorithms. However, because CMS does not perform compaction, fragmentation can become an issue over time. This can lead to longer pause times or even cause a fallback to a full GC using the Serial Old collector, which does include compaction.
- **Incremental compacting algorithms:** The G1 GC introduced incremental compaction to deal with the fragmentation issue found in CMS. G1 divides the heap into smaller regions and performs garbage collection on a subset of regions during a collection cycle. This approach helps maintain more predictable pause times while also handling compaction.
- **Thread-local handshakes:** Newer GCs like Shenandoah and ZGC leverage thread-local handshakes to minimize STW pauses. By employing this mechanism, they can perform certain GC operations on a per-thread basis, allowing application threads to continue running while the GC works. This approach helps to reduce the overall impact of garbage collection on application performance.
- **Ultra-low-pause-time collectors:** The Shenandoah and ZGC aim to have ultra-low pause times by performing concurrent marking, relocation, and compaction. Both minimize the STW pauses to a small fraction of the overall garbage collection work, offering consistent low latency for applications. While these GCs are not generational in the traditional sense, they do divide the heap into regions and collect different regions at different times. This approach builds upon the principles of incremental and "garbage first" collection. As of this writing, efforts are ongoing to further develop these newer collectors into generational ones, but they are included in this section due to their innovative strategies that enhance the principles of generational garbage collection.

Each collector has its advantages and trade-offs, allowing developers to choose the one that best suits their application requirements.

Young Collections and Weak Generational Hypothesis

In the realm of a generational heap, the majority of allocations take place in the *eden* space of the *young* generation. An allocating thread may encounter an allocation failure when this eden space is near its capacity, indicating that the GC must step in and reclaim space.

During the first *young* collection, the eden space undergoes a scavenging process in which live objects are identified and subsequently moved into the *to* survivor space. The survivor

space serves as a transitional area where surviving objects are copied, aged, and moved back and forth between the *from* and *to* spaces until they cross a tenuring threshold. Once an object crosses this threshold, it is promoted to the *old* generation. The underlying objective here is to promote only those objects that have proven their longevity, thereby creating a "Teenage Wasteland," as Charlie Hunt[5] would explain. 😉

The generational garbage collection is based on two main characteristics related to the weak-generational hypothesis:

1. **Most objects die young:** This means that we promote only long-lived objects. If the generational GC is efficient, we don't promote transients, nor do we promote medium-lived objects. This usually results in smaller long-lived data sets, keeping premature promotions, fragmentation, evacuation failures, and similar degenerative issues at bay.
2. **Maintenance of generations:** The generational algorithm has proven to be a great help to OpenJDK GCs, but it comes with a cost. Because the young-generation collector works separately and more often than the old-generation collector, it ends up moving live data. Therefore, generational GCs incur maintenance/bookkeeping overhead to ensure that they mark all reachable objects—a feat achieved through the use of "write barriers" that track cross-generational references.

Figure 1.1 depicts the three key concepts of generational GCs, providing a visual reinforcement of the information discussed here. The word cloud consists of the following phrases:

- **Objects die young:** Highlighting the idea that most objects are short-lived and only long-lived objects are promoted.
- **Small long-lived data sets:** Emphasizing the efficiency of the generational GC in not promoting transients or medium-lived objects, resulting in smaller long-lived data sets.
- **Maintenance barriers:** Highlighting the overhead and bookkeeping required by generational GCs to mark all reachable objects, achieved through the use of write barriers.

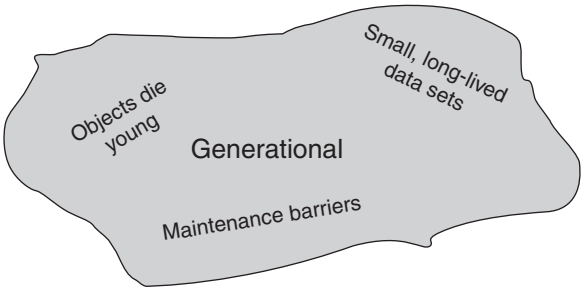

Figure 1.1 Key Concepts for Generational Garbage Collectors

[5] Charlie Hunt is my mentor, the author of *Java Performance* (https://ptgmedia.pearsoncmg.com/images/9780137142521/samplepages/0137142528.pdf), and my co-author for *Java Performance Companion* (www.pearson.com/en-us/subject-catalog/p/java-performance-companion/P200000009127/9780133796827).

Most HotSpot GCs employ the renowned "scavenge" algorithm for young collections. The Serial GC in HotSpot VM employs a single garbage collection thread dedicated to efficiently reclaiming memory within the young-generation space. In contrast, generational collectors such as the Parallel GC (throughput collector), G1 GC, and CMS GC leverage multiple GC worker threads.

Old-Generation Collection and Reclamation Triggers

Old-generation reclamation algorithms in HotSpot VM's generational GCs are optimized for throughput, responsiveness, or a combination of both. The Serial GC employs a single-threaded mark–sweep–compacting (MSC) GC. The Parallel GC uses a similar MSC GC with multiple threads. The CMS GC performs mostly concurrent marking, dividing the process into STW or concurrent phases. After marking, CMS reclaims old-generation space by performing in-place deallocation without compaction. If fragmentation occurs, CMS falls back to the serial MSC.

G1 GC, introduced in Java 7 update 4 and refined over time, is the first incremental collector. Specifically, it incrementally reclaims and compacts the old-generation space, as opposed to performing the single monolithic reclamation and compaction that is part of MSC. G1 GC divides the heap into smaller regions and performs garbage collection on a subset of regions during a collection cycle, which helps maintain more predictable pause times while also handling compaction.

After multiple young-generation collections, the old generation starts filling up, and garbage collection kicks in to reclaim space in the old generation. To do so, a full heap marking cycle must be triggered by either (1) a promotion failure, (2) a promotion of a regular-sized object that makes the old generation or the total heap cross the marking threshold, or (3) a large object allocation (also known as humongous allocation in the G1 GC) that causes the heap occupancy to cross a predetermined threshold.

Shenandoah GC and ZGC—introduced in JDK 12 and JDK 11, respectively—are ultra-low-pause-time collectors that aim to minimize STW pauses. In JDK 17, they are single-generational collectors. Apart from utilizing thread-local handshakes, these collectors know how to optimize for low-pause scenarios either by employing the application threads to help out or by asking the application threads to back off. This GC technique is known as graceful degradation.

Parallel GC Threads, Concurrent GC Threads, and Their Configuration

In the HotSpot VM, the total number of GC worker threads (also known as parallel GC threads) is calculated as a fraction of the total number of processing cores available to the Java process at start-up. Users can adjust the parallel GC thread count by assigning it directly on the command line using the `-XX:ParallelGCThreads=<n>` flag.

This configuration flag enables developers to define the number of parallel GC threads for GC algorithms that use parallel collection phases. It is particularly useful for tuning generational GCs, such as the Parallel GC and G1 GC. Recent additions like Shenandoah and ZGC, also use multiple GC worker threads and perform garbage collection concurrently with the application threads to minimize pause times. They benefit from load balancing, work sharing, and work stealing, which enhance performance and efficiency by parallelizing the garbage collection

process. This parallelization is particularly beneficial for applications running on multi-core processors, as it allows the GC to make better use of the available hardware resources.

In a similar vein, the `-XX:ConcGCThreads=<n>` configuration flag allows developers to specify the number of concurrent GC threads for specific GC algorithms that use concurrent collection phases. This flag is particularly useful for tuning GCs like G1, which performs concurrent work during marking, and Shenandoah and ZGC, which aim to minimize STW pauses by executing concurrent marking, relocation, and compaction.

By default, the number of parallel GC threads is automatically calculated based on the available CPU cores. Concurrent GC threads usually default to one-fourth of the parallel GC threads. However, developers may want to adjust the number of parallel or concurrent GC threads to better align with their application's performance requirements and available hardware resources.

Increasing the number of parallel GC threads can help improve overall GC throughput, as more threads work simultaneously on the parallel phases of this process. This increase may result in shorter GC pause times and potentially higher application throughput, but developers should be cautious not to over-commit processing resources.

By comparison, increasing the number of concurrent GC threads can enhance overall GC performance and expedite the GC cycle, as more threads work simultaneously on the concurrent phases of this process. However, this increase may come at the cost of higher CPU utilization and competition with application threads for CPU resources.

Conversely, reducing the number of parallel or concurrent GC threads may lower CPU utilization but could result in longer GC pause times, potentially affecting application performance and responsiveness. In some cases, if the concurrent collector is unable to keep up with the rate at which the application allocates objects (a situation referred to as the GC "losing the race"), it may lead to a graceful degradation—that is, the GC falls back to a less optimal but more reliable mode of operation, such as a STW collection mode, or might employ strategies like throttling the application's allocation rate to prevent it from overloading the collector.

Figure 1.2 shows the key concepts as a word cloud related to GC work:

- **Task queues:** Highlighting the mechanisms used by GCs to manage and distribute work among the GC threads.
- **Concurrent work:** Emphasizing the operations performed by the GC simultaneously with the application threads, aiming to minimize pause times.
- **Graceful degradation:** Referring to the GC's ability to switch to a less optimal but more reliable mode of operation when it can't keep up with the application's object allocation rate.
- **Pauses:** Highlighting the STW events during which application threads are halted to allow the GC to perform certain tasks.
- **Task stealing:** Emphasizing the strategy employed by some GCs in which idle GC threads "steal" tasks from the work queues of busier threads to ensure efficient load balancing.
- **Lots of threads:** Highlighting the use of multiple threads by GCs to parallelize the garbage collection process and improve throughput.

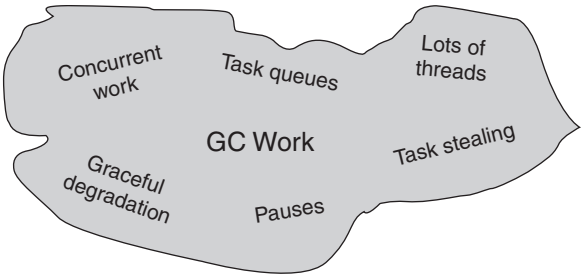

Figure 1.2 Key Concepts for Garbage Collection Work

It is crucial to find the right balance between the number of parallel and concurrent GC threads and application performance. Developers should conduct performance testing and monitoring to identify the optimal configuration for their specific use case. When tuning these threads, consider factors such as the number of available CPU cores, the nature of the application workload, and the desired balance between garbage collection throughput and application responsiveness.

The Evolution of the Java Programming Language and Its Ecosystem: A Closer Look

The Java language has evolved steadily since the early Java 1.0 days. To appreciate the advancements in the JVM, and particularly the HotSpot VM, it's crucial to understand the evolution of the Java programming language and its ecosystem. Gaining insight into how language features, libraries, frameworks, and tools have shaped and influenced the JVM's performance optimizations and garbage collection strategies will help us grasp the broader context.

Java 1.1 to Java 1.4.2 (J2SE 1.4.2)

Java 1.1, originally known as JDK 1.1, introduced JavaBeans, which allowed multiple objects to be encapsulated in a bean. This version also brought Java Database Connectivity (JDBC), Remote Method Invocation (RMI), and inner classes. These features set the stage for more complex applications, which in turn demanded improved JVM performance and garbage collection strategies.

From Java 1.2 to Java 5.0, the releases were renamed to include the version name, resulting in names like J2SE (Platform, Standard Edition). The renaming helped differentiate between Java 2 Micro Edition (J2ME) and Java 2 Enterprise Edition (J2EE).[6] J2SE 1.2 introduced two significant improvements to Java: the Collections Framework and the JIT compiler. The Collections Framework provided "a unified architecture for representing and manipulating collections",[7]

[6] www.oracle.com/java/technologies/javase/javanaming.html

[7] Collections Framework Overview. https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html.

which became essential for managing large-scale data structures and optimizing memory management in the JVM.

Java 1.3 (J2SE 1.3) added new APIs to the Collections Framework, introduced Math classes, and made the HotSpot VM the default Java VM. A directory services API was included for Java RMI to look up any directory or name service. These enhancements further influenced JVM efficiency by enabling more memory-efficient data management and interaction patterns.

The introduction of the New Input/Output (NIO) API in Java 1.4 (J2SE 1.4), based on Java Specification Request (JSR) #51,[8] significantly improved I/O operation efficiency. This enhancement resulted in reduced waiting times for I/O tasks and an overall boost in JVM performance. J2SE 1.4 also introduced the Logging API, which allowed for generating text or XML-formatted log messages that could be directed to a file or console.

J2SE 1.4.1 was soon superseded by J2SE 1.4.2, which included numerous performance enhancements in HotSpot's client and server compilers. Security enhancements were added as well, and Java users were introduced to Java updates via the Java Plug-in Control Panel Update tab. With the continuous improvements in the Java language and its ecosystem, JVM performance strategies evolved to accommodate increasingly more complex and resource-demanding applications.

Java 5 (J2SE 5.0)

The Java language made its first significant leap toward language refinement with the release of Java 5.0. This version introduced several key features, including generics, autoboxing/unboxing, annotations, and an enhanced `for` loop.

Language Features

Generics introduced two major changes: (1) a change in syntax and (2) modifications to the core API. Generics allow you to reuse your code for different data types, meaning you can write just a single class—there is no need to rewrite it for different inputs.

To compile the generics-enriched Java 5.0 code, you would need to use the Java compiler `javac`, which was packaged with the Java 5.0 JDK. (Any version prior to Java 5.0 did not have the core API changes.) The new Java compiler would produce errors if any type safety violations were detected at compile time. Hence, generics introduced type safety into Java. Also, generics eliminated the need for explicit casting, as casting became implicit.

Here's an example of how to create a generic class named `FreshmenAdmissions` in Java 5.0:

```
class FreshmenAdmissions<K, V> {
     //…
}
```

[8] https://jcp.org/en/jsr/detail?id=51

In this example, K and V are placeholders for the actual types of objects. The class `FreshmenAdmissions` is a generic type. If we declare an instance of this generic type without specifying the actual types for K and V, then it is considered to be a raw type of the generic type `FreshmenAdmissions<K, V>`. Raw types exist for generic types and are used when the specific type parameters are unknown.

```
FreshmenAdmissions applicationStatus;
```

However, suppose we declare the instance with actual types:

```
FreshmenAdmissions<String, Boolean>
```

Then `applicationStatus` is said to be a parameterized type—specifically, it is parameterized over types `String` and `Boolean`.

```
FreshmenAdmissions<String, Boolean> applicationStatus;
```

NOTE Many C++ developers may see the angle brackets < > and immediately draw parallels to C++ templates. Although both C++ and Java use generic types, C++ templates are more like a compile-time mechanism, where the generic type is replaced by the actual type by the C++ compiler, offering robust type safety.

While discussing generics, we should also touch on autoboxing and unboxing. In the `FreshmenAdmissions<K, V>` class, we can have a generic method that returns the type V:

```
public V getApprovalInfo() {
     return boolOrNumValue;
}
```

Based on our declaration of V in the parameterized type, we can perform a `Boolean` check and the code would compile correctly. For example:

```
applicationStatus = new FreshmenAdmissions<>();
if (applicationStatus.getApprovalInfo()) {
    //...
}
```

In this example, we see a generic type invocation of V as a `Boolean`. Autoboxing ensures that this code will compile correctly. In contrast, if we had done the generic type invocation of V as an `Integer`, we would get an "incompatible types" error. Thus, autoboxing is a Java compiler conversion that understands the relationship between a primitive and its object class. Just as autoboxing encapsulates a `boolean` value into its `Boolean` wrapper class, unboxing helps return a `boolean` value when the return type is a `Boolean`.

Here's the entire example (using Java 5.0):

```
class FreshmenAdmissions<K, V> {
    private K key;
    private V boolOrNumValue;

    public void admissionInformation(K name, V value) {
        key = name;
        boolOrNumValue = value;
    }

    public V getApprovalInfo() {
        return boolOrNumValue;
    }

    public K getApprovedName() {
        return key;
    }
}

public class GenericExample {

    public static void main(String[] args) {
        FreshmenAdmissions<String, Boolean> applicationStatus;
        applicationStatus = new FreshmenAdmissions<String, Boolean>();

        FreshmenAdmissions<String, Integer> applicantRollNumber;
        applicantRollNumber = new FreshmenAdmissions<String, Boolean>();

        applicationStatus.admissionInformation("Annika", true);

        if (applicationStatus.getApprovalInfo()) {
                applicantRollNumber.admissionInformation(applicationStatus.getApprovedName(), 4);
        }

        System.out.println("Applicant " + applicantRollNumber.getApprovedName() +
            " has been admitted with roll number of " + applicantRollNumber.getApprovalInfo());
    }

}
```

Figure 1.3 shows a class diagram to help visualize the relationship between the classes. In the diagram, the `FreshmenAdmissions` class has three methods: `admissionInformation(K,V)`, `getApprovalInfo():V`, and `getApprovedName():K`. The `GenericExample` class uses the `FreshmenAdmissions` class and has a `main(String[] args)` method.

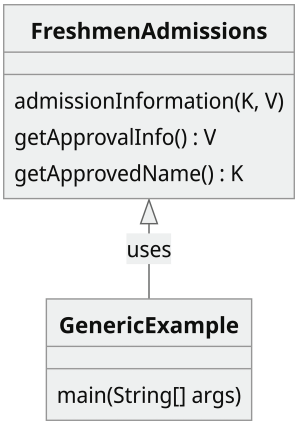

Figure 1.3 A Generic Type Example Showcasing the FreshmenAdmission Class

JVM and Packages Enhancements

Java 5.0 also brought significant additions to the `java.lang.*` and `java.util.*` packages and added most of the concurrent utilities as specified in JSR 166.[9]

Garbage collection *ergonomics* was a term coined in J2SE 5.0; it referred to a default collector for server-class machines.[10] The default collector was chosen to be the parallel GC, and default values for the initial and maximum heap sizes for parallel GC were set automatically.

> **NOTE** The Parallel GC of J2SE 5.0 days didn't have parallel compaction of the old generation. Thus, only the young generation could parallel scavenge; the old generation would still employ the serial GC algorithm, MSC.

Java 5.0 also introduced the enhanced `for` loop, often called the `for-each` loop, which greatly simplified iterating over arrays and collections. This new loop syntax automatically handled the iteration without the need for explicit index manipulation or calls to iterator methods. The enhanced `for` loop was both more concise and less error-prone than the traditional `for` loop, making it a valuable addition to the language. Here's an example to demonstrate its usage:

```
List<String> names = Arrays.asList("Monica", "Ben", "Annika", "Bodin");
for (String name : names) {
    System.out.println(name);
}
```

[9] Java Community Process. *JSRs: Java Specification Requests*, detail JSR# 166. https://jcp.org/en/jsr/detail?id=166.

[10] https://docs.oracle.com/javase/8/docs/technotes/guides/vm/server-class.html

In this example, the enhanced `for` loop iterates over the elements of the `names` list and assigns each element to the `name` variable in turn. This is much cleaner and more readable than using an index-based `for` loop or an iterator.

Java 5.0 brought several enhancements to the `java.lang` package, including the introduction of annotations, which allows for adding metadata to Java source code. Annotations provide a way to attach information to code elements such as classes, methods, and fields. They can be used by the Java compiler, runtime environment, or various development tools to generate additional code, enforce coding rules, or aid in debugging. The `java.lang.annotation` package contains the necessary classes and interfaces to define and process annotations. Some commonly used built-in annotations are `@Override`, `@Deprecated`, and `@SuppressWarnings`.

Another significant addition to the `java.lang` package was the `java.lang.instrument` package, which made it possible for Java agents to modify running programs on the JVM. The new services enabled developers to monitor and manage the execution of Java applications, providing insights into the behavior and performance of the code.

Java 6 (Java SE 6)

Java SE 6, also known as Mustang, primarily focused on additions and enhancements to the Java API,[11] but made only minor adjustments to the language itself. In particular, it introduced a few more interfaces to the Collections Framework and improved the JVM. For more information on Java 6, refer to the driver JSR 270.[12]

Parallel compaction, introduced in Java 6, allowed for the use of multiple GC worker threads to perform compaction of the old generation in the generational Java heap. This feature significantly improved the performance of Java applications by reducing the time taken for garbage collection. The performance implications of this feature are profound, as it allows for more efficient memory management, leading to improved scalability and responsiveness of Java applications.

JVM Enhancements

Java SE 6 made significant strides in improving scripting language support. The `javax.script` package was introduced, enabling developers to embed and execute scripting languages within Java applications. Here's an example of executing JavaScript code using the `ScriptEngine` API:

```
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class ScriptingExample {
    public static void main(String[] args) {
```

[11] Oracle. "Java SE 6 Features and Enhancements." www.oracle.com/java/technologies/javase/features.html.

[12] Java Community Process. *JSRs: Java Specification Requests*, detail JSR# 270. www.jcp.org/en/jsr/detail?id=270.

```
        FreshmenAdmissions<String, Integer> applicantGPA;
        applicantGPA = new FreshmenAdmissions<String,Integer>();
        applicantGPA.admissionInformation("Annika", 98);

        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("JavaScript");

        try {
            engine.put("score", applicantGPA.getApprovalInfo());
            engine.eval("var gpa = score / 25; print('Applicant GPA: ' + gpa);");
        } catch (ScriptException e) {
            e.printStackTrace();
        }
    }
}
```

Java SE 6 also introduced the Java Compiler API (JSR 199[13]), which allows developers to invoke the Java compiler programmatically. Using the `FreshmenAdmissions` example, we can compile our application's Java source file as follows:

```
import javax.tools.JavaCompiler;
import javax.tools.ToolProvider;
import java.io.File;

public class CompilerExample {
    public static void main(String[] args) {
        JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();

        int result = compiler.run(null, null, null, "path/to/FreshmenAdmissions.java");
        if (result == 0) {
            System.out.println("Congratulations! You have successfully compiled your program");
        } else {
            System.out.println("Apologies, your program failed to compile");
        }
    }
}
```

Figure 1.4 is a visual representation of the two classes and their relationships.

[13] https://jcp.org/en/jsr/detail?id=199

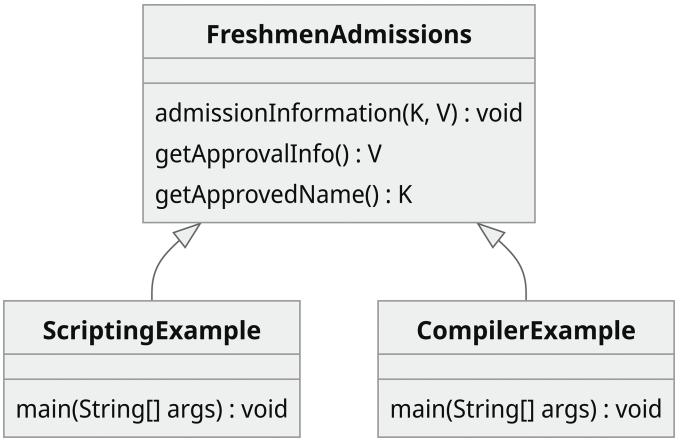

Figure 1.4 Script Engine API and Java Compiler API Examples

Additionally, Java SE 6 enhanced web services support by adding JAXB 2.0 and JAX-WS 2.0 APIs, simplifying the creation and consumption of SOAP-based web services.

Java 7 (Java SE 7)

Java SE 7 marked another significant milestone in Java's evolution, introducing numerous enhancements to both the Java language and the JVM. These enhancements are essential for understanding modern Java development.

Language Features

Java SE 7 introduced the diamond operator (<>), which simplified generic type inference. This operator enhances code readability and reduces redundancy. For instance, when declaring a `FreshmenAdmissions` instance, you no longer need to specify the type parameters twice:

```
// Before Java SE 7
FreshmenAdmissions<String, Boolean> applicationStatus = new FreshmenAdmissions<String, Boolean>();

// With Java SE 7
FreshmenAdmissions<String, Boolean> applicationStatus = new FreshmenAdmissions<>();
```

Another significant addition is the try-with-resources statement, which automates resource management, reducing the likelihood of resource leaks. Resources that implement `java.lang.AutoCloseable` can be used with this statement, and they are automatically closed when no longer needed.

Project Coin,[14] a part of Java SE 7, introduced several small but impactful enhancements, including improving literals, support for strings in switch statements, and enhanced error handling with try-with-resources and multi-catch blocks. Performance improvements such as compressed ordinary object pointers (oops), escape analysis, and tiered compilation also emerged during this time.

JVM Enhancements

Java SE 7 further enhanced the Java NIO library with NIO.2, providing better support for file I/O. Additionally, the fork/join framework, originating from JSR 166: *Concurrency Utilities*,[15] was incorporated into Java SE 7's concurrent utilities. This powerful framework enables efficient parallel processing of tasks by recursively breaking them down into smaller, more manageable subtasks and then merging the outcomes for a result.

The fork/join framework in Java reminds me of a programming competition I participated in during my student years. The event was held by the Institute of Electrical and Electronics Engineers (IEEE) and was a one-day challenge for students in computer science and engineering. My classmate and I decided to tackle the competition's two puzzles individually. I took on a puzzle that required optimizing a binary split-and-sort algorithm. After some focused work, I managed to significantly refine the algorithm, resulting in one of the most efficient solutions presented at the competition.

This experience was a practical demonstration of the power of binary splitting and sorting—a concept that is akin to the core principle of Java's fork/join framework. In the following fork/join example, you can see how the framework is a powerful tool to efficiently process tasks by dividing them into smaller subtasks and subsequently unifying them.

Here's an example of using the fork/join framework to calculate the average GPA of all admitted students:

```
// Create a list of admitted students with their GPAs
List<Integer> admittedStudentGPAs = Arrays.asList(95, 88, 76, 93, 84, 91);

// Calculate the average GPA using the fork/join framework
ForkJoinPool forkJoinPool = new ForkJoinPool();
AverageGPATask averageGPATask = new AverageGPATask(admittedStudentGPAs, 0,
                                                   admittedStudentGPAs.size());
double averageGPA = forkJoinPool.invoke(averageGPATask);

System.out.println("The average GPA of admitted students is: " + averageGPA);
```

In this example, we define a custom class `AverageGPATask` extending `RecursiveTask<Double>`. It's engineered to compute the average GPA from a subset of students by recursively splitting the list of GPAs into smaller tasks—a method reminiscent of the divide-and-conquer approach I refined during my competition days. Once the tasks are

[14] https://openjdk.org/projects/coin/

[15] https://jcp.org/en/jsr/detail?id=166

sufficiently small, they can be computed directly. The `compute()` method of `AverageGPATask` is responsible for both the division and the subsequent combination of these results to determine the overall average GPA.

Here's a simplified version of what the `AverageGPATask` class might look like:

```
class AverageGPATask extends RecursiveTask<Double> {
    private final List<Integer> studentGPAs;
    private final int start;
    private final int end;

    AverageGPATask(List<Integer> studentGPAs, int start, int end) {
        this.studentGPAs = studentGPAs;
        this.start = start;
        this.end = end;
    }

    @Override
    protected Double compute() {
        // Split the task if it's too large, or compute directly if it's small enough
        // Combine the results of subtasks
    }
}
```

In this code, we create an instance of `AverageGPATask` to calculate the average GPA of all admitted students. We then use a `ForkJoinPool` to execute this task. The `invoke()` method of `ForkJoinPool` initiates the task and actively waits for its completion, akin to calling a `sort()` function and waiting for a sorted array, thereby returning the computed result. This is a great example of how the fork/join framework can be used to enhance the efficiency of a Java application.

Figure 1.5 shows a visual representation of the classes and their relationships. In this diagram, there are two classes: `FreshmenAdmissions` and `AverageGPATask`. The `FreshmenAdmissions` class has methods for setting admission information, getting approval information, and getting the approved name. The `AverageGPATask` class, which calculates the average GPA, has a `compute` method that returns a `Double`. The `ForkJoinPool` class uses the `AverageGPATask` class to calculate the average GPA; the `FreshmenAdmissions` class uses the `AverageGPATask` to calculate the average GPA of admitted students.

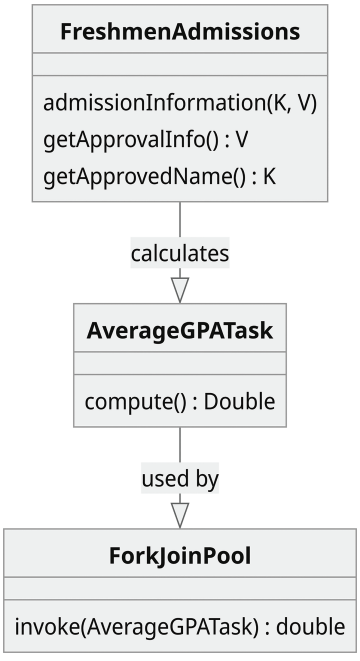

Figure 1.5 An Elaborate ForkJoinPool Example

Java SE 7 also brought significant enhancements to garbage collection. The parallel GC became NUMA-aware, optimizing performance on systems with the Non-Uniform Memory Access (NUMA) architecture. At its core, NUMA enhances memory access efficiency by taking into account the physical proximity of memory to a processing unit, such as a CPU or core. Prior to the introduction of NUMA, computers exhibited uniform memory access patterns, with all processing units sharing a single memory resource equally. NUMA introduced a paradigm in which processors access their local memory—memory that is physically closer to them—more swiftly than distant, non-local memory, which might be adjacent to other processors or shared among various cores. Understanding the intricacies of NUMA-aware garbage collection is vital for optimizing Java applications running on NUMA systems. (For an in-depth exploration of the NUMA-awareness in GC, see Chapter 6, "Advanced Memory Management and Garbage Collection in OpenJDK.")

In a typical NUMA setup, the memory access times can vary because data must traverse certain paths—or *hops*—within the system. These *hops* refer to the transfer of data between different nodes. Local memory access is direct and thus faster (fewer *hops*), while accessing non-local memory—residing farther from the processor—incurs additional *hops* resulting in increased latency due to the longer traversal paths. This architecture is particularly advantageous in

systems with multiple processors, as it maximizes the efficiency of memory utilization by leveraging the proximity factor.

NOTE My involvement with the implementation of NUMA-aware garbage collection first came during my tenure at AMD. I was involved in the performance characterization of AMD's Opteron processor,[16] a groundbreaking product that bought NUMA architecture into the mainstream. The Opteron[17] was designed with an integrated memory controller and a HyperTransport interconnect,[18] which allowed for direct, high-speed connections both between processors and between the processor and the memory. This design made the Opteron ideal for NUMA systems, and it was used in many high-performance servers, including Sun Microsystems' V40z.[19] The V40z was a high-performance server that was built by Sun Microsystems using AMD's Opteron processors. It was a NUMA system, so it could significantly benefit from NUMA-aware garbage collection. I was instrumental in providing the JDK team with data on cross-traffic and multi-JVM-local traffic, as well as insights into interleaved memory to illustrate how to amortize the latency of the *hops* between memory nodes.

To further substantiate the benefits of this feature, I implemented a prototype for the HotSpot VM with `-XX:+UseLargePages`. This prototype utilized large pages (also known as HugeTLB pages[20]) on Linux and the `numactl` API,[21] a tool for controlling NUMA policy on Linux systems. It served as a tangible demonstration of the potential performance improvements that could be achieved with NUMA-aware GC.

When the Parallel GC is NUMA-aware, it optimizes the way it handles memory to improve performance on NUMA systems. The young generation's eden space is allocated in NUMA-aware regions. In other words, when an object is created, it's placed in an eden space local to the processor that's executing the thread that created the object. This can significantly improve performance because accessing local memory is faster than accessing non-local memory.

The survivor spaces and the old generation are page interleaved in memory. Page interleaving is a technique used to distribute memory pages across different nodes in a NUMA system. It can help balance memory utilization across the nodes, leading to more efficient utilization of the memory bandwidth available on each node.

Note, however, that these optimizations are most beneficial on systems with a large number of processors and a significant disparity between local and non-local memory access times. On systems with a small number of processors or near-similar local and non-local memory access times, the impact of NUMA-aware garbage collection might be less noticeable. Also,

[16] Within AMD's Server Perf and Java Labs, my focus was on refining the HotSpot VM to enhance performance for Java workloads on NUMA-aware architectures, concentrating on JIT compiler optimizations, code generation, and GC efficiency for Opteron processor family.

[17] The Opteron was the first processor that supported AMD64, also known as x86-64ISA: https://en.wikipedia.org/wiki/Opteron.

[18] https://en.wikipedia.org/wiki/HyperTransport

[19] Oracle. "Introduction to the Sun Fire V20z and Sun Fire V40z Servers." https://docs.oracle.com/cd/E19121-01/sf.v40z/817-5248-21/chapter1.html.

[20] "HugeTLB Pages." www.kernel.org/doc/html/latest/admin-guide/mm/hugetlbpage.html.

[21] https://halobates.de/numaapi3.pdf

NUMA-aware garbage collection is not enabled by default; instead, it is enabled using the `-XX:+UseNUMA` JVM option.

In addition to the NUMA-aware GC, Java SE 7 Update 4 introduced the Garbage First Garbage Collector (G1 GC). G1 GC was designed to offer superior performance and predictability compared to its predecessors. Chapter 6 of this book delves into the enhancements brought by G1 GC, providing detailed examples and practical tips for its usage. For a comprehensive understanding of G1 GC, I recommend the *Java Performance Companion* book.[22]

Java 8 (Java SE 8)

Java 8 brought several significant features to the table that enhanced the Java programming language and its capabilities. These features can be leveraged to improve the efficiency and functionality of Java applications.

Language Features

One of the most notable features introduced in Java 8 was lambda expressions, which allow for more concise and functional-style programming. This feature can improve the simplicity of code by enabling more efficient implementation of functional programming concepts. For instance, if we have a list of students and we want to filter out those with a GPA greater than 80, we could use a lambda expression to do so concisely.

```
List<Student> admittedStudentsGPA = Arrays.asList(student1, student2, student3, student4,
student5, student6);
List<Student> filteredStudentsList = admittedStudentsGPA.stream().filter(s -> s.getGPA() > 80).
collect(Collectors.toList());
```

The lambda expression `s -> s.getGPA() > 80` is essentially a short function that takes a `Student` object `s` and returns a `boolean` value based on the condition `s.getGPA() > 80`. The `filter` method uses this lambda expression to filter the stream of students and `collect` gathers the results back into a list.

Java 8 also extended annotations to cover any place where a type is used, a capability known as *type annotations*. This helped enhance the language's expressiveness and flexibility. For example, we could use a type annotation to specify that the `studentGPAs` list should contain only non-null `Integer` objects:

```
private final List<@NonNull Integer> studentGPAs;
```

Additionally, Java 8 introduced the Stream API, which allows for efficient data manipulation and processing on collections. This addition contributed to JVM performance improvements, as developers could now optimize data processing more effectively. Here's an example using the Stream API to calculate the average GPA of the students in the `admittedStudentsGPA` list:

```
double averageGPA = admittedStudentsGPA.stream().mapToInt(Student::getGPA).average().orElse(0);
```

[22] www.pearson.com/en-us/subject-catalog/p/java-performance-companion/P200000009127/9780133796827

In this example, the `stream` method is called to create a stream from the list, `mapToInt` is used with a method reference `Student::getGPA` to convert each `Student` object to its GPA (an integer), `average` calculates the average of these integers, and `orElse` provides a default value of 0 if the stream is empty and an average can't be calculated.

Another enhancement in Java 8 was the introduction of default methods in interfaces, which enable developers to add new methods without breaking existing implementations, thus increasing the flexibility of interface design. If we were to define an interface for our tasks, we could use default methods to provide a standard implementation of certain methods. This could be useful if most tasks need to perform a set of common operations. For instance, we could define a `ComputableTask` interface with a `logStart` default method that logs when a task starts:

```
public interface Task {
    default void logStart() {
        System.out.println("Task started");
    }
    Double compute();
}
```

This `ComputableTask` interface could be implemented by any class that represents a task, providing a standard way to log the start of a task and ensuring that each task can be computed. For example, our `AverageGPATask` class could implement the `ComputableTask` interface:

```
class AverageGPATask extends RecursiveTask<Double> implements ComputableTask {
    // ...
}
```

JVM Enhancements

On the JVM front, Java 8 removed the Permanent Generation (PermGen) memory space, which was previously used to store metadata about loaded classes and other objects not associated with any instance. PermGen often caused problems due to memory leaks and the need for manual tuning. In Java 8, it was replaced by the Metaspace, which is located in native memory, rather than on the Java heap. This change eliminated many PermGen-related issues and improved the overall performance of the JVM. The Metaspace is discussed in more detail in Chapter 8, "Accelerating Time to Steady State with OpenJDK HotSpot VM."

Another valuable optimization introduced in JDK 8 update 20 was String Deduplication, a feature was specifically designed for the G1 GC. In Java, `String` objects are immutable, which means that once created, their content cannot be changed. This characteristic often leads to multiple `String` objects containing the same character sequences. While these duplicates don't impact the correctness of the application, they do contribute to increased memory footprint and can indirectly affect performance due to higher garbage collection overhead.

String Deduplication takes care of the issue of duplicate `String` objects by scanning the Java heap during GC cycles. It identifies duplicate strings and replaces them with references to a

single canonical instance. By decreasing the overall heap size and the number of live objects, the time taken for GC pauses can be reduced, contributing to lower tail latencies (this feature is discussed in more detail in Chapter 6). To enable the String Deduplication feature with the G1 GC, you can add the following JVM options:

```
-XX:+UseG1GC -XX:+UseStringDeduplication
```

Java 9 (Java SE 9) to Java 16 (Java SE 16)

Java 9: Project Jigsaw, JShell, AArch64 Port, and Improved Contended Locking

Java 9 brought several significant enhancements to the Java platform. The most notable addition was Project Jigsaw,[23] which implemented a module system, enhancing the platform's scalability and maintainability. We will take a deep dive into modularity in Chapter 3, "From Monolithic to Modular Java: A Retrospective and Ongoing Evolution."

Java 9 also introduced JShell, an interactive Java REPL (read–evaluate–print loop), enabling developers to quickly test and experiment with Java code. JShell allows for the evaluation of code snippets, including statements, expressions, and definitions. It also supports commands, which can be entered by adding a forward slash ("/") to the beginning of the command. For example, to change the interaction mode to verbose in JShell, you would enter the command `/set feedback verbose`.

Here's an example of how you might use JShell:

```
$ jshell
|  Welcome to JShell -- Version 17.0.7
|  For an introduction type: /help intro

jshell> /set feedback verbose
|  Feedback mode: verbose

jshell> boolean trueMorn = false
trueMorn ==> false
|  Created variable trueMorn : boolean

jshell> boolean Day(char c) {
   ...> return (c == 'Y');
   ...> }
|  Created method Day(char)

jshell> System.out.println("Did you wake up before 9 AM? (Y/N)")
Did you wake up before 9 AM? (Y/N)
```

[23] https://openjdk.org/projects/jigsaw/

```
jshell> trueMorn = Day((char) System.in.read())
Y
trueMorn ==> true
|  Assigned to trueMorn : boolean

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

In this example, we first initialize `trueMorn` as `false` and then define our `Day()` method, which returns `true` or `false` based on the value of `c`. We then use `System.out.println` to ask our question and read the char input. We provide `Y` as input, so `trueMorn` is evaluated to be `true`.

JShell also provides commands like `/list`, `/vars`, `/methods`, and `/imports`, which provide useful information about the current JShell session. For instance, `/list` shows all the code snippets you've entered:

```
jshell> /list

   1 : boolean trueMorn = false;
   2 : boolean Day(char c) {
       return (c=='Y');
       }
   3 : System.out.println("Did you wake up before 9 AM? (Y/N)")
   4 : trueMorn = Day((char) System.in.read())
   5 : System.out.println("It is " + trueMorn + " that you are a morning person")
```

`/vars` lists all the variables you've declared:

```
jshell> /vars
|    boolean trueMorn = true
```

`/methods` lists all the methods you've defined:

```
jshell> /methods
|    boolean Day(char)
```

`/imports` shows all the imports in your current session:

```
jshell> /imports
|    import java.io.*
|    import java.math.*
|    import java.net.*
|    import java.nio.file.*
|    import java.util.*
|    import java.util.concurrent.*
```

```
|    import java.util.function.*
|    import java.util.prefs.*
|    import java.util.regex.*
|    import java.util.stream.*
```

The imports will also show up if you run the `/list –all` or `/list -start` command.

Furthermore, JShell can work with modules using the `--module-path` and `--add-modules` options, allowing you to explore and experiment with modular Java code.

Java 9 also added the AArch64 (also known as Arm64) port, thereby providing official support for the Arm 64-bit architecture in OpenJDK. This port expanded Java's reach in the ecosystem, allowing it to run on a wider variety of devices, including those with Arm processors.

Additionally, Java 9 improved contended locking through JEP 143: *Improve Contended Locking*.[24] This enhancement optimized the performance of intrinsic object locks and reduced the overhead associated with contended locking, improving the performance of Java applications that heavily rely on synchronization and contended locking. In Chapter 7, "Runtime Performance Optimizations: A Focus on Strings, Locks, and Beyond" we will take a closer look at locks and strings performance.

Release Cadence Change and Continuous Improvements

Starting from Java 9, the OpenJDK community decided to change the release cadence to a more predictable six-month cycle. This change was intended to provide more frequent updates and improvements to the language and its ecosystem. Consequently, developers could benefit from new features and enhancements more quickly.

Java 10: Local Variable Type Inference and G1 Parallel Full GC

Java 10 introduced local-variable type inference with the `var` keyword. This feature simplifies code by allowing the Java compiler to infer the variable's type from its initializer. This results in less verbose code, making it more readable and reducing boilerplate. Here's an example:

```
// Before Java 10
List<Student> admittedStudentsGPA = new ArrayList<>();

// With 'var' in Java 10
var admittedStudentsGPA = new ArrayList<Student>();
```

In this example, the `var` keyword is used to declare the variable `admittedStudentsGPA`. The compiler infers the type of `admittedStudentsGPA` from its initializer `new ArrayList <Student>()`, so you don't need to explicitly declare it.

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

```
// Using 'var' for local-variable type inference with admittedStudentsGPA list
var filteredStudentsList = admittedStudentsGPA.stream().filter(s -> s.getGPA() > 80).
collect(Collectors.toList());
```

In the preceding example, the `var` keyword is used in a more complex scenario involving a stream operation. The compiler infers the type of `filteredStudentsList` from the result of the stream operation.

In addition to addressing local-variable type inference, Java 10 improved the G1 GC by enabling parallel full garbage collections. This enhancement increased the efficiency of garbage collection, especially for applications with large heaps and those that suffer from some pathological (corner) case that causes evacuation failures, which in turn evoke the fallback full GC.

Java 11: New HTTP Client and String Methods, and Epsilon and Z GC

Java 11, a long-term support (LTS) release, introduced a variety of improvements to the Java platform, enhancing its performance and utility. One significant JVM addition in this release was the Epsilon GC,[25] an experimental no-op GC designed to test the performance of applications with minimal GC interference. The Epsilon GC allows developers to understand the impact of garbage collection on their application's performance, helping them make informed decisions about GC tuning and optimization.

```
$ java -XX:+UnlockExperimentalVMOptions -XX:+UseEpsilonGC -jar myApplication.jar
```

Epsilon GC is a unique addition to the JVM. It manages memory allocation, but doesn't implement any actual memory reclamation process. With this GC, once the available Java heap is used up, the JVM will shut down. Although this behavior might seem unusual, it's actually quite useful in certain scenarios—for instance, with extremely short-lived microservices. Moreover, for ultra-low-latency applications that can afford the heap footprint, Epsilon GC can help avoid all marking, moving, and compaction, thereby eliminating performance overhead. This streamlined behavior makes it an excellent tool for performance testing an application's memory pressures and understanding the overhead associated with VM/memory barriers.

Another noteworthy JVM enhancement in Java 11 was the introduction of the Z Garbage Collector (ZGC), which was initially available as an experimental feature. ZGC is a low-latency, scalable GC designed to handle large heaps with minimal pause times. By providing better garbage collection performance for applications with large memory requirements, ZGC enables developers to create high-performance applications that can effectively manage memory resources. (We'll take a deep dive into ZGC in Chapter 6 of this book.)

Keep in mind that these GCs are experimental in JDK 11, so using the `+UnlockExperimentalVMOptions` flag is necessary to enable them. Here's an example for ZGC:

```
$ java -XX:+UnlockExperimentalVMOptions -XX:+UseZGC -jar myApplication.jar
```

[25] https://blogs.oracle.com/javamagazine/post/epsilon-the-jdks-do-nothing-garbage-collector

In addition to JVM enhancements, Java 11 introduced a new HTTP client API that supports HTTP/2 and WebSocket. This API improved performance and provided a modern alternative to the `HttpURLConnection` API. As an example of its use, assume that we want to fetch some extracurricular activities' data related to our students from a remote API:

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("http://students-extracurricular.com/students"))
    .build();

client.sendAsync(request, HttpResponse.BodyHandlers.ofString())
    .thenApply(HttpResponse::body)
    .thenAccept(System.out::println);
```

In this example, we're making a GET request to `http://students-extracurricular.com/students` to fetch the extracurricular data. The response is handled asynchronously, and when the data is received, it's printed to the console. This new API provides a more modern and efficient way to make HTTP requests compared to the older `HttpURLConnection` API.

Furthermore, Java 11 added several new `String` methods, such as `strip()`, `repeat()`, and `isBlank()`. Let's look at our student GPA example and see how we can enhance it using these methods:

```
// Here we have student names with leading and trailing whitespaces
List<String> studentNames = Arrays.asList("  Monica  ", "  Ben  ", "  Annika  ", "  Bodin  ");

// We can use the strip() method to clean up the names
List<String> cleanedNames = studentNames.stream()
    .map(String::strip)
    .collect(Collectors.toList());

// Now suppose we want to create a String that repeats each name three times
// We can use the repeat() method for this
List<String> repeatedNames = cleanedNames.stream()
    .map(name -> name.repeat(3))
    .collect(Collectors.toList());

// And suppose we want to check if any name is blank after stripping out whitespaces
// We can use the isBlank() method for this
boolean hasBlankName = cleanedNames.stream()
    .anyMatch(String::isBlank);
```

These enhancements, along with numerous other improvements, made Java 11 a robust and efficient platform for developing high-performance applications.

Java 12: Switch Expressions and Shenandoah GC

Java 12 introduced the Shenandoah GC as an experimental feature. Like ZGC, Shenandoah GC is designed for large heaps with low latency requirements. This addition contributed to improved JVM performance for latency-sensitive applications. To enable Shenandoah GC, you can use the following JVM option:

```
$ java -XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC -jar myApplication.jar
```

Java 12 also introduced switch expressions as a preview feature, simplifying `switch` statements and making them more readable and concise:

```
String admissionResult = switch (status) {
    case "accepted" -> "Congratulations!";
    case "rejected" -> "Better luck next time.";
    default -> "Awaiting decision.";
};
```

In Java 14, you also have the option to use the `yield` keyword to return a value from a switch block. This is particularly useful when you have multiple statements in a case block. Here's an example:

```
String admissionResult = switch (status) {
    case "accepted" -> {
        // You can have multiple statements here
        yield "Congratulations!";
    }
    case "rejected" -> {
        // You can have multiple statements here
        yield "Better luck next time.";
    }
    default -> {
        // You can have multiple statements here
        yield "Awaiting decision.";
    }
};
```

Java 13: Text Blocks and ZGC Enhancements

Java 13 offered a number of new features and enhancements, including the introduction of text blocks as a preview feature. Text blocks simplify the creation of multiline string literals, making it easier to write and read formatted text:

```
String studentInfo = """
                     Name: Monica Beckwith
                     GPA: 3.83
                     Status: Accepted
                     """;
```

JDK 13 also included several enhancements to ZGC. For example, the new ZGC uncommit feature allowed unused heap memory to be returned to the operating system (OS) in a more timely and efficient manner. Prior to this feature, ZGC would release memory back to the OS only during a full GC cycle,[26] which could be infrequent and might result in large amounts of memory being held by the JVM unnecessarily. With the new feature, ZGC could return memory back to the operating system immediately upon detecting that a page is no longer needed by the JVM.

Java 14: Pattern Matching for instanceof and NUMA-Aware Memory Allocator for G1

Java 14 introduced pattern matching for the `instanceof` operator as a preview feature, simplifying type checking and casting:

```
if (object instanceof Student student) {
    System.out.println("Student name: " + student.getName());
}
```

Java 14 also introduced NUMA-aware memory allocation for the G1 GC. This feature is especially useful in multi-socket systems, where memory access latency and bandwidth efficiency can vary significantly across the sockets. With NUMA-aware allocation, G1 can allocate memory with a preference for the local NUMA node, which can significantly reduce memory access latency and improve performance. This feature can be enabled on supported platforms using the command-line option `-XX:+UseNUMA`.

Java 15: Sealed and Hidden Classes

Java 15 introduced two significant features that give developers more control over class hierarchies and the encapsulation of implementation details.

Sealed classes, introduced as a preview feature,[27] enable developers to define a limited set of subclasses for a superclass, providing more control over class hierarchies. For example, in a university application system, you could have:

```
public sealed class Student permits UndergraduateStudent, GraduateStudent, ExchangeStudent {
    private final String name;
    private final double gpa;
    // Common student properties and methods
}

final class UndergraduateStudent extends Student {
    // Undergraduate-specific properties and methods
}
```

[26] https://openjdk.org/jeps/351

[27] https://openjdk.org/jeps/360

```
final class GraduateStudent extends Student {
    // Graduate-specific properties and methods
}

final class ExchangeStudent extends Student {
    // Exchange-specific properties and methods
}
```

Here, `Student` is a sealed class, and only the specified classes (`UndergraduateStudent`, `GraduateStudent`, `ExchangeStudent`) can extend it, ensuring a controlled class hierarchy.

Hidden classes,[28] in contrast, are classes that are not discoverable by their name at runtime and can be used by frameworks that generate classes dynamically. For instance, a hidden class might be generated for a specific task like a query handler in a database framework:

```
// Within a database framework method
MethodHandles.Lookup lookup = MethodHandles.lookup();
Class<?> queryHandlerClass = lookup.defineHiddenClass(
    queryHandlerBytecode, true).lookupClass();
// The queryHandlerClass is now a hidden class, not directly usable by other classes.
```

The `defineHiddenClass` method generates a class from bytecode. Since it's not referenced elsewhere, the class remains internal and inaccessible to external classes.[29]

Both sealed and hidden classes enhance the Java language by providing tools for more precise control over class design, improving maintainability, and securing class implementation details.

Java 16: Records, ZGC Concurrent Thread-Stack Processing, and Port to Windows on Arm

Java 16 introduced records, a new class type that streamlines the creation of data-centric classes. Records provide a succinct syntax for defining immutable data structures:

```
record Student(String name, double gpa) {}
```

Leading my team at Microsoft, we contributed to the Java ecosystem by enabling the JDK to function on Windows running on Arm 64 hardware. This significant undertaking was encapsulated in JEP 388: *Windows/AArch64 Port*,[30] where we successfully enabled the

[28] https://openjdk.org/jeps/371

[29] Although the hidden class example omits `MethodHandles.Lookup.ClassOption.NESTMATE` for simplicity, it's worth mentioning for a fuller understanding. `NESTMATE` enables a hidden class to access private members (like methods and fields) of its defining class, which is useful in advanced Java development. This specialized feature is not typically needed for everyday Java use but can be important for complex internal implementations.

[30] https://openjdk.org/jeps/388

template interpreter, the C1 and C2 JIT compilers, and all supported GCs. Our work broadened the environments in which Java can operate, further solidifying its position in the technology ecosystem.

In addition, ZGC in Java 15 evolved into a full supported feature. In Java 16, with JEP 376: *ZGC: Concurrent Thread-Stack Processing*,[31] ZGC underwent further enhancement. This improvement involves moving thread-stack processing to the concurrent phase, significantly reducing GC pause times—bolstering performance for applications managing large data sets.

Java 17 (Java SE 17)

Java 17, the most recent LTS release as of this book's writing, brings a variety of enhancements to the JVM, the Java language, and the Java libraries. These improvements significantly boost the efficiency, performance, and security of Java applications.

JVM Enhancements: A Leap Forward in Performance and Efficiency

Java 17 introduced several notable JVM improvements. JEP 356: *Enhanced Pseudo-Random Number Generators*[32] provides new interfaces and implementations for random number generation, offering a more flexible and efficient approach to generating random numbers in Java applications.

Another significant enhancement is JEP 382: *New macOS Rendering Pipeline*,[33] which improves the efficiency of Java two-dimensional graphics rendering on macOS. This new rendering pipeline leverages the native graphics capabilities of macOS, resulting in better performance and rendering quality for Java applications running on macOS.

Furthermore, JEP 391: *macOS/AArch64 Port*[34] adds a macOS/AArch64 port to the JDK. This port, supported by our Windows/AArch64 port, enables Java to run natively on macOS devices with M1-based architecture, further expanding the platform support for the JVM and enhancing performance on these architectures.

Security and Encapsulation Enhancements: Fortifying the JDK

Java 17 also sought to increase the security of the JDK. JEP 403: *Strongly Encapsulate JDK Internals*[35] makes it more challenging to access internal APIs that are not intended for general use. This change fortifies the security of the JDK and ensures compatibility with future releases by discouraging the use of internal APIs that may change or be removed in subsequent versions.

Another key security enhancement in Java 17 is the introduction of JEP 415: *Context-Specific Deserialization Filters*.[36] This feature provides a mechanism for defining deserialization filters based on context, offering a more granular control over the deserialization process. It addresses

[31] https://openjdk.org/jeps/376
[32] https://openjdk.org/jeps/356
[33] https://openjdk.org/jeps/382
[34] https://openjdk.org/jeps/391
[35] https://openjdk.org/jeps/403
[36] https://openjdk.org/jeps/415

some of the security concerns associated with Java's serialization mechanism, making it safer to use. As a result of this enhancement, application developers can now construct and apply filters to every deserialization operation, providing a more dynamic and context-specific approach compared to the static JVM-wide filter introduced in Java 9.

Language and Library Enhancements

Java 17 introduced several language and library enhancements geared toward improving developer productivity and the overall efficiency of Java applications. One key feature was JEP 406: *Pattern Matching for switch*,[37] which was introduced as a preview feature. This feature simplifies coding by allowing common patterns in `switch` statements and expressions to be expressed more concisely and safely. It enhances the readability of the code and reduces the chances of errors.

Suppose we're using a switch expression to generate a custom message for a student based on their status. The `String.format()` function is used to insert the student's name and GPA into the message:

```
record Student(String name, String status, double gpa) {}

// Let's assume we have a student
Student student = new Student("Monica", "accepted", 3.83);

String admissionResult = switch (student.status()) {
  case "accepted" -> String.format("%s has been accepted with a GPA of %.2f. Congratulations!",
student.name(), student.gpa());
  case "rejected" -> String.format("%s has been rejected despite a GPA of %.2f. Better luck next
time.", student.name(), student.gpa());
  case "waitlisted" -> String.format("%s is waitlisted. Current GPA is %.2f. Keep your fingers
crossed!", student.name(), student.gpa());
  case "pending" -> String.format("%s's application is still pending. Current GPA is %.2f. We hope
for the best!", student.name(), student.gpa());
  default -> String.format("The status of %s is unknown. Please check the application.", student.
name());
};

System.out.println(admissionResult);
```

Moreover, Java 17 includes JEP 412: *Foreign Function and Memory API (Incubator)*,[38] which provides a pure Java API for calling native code and working with native memory. This API is designed to be safer, more efficient, and easier to use than the existing Java Native Interface (JNI). We will delve deeper into foreign function and memory API in Chapter 9, "Harnessing Exotic Hardware: The Future of JVM Performance Engineering."

[37] https://openjdk.org/jeps/406

[38] https://openjdk.org/jeps/412

Deprecations and Removals

Java 17 is also marked by the deprecation and removal of certain features that are no longer relevant or have been superseded by newer functionalities.

JEP 398: *Deprecate the Applet API for Removal*[39] deprecates the Applet API. This API has become obsolete, as most web browsers have removed support for Java browser plug-ins.

Another significant change is the removal of the experimental ahead-of-time (AOT) and JIT compiler associated with Graal, as per JEP 410: *Remove the Experimental AOT and JIT Compiler.*[40] The Graal compiler was introduced as an experimental feature in JDK 9 and was never intended to be a long-term feature of the JDK.

Lastly, JEP 411: *Deprecate the Security Manager for Removal*[41] made the Security Manager, a feature dating back to Java 1.0, no longer relevant and marked it for removal in a future release. The Security Manager was originally designed to protect the integrity of users' machines and the confidentiality of their data by running applets in a sandbox, which denied access to resources such as the file system or the network. However, with the decline of Java applets and the rise of modern security measures, the Security Manager has become less significant.

These changes reflect the ongoing evolution of the Java platform, as it continues to adapt to modern development practices and the changing needs of the developer community. Java 17 is unequivocally the best choice for developing robust, reliable, and high-performing applications. It takes the Java legacy to the next level by significantly improving performance, reinforcing security, and enhancing developer efficiency, thereby making it the undisputed leader in the field.

Embracing Evolution for Enhanced Performance

Throughout Java's evolution, numerous improvements to the language and the JVM have optimized performance and efficiency. Developers have been equipped with innovative features and tools, enabling them to write more efficient code. Simultaneously, the JVM has seen enhancements that boost its performance, such as new garbage collectors and support for different platforms and architectures.

As Java continues to evolve, it's crucial to stay informed about the latest language features, enhancements, and JVM improvements. By doing so, you can effectively leverage these advancements in your applications, writing idiomatic and pragmatic code that fully exploits the latest developments in the Java ecosystem.

In conclusion, the advancements in Java and the JVM have consistently contributed to the performance and efficiency of Java applications. For developers, staying up-to-date with these improvements is essential to fully utilize them in your work. By understanding and applying these new features and enhancements, you can optimize your Java applications' performance and ensure their compatibility with the latest Java releases.

---

[39] https://openjdk.org/jeps/398

[40] https://openjdk.org/jeps/410

[41] https://openjdk.org/jeps/411

Chapter 2

Performance Implications of Java's Type System Evolution

The type system is a fundamental aspect of any programming language, and Java is no exception. It governs how variables are declared, how they interact with each other, and how they can be used in your code. Java is known for its (strongly) static-typed nature. To better understand what that means, let's break down this concept:

- **Java is a statically type-checked language:** That means the variable types are checked (mostly) at compile time. In contrast, dynamically type-checked languages, like JavaScript, perform type checking at runtime. This static type-checking contributes to Java's performance by catching type errors at compile time, leading to more robust and efficient code.
- **Java is a strongly typed language:** This implies that Java will produce errors at compile time if the variable's declared type does not match the type of the value assigned to it. In comparison, languages like C may implicitly convert the value's type to match the variable type. For example, because C language performs implicit conversions, it can be considered a (weakly) static-typed language. In contrast, languages such as assembly languages are untyped.

Over the years, Java's type system has evolved significantly, with each new version bringing its own set of enhancements. This evolution has made the language not only more powerful and flexible, but also easier to use and understand. For instance, the introduction of generics in Java SE 5.0 allowed developers to write more type-safe code, reducing the likelihood of runtime type errors and improving the robustness of applications. The introduction of annotations in Java SE 5.0 provided a way for developers to embed metadata directly into their code; this metadata can be used by the JVM or other tools to optimize functionality or scalability. For example, the `@FunctionalInterface` annotation introduced in Java SE 8 indicates that an interface is intended for use with lambda expressions, aiding in code clarity.

The introduction of variable handle typed references in Java 9 provided a more efficient way to access variables, such as static or instance fields, array elements, and even off-heap areas, through their handles. These handles are typed references to their variables and maintain a minimum viable access set that is compatible with the current state of the Java Memory Model

and C/C++11 atomics. These standards define how threads interact through memory and which atomic operations they can perform. This feature enhances performance and scalability for applications requiring frequent variable access.

The introduction of switch expressions and sealed classes in Java 11 to Java 17 provided more expressive and concise ways to write code, reducing the likelihood of errors and improving the readability of the code. More recently, the ongoing work on Project Valhalla aims to introduce inline classes and generics over primitive types. These features are expected to significantly improve the scalability of Java applications by reducing the memory footprint of objects and enabling flattened data structures, which provide for better memory efficiency and performance by storing data in a more compact format, which both reduces memory usage and increases cache efficiency.

In Chapter 1, "The Performance Evolution of Java: The Language and the Virtual Machine," we explored the performance evolution of Java—the language and the JVM. With that context in mind, let's delve into Java's type system and build a similar timeline of its evolution.

Java's Primitive Types and Literals Prior to J2SE 5.0

Primitive data types, along with object types and arrays, form the foundation of the simplest type system in Java. The language has reserved keywords for these primitive types, such as `int`, `short`, `long`, and `byte` for integer types. Floating-point numbers use `float` and `double` keywords; `boolean` represents boolean types; and `char` is used for characters. String objects are essentially groups of characters and are immutable in nature; they have their own class in Java: `java.lang.String`. The various types of integers and floating-point numbers are differentiated based on their size and range, as shown in Table 2.1.

Table 2.1 **Java Primitive Data Types**

Data Types: Keywords	Size	Range*
char	16-bit Unicode	\u0000–\uffff
boolean	NA	true or false
byte	8-bit signed	$-128 < b \leq 127$
short	16-bit signed	$-32768 < s \leq 32767$
int	32-bit signed	(-2^{31})–$(2^{31} - 1)$*
long	64-bit	(-2^{63})–$(2^{63} - 1)$
float	32-bit single-precision	See [1]
double	64-bit double-precision	See [1]

* Range as defined prior to J2SE 5.0.

The use of primitive types in Java contributes to both performance and scalability of applications. Primitive types are more memory efficient than their wrapper class counterparts, which

are objects that encapsulate primitive types. Thus, the use of primitive types leads to more efficient code execution.

For some primitive types, Java can handle *literals*, which represent these primitives as specific values. Here are a few examples of acceptable literals prior to J2SE 5.0:

```
String wakie = "Good morning!";
char beforeNoon = 'Y';
boolean awake = true;
byte b = 127;
short s = 32767;
int i = 2147483647;
```

In the preceding examples, `"Good morning!"` is a string literal. Similarly, `'Y'` is a char literal, `true` is a boolean literal, and `127`, `32767`, and `2147483647` are `byte`, `short`, and `int` literals, respectively.

Another important literal in Java is *null*. While it isn't associated with any object or type, *null* is used to indicate an unavailable reference or an uninitialized state. For instance, the default value for a string object is `null`.

Java's Reference Types Prior to J2SE 5.0

Java reference types are quite distinct from the primitive data types. Before J2SE 5.0, there were three types of references: interfaces, instantiable classes, and arrays. Interfaces define functional specifications for attributes and methods, whereas instantiable classes (excluding utility classes, which mainly contain static methods and are not meant to be instantiated) implement interfaces to define attributes and methods and to describe object behaviors. Arrays are fixed-size data structures. The use of reference types in Java contributes to scalability of the Java applications, as using interfaces, instantiable classes, and arrays can lead to more modular and maintainable code.

Java Interface Types

Interfaces provide the foundation for methods that are implemented in classes that might otherwise be unrelated. Interfaces can specify code execution but cannot contain any executable code. Here's a simplified example:

```
public interface WakeupDB {
    int TIME_SYSTEM = 24;

    void setWakeupHr(String name, int hrDigits);

    int getWakeupHr(String name);

}
```

In this example, both `setWakeupHr` and `getWakeupHr` are abstract methods. As you can see, there are no implementation details in these abstract methods. We will revisit interface methods when we discuss Java 8. For now, since this section covers the evolution of types only up to J2SE 5.0, just know that each class implementing the interface must include the implementation details for these interface methods. For example, a class `DevWakeupDB` that implements `WakeupDB` will need to implement both the `setWakeupHr()` and `getWakeupHr()` methods, as shown here:

```
public class DevWakeupDB implements WakeupDB {
    //…

    public void SetWakeupHr(String name, int hrDigits) {
        //…
    }

    public int getWakeupHr(String name) {
        //…
    }

    //…
}
```

The earlier example also included a declaration for `TIME_SYSTEM`. A constant declared within an interface is implicitly considered `public`, `static`, and `final`, which means it is always accessible to all classes that implement the interface and is immutable.

Figure 2.1 is a simple class diagram in which `DevWakeupDB` is a class that implements the `WakeupDB` interface. The methods `setWakeupHr` and `getWakeupHr` are defined in the `WakeupDB` interface and implemented in the `DevWakeupDB` class. The `TIME_SYSTEM` attribute is also a part of the `WakeupDB` interface.

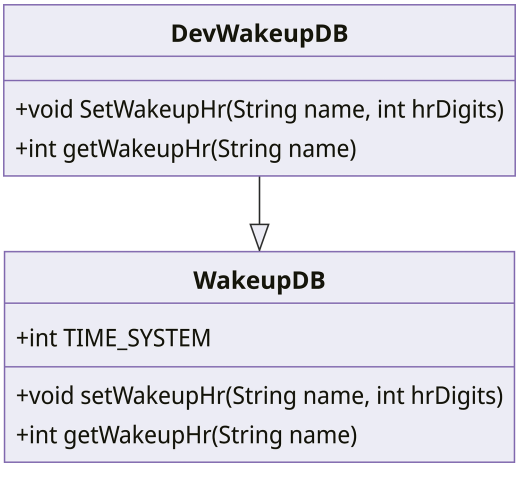

Figure 2.1 Class Diagram Showing Interface

Java Class Types

In the Java type system, a class defines a custom, aggregate data type that can contain heterogeneous data. A Java class encapsulates data and behavior and serves as a blueprint or template for creating objects. Objects have attributes (also known as fields or instance variables) that store information about the object itself. Objects also have methods, which are functions that can perform actions on or with the object's data. Let's look at an example using coding techniques prior to Java SE 5.0, with explicit field and object declarations:

```
class MorningPerson {

    private static boolean trueMorn = false;

    private static boolean Day(char c) {
        return (c == 'Y');
    }

    private static void setMornPers() throws IOException {
        System.out.println("Did you wake up before 9 AM? (Y/N)");
        trueMorn = Day((char) System.in.read());
    }

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

    public static void main(String[] args) throws IOException {
        setMornPers();
        getMornPers();
    }
}
```

This example shows a class `MorningPerson` with four methods: `Day()`, `setMornPers()`, `getMornPers()`, and `main()`. There is a static boolean class field called `trueMorn`, which stores the boolean value of a `MorningPerson` object. Each object will have access to this copy. The class field is initialized as follows:

```
private static boolean trueMorn = false;
```

If we didn't initialize the field, it would have been initialized to `false` by default, as all declared fields have their own defaults based on the data type.

Here are the two outputs depending on your input character:

```
$ java MorningPerson
Did you wake up before 9 AM? (Y/N)
N
```

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

$ java MorningPerson
Did you wake up before 9 AM? (Y/N)
Y
It is true that you are a morning person
```

Java Array Types

Like classes, arrays in Java are considered aggregate data types. However, unlike classes, arrays store homogeneous data, meaning all elements in the array must be of the same type. Let's examine Java array types using straightforward syntax for their declaration and usage.

```
public class MorningPeople {

    static String[] peopleNames = {"Monica ", "Ben ", "Bodin ", "Annika"};
    static boolean[] peopleMorn = {false, true, true, false};

    private static void getMornPeople() {
        for (int i = 0; i < peopleMorn.length; i++) {
            if (peopleMorn[i]) {
                System.out.println(peopleNames[i] + "is a morning person");
            }
        }
    }

    public static void main (String[] args){
        getMornPeople();
    }

}
```

This example includes two single-dimensional arrays: a `String[]` array and a `boolean[]` array, both containing four elements. In the `for` loop, the `string.length` property is accessed, which provides the total count of elements in the array. The expected output is shown here:

```
Ben is a morning person
Bodin is a morning person
```

Arrays provide efficient access to multiple elements of the same type. So, for example, if we want to find early risers based on the day of the week, we could modify the single-dimensional boolean array by turning it into a multidimensional one. In this new array, one dimension (e.g., columns) would represent the days of the week, while the other dimension (rows) would represent the people. This structure can be visualized as shown in Table 2.2.

Table 2.2 **A Multidimensional Array**

	Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
Monica	False	False	True	True	True	True	False
Ben	False	True	True	True	True	True	True
Bodin	True	True	True	True	True	True	False
Annika	False	True	True	True	True	True	True

This would help us get more information, so, for example, we don't disturb Monica on Saturday, Sunday, or Monday mornings. 😉

Java's Type System Evolution from J2SE 5.0 until Java SE 8

J2SE 5.0 brought significant enhancements to the Java type system, including generics, autoboxing, annotations, and enumerations. Both annotations and enumerations were new reference types for Java.

Enumerations

Enumerations provide a way to represent a related set of objects. For instance, to list common environmental allergens in the Austin, Texas, area, we could define an enumeration as follows:

```
enum Season {
    WINTER, SPRING, SUMMER, FALL
}

enum Allergens {
    MOLD(Season.WINTER),
    CEDAR(Season.WINTER),
    OAK(Season.SPRING),
    RAGWEED(Season.FALL),
    DUST(Season.WINTER);

    private Season season;

    Allergens(Season season) {
        this.season = season;
    }

    public Season getSeason() {
        return season;
    }
}
```

Enumerations enhance the type system by providing a type-safe way to define a fixed set of related objects. This can improve code readability, maintainability, and safety. Additionally, the ability to add behavior and associated data to enumerations adds flexibility and expressiveness to your code.

Annotations

Annotations (also referred to as metadata) are modifiers that are preceded by an @ symbol. Users can create their own annotation types by declaring them much as they would declare an interface. Annotation types can have methods, known as annotation type elements. However, annotation type elements can't have any parameters. Let's look at an example:

```
public @interface BookProgress {
    int chapterNum();
    String startDate(); // Format: "YYYY-MM-DD"
    String endDate(); // Format: "YYYY-MM-DD"
    boolean submittedForReview() default false;
    int revNum() default 0;
    String[] reviewerNames(); // List of names of reviewers
    boolean[] reReviewNeeded(); // Corresponding to reviewerNames, indicates if re-review needed
}
```

In this example, we have defined an annotation type called `BookProgress`. Now we can use the `BookProgress` annotation type and provide values for the elements as shown here:

```
@BookProgress(
    chapterNum = 2,
    startDate = "2018-01-01", // Format: "YYYY-MM-DD"
    endDate = "2021-02-01",   // Format: "YYYY-MM-DD"
    submittedForReview = true,
    revNum = 1,
    reviewerNames = {"Ben", "Greg", "Charlie"},
    reReviewNeeded = {true, true, false} // Each boolean corresponds to a reviewer in the same order
)

public class Chapter2 {
        //chapter content …
}
```

Annotations enhance the type system by providing a way to attach metadata to various program elements. While the annotations themselves don't directly affect program performance, they enable tools and frameworks to perform additional checks, generate code, or provide runtime behavior. This can contribute to improved code quality, maintainability, and efficiency.

Other Noteworthy Enhancements (Java SE 8)

As discussed in Chapter 1, the Java language provides some predefined annotation types, such as `@SuppressWarnings`, `@Override`, and `@Deprecated`. Java SE 7 and Java SE 8 introduced two more predefined annotation types: `@SafeVarargs` and `@FunctionalInterface`, respectively. There are a few other annotations that apply to other annotations; they are called *meta-annotations*.

In Java SE 8, generics took a step further by helping the compiler infer the parameter type using "target-type" inference. Additionally, Java SE 8 introduced type annotations that extended the usage of annotations from just declarations to anywhere a type is used. Let's look at an example of how we could use Java SE 8's "generalized target-type inference" to our advantage:

```
HashMap<@NonNull String, @NonNull List<@Readonly String>> allergyKeys;
```

In this code snippet, the `@NonNull` annotation is used to indicate that the keys and values of the `allergyKeys` *HashMap* should never be null. This can help prevent `NullPointerExceptions` at runtime. The hypothetical `@Readonly` annotation on the `String` elements of the `List` indicates that these strings should not be modified. This can be useful in situations where you want to ensure the integrity of the data in your *HashMap*.

Building on these enhancements, Java SE 8 introduced another significant feature: lambda expressions. Lambda expressions brought a new level of expressiveness and conciseness to the Java language, particularly in the context of functional programming.

A key aspect of this evolution is the concept of "target typing." In the context of lambda expressions, target typing refers to the ability of the compiler to infer the type of a lambda expression from its target context. The target context can be a variable declaration, an assignment, a return statement, an array initializer, a method or constructor invocation, or a ternary conditional expression.

For example, we can sort the allergens based on their seasons using a lambda expression:

```
List<Allergens> sortedAllergens = Arrays.stream(Allergens.values())
    .sorted(Comparator.comparing(Allergens::getSeason))
    .collect(Collectors.toList());
```

In this example, the lambda expression `Allergens::getSeason` is a method reference that is equivalent to the lambda expression `allergen -> allergen.getSeason()`. The compiler uses the target type `Comparator<Allergens>` to infer the type of the lambda expression.

Java SE 7 and Java SE 8 introduced several other enhancements, such as the use of the underscore "_" character to enhance code readability by separating digits in numeric literals. Java SE 8 added the ability to use the `int` type for unsigned 32-bit integers (range: 0 to 232 – 1) and expanded the interface types to include static and default methods, in addition to method declarations. These enhancements benefit existing classes that implemented the interface by providing default and static methods with their own implementations.

Let's look at an example of how the interface enhancements can be used:

```
public interface MorningRoutine {
    default void wakeUp() {
        System.out.println("Waking up...");
    }

    void eatBreakfast();

    static void startDay() {
        System.out.println("Starting the day!");
    }
}

public class MyMorningRoutine implements MorningRoutine {
    public void eatBreakfast() {
        System.out.println("Eating breakfast...");
    }
}
```

In this example, `MyMorningRoutine` is a class that implements the `MorningRoutine` interface. It provides its own implementation of the `eatBreakfast()` method but inherits the default `wakeUp()` method from the interface. The `startDay()` method can be called directly from the interface, like so: `MorningRoutine.startDay();`.

The enhancements in Java SE 8 provided more flexibility and extensibility to interfaces. The introduction of static and default methods allowed interfaces to have concrete implementations, reducing the need for abstract classes in certain cases and supporting better code organization and reusability.

Java's Type System Evolution: Java 9 and Java 10

Variable Handle Typed Reference

The introduction of *VarHandles* in Java 9 provided developers with a new way to access variables through their handles, offering lock-free access without the need for `java.util.concurrent.atomic`. Previously, for fast memory operations like atomic updates or prefetching on fields and elements, core API developers or performance engineers often resorted to `sun.misc.Unsafe` for JVM *intrinsics*.

With the introduction of *VarHandles*, the *MethodHandles*[1] API was updated, and changes were made at the JVM and compiler levels to support *VarHandles*. *VarHandles* themselves are represented by the `java.lang.invoke.VarHandle` class, which utilizes a signature polymorphic method. Signature polymorphic methods can have multiple definitions based on the runtime types of the arguments. This enables dynamic dispatch, with the appropriate implementation

[1] "Class MethodHandles." https://docs.oracle.com/javase/9/docs/api/java/lang/invoke/MethodHandles.html.

being selected at runtime. These methods often leverage generic types, promoting code reusability and flexibility. Furthermore, *VarHandles*, as an example of a signature polymorphic method, can handle multiple call sites and support various return types.

The variable access modes can be classified as follows based on their type:

1. Read access (e.g., `getVolatile`): For reading variables with *volatile* memory ordering effects.
2. Write access (e.g., `setVolatile`): For updating variables with *release* memory ordering effects.
3. Read–modify–write access (e.g., `compareAndExchange`, `getAndAddRelease`, `getAndBitwiseXorRelease`): Encompasses three types:
   a. Atomic update: For performing *compare-and-set* operations on variables with *volatile* memory ordering.
   b. Numeric atomic update: For performing *get-and-add* with *release* memory-order effects for updating and *acquire* memory-order effects for reading.
   c. Bitwise atomic update: For performing *get-and-bitwise-xor* with *release* memory-order effects for updating and *volatile* memory-order effects for reading.

The goals of the JDK Enhancement Proposal (JEP) for variable handles[2] were to have predictable performance (as it doesn't involve boxing or arguments packing) equivalent to what `sun.misc.Unsafe` provides, safety (no corrupt state caused by access or update), usability (superior to `sun.misc.Unsafe`), and integrity similar to that provided by `getfield`, `putfield`, and non-updating final fields.

Let's explore an example that demonstrates the creation of a *VarHandle* instance for a static field. Note that we will follow the conventions outlined in "Using JDK 9 Memory Order Modes";[3] that is, *VarHandles* will be declared as static final fields and initialized in static blocks. Also, *VarHandle* field names will be in uppercase.

```
// Declaration and Initialization Example
class SafePin {
    int pin;
}

class ModifySafePin {
    // Declare a static final VarHandle for the "pin" field
    private static final VarHandle VH_PIN;

    static {
         try {
            // Initialize the VarHandle by finding it for the "pin" field of SafePin class
            VH_PIN = MethodHandles.lookup().findVarHandle(SafePin.class, "pin", int.class);
         } catch (Exception e) {
```

[2] JEP 193: *Variable Handles*. https://openjdk.org/jeps/193.

[3] Doug Lea. "Using JDK 9 Memory Order Modes." https://gee.cs.oswego.edu/dl/html/j9mm.html.

```
            // Throw an error if VarHandle initialization fails
            throw new Error(e);
        }
    }
}
```

In this example, we have successfully created a *VarHandle* for direct access to the `pin` field. This *VarHandle* enables us to perform a bitwise or a numeric atomic update on it.

Now, let's explore an example that demonstrates how to use a *VarHandle* to perform an atomic update:

```
// Atomic Update Example
class ModifySafePin {
    private static final VarHandle VH_PIN = ModifySafePin.getVarHandle();

    private static VarHandle getVarHandle() {
          try {
              // Find and return the VarHandle for the "pin" field of SafePin class
              return MethodHandles.lookup().findVarHandle(SafePin.class, "pin", int.class);
          } catch (Exception e) {
              // Throw an error if VarHandle initialization fails
              throw new Error(e);
          }
    }

    public static void atomicUpdatePin(SafePin safePin, int newValue) {
         int prevValue;
         do {
              prevValue = (int) VH_PIN.getVolatile(safePin);
         } while (!VH_PIN.compareAndSet(safePin, prevValue, newValue));
    }
}
```

In this example, the `atomicUpdatePin` method performs an atomic update on the `pin` field of a `SafePin` object by using the *VarHandle* `VH_PIN`. It uses a loop to repeatedly get the current value of `pin` and attempts to set it to `newValue`. The `compareAndSet` method atomically sets the value of `pin` to `newValue` if the current value is equal to `prevValue`. If another thread modifies the value of `pin` between the read operation (`getVolatile`) and the compare-and-set operation (`compareAndSet`), the `compareAndSet` fails, and the loop will repeat until the update succeeds.

This is a simple example of how to use *VarHandles* to perform atomic updates. Depending on your requirements, you might need to handle exceptions differently, consider synchronization, or utilize other methods provided by *VarHandle*. You should always exercise caution when using *VarHandles* because incorrect usage can lead to data races and other concurrency issues. Thoroughly test your code and consider using higher-level concurrency utilities, when possible, to ensure proper management of concurrent variable access.

Java's Type System Evolution: Java 11 to Java 17

Switch Expressions

Java 12 introduced a significant enhancement to the traditional `switch` statement with the concept of switch expressions. This feature was further refined and finalized in Java 14. Switch expressions simplify the coding patterns for handling multiple cases by automatically inferring the type of the returned value. The `yield` keyword (introduced in Java 13) is used to return a value from a switch block.

One of the key improvements was the introduction of the arrow syntax (->) for case labels, which replaced the traditional colon syntax (:). This syntax not only makes the code more readable but also eliminates the need for `break` statements, reducing the risk of fall-through cases.

Here's an example showing how switch expressions can be used:

```
public enum Day {
   MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

public String getDayName(Day day) {
   return switch (day) {
      case MONDAY -> "Monday";
      case TUESDAY -> "Tuesday";
      case WEDNESDAY, THURSDAY -> {
         String s = day == Day.WEDNESDAY ? "Midweek" : "Almost Weekend";
         yield s; // yield is used to return a value from a block of code
      }
      // Other cases …
      default -> throw new IllegalArgumentException("Invalid day: " + day);
   };
}
```

In this example, the switch expression is used to return the name of a day based on the value of the `day` variable. The `yield` keyword is used to return a value from a block of code in the case of `WEDNESDAY` and `THURSDAY`.

Switch expressions also improve the compiler's exhaustiveness checking. As a result, the compiler can detect whether all possible cases are covered in a switch expression and issue a warning if any are missing. This feature is especially beneficial when you are dealing with `enum` values. In a traditional `switch` statement, if a new `enum` value is introduced, it could easily be overlooked in the corresponding switch cases, leading to potential bugs or unexpected behavior. However, with switch expressions, the compiler will prompt you to explicitly handle these new cases. This not only ensures code correctness but also enhances maintainability by making it easier to accommodate later changes in your `enum` definitions.

From a maintainability perspective, switch expressions can significantly improve the readability of your code. The compiler can optimize the execution of switch expressions effectively as is

possible with traditional `switch` statements. This optimization is particularly noticeable when you are dealing with cases involving strings.

In traditional `switch` statements, handling string cases involves a sequence of operations by the Java compiler, which may include techniques such as hashing and the use of lookup tables as a part of the compiled bytecode to efficiently manage to jump to the correct case label. This process may seem less direct, but the compiler is equipped to optimize these operations for a multitude of cases.

Switch expressions allow for more concise code and eliminate the possibility of fall-through, which can simplify the compiler's task when generating efficient bytecode. This is particularly helpful with multiple case labels, enabling a streamlined way to handle cases that yield the same result. This approach drastically reduces the number of comparisons needed to find the correct case, thereby significantly streamlining code complexity when there are a substantial number of cases.

Sealed Classes

Sealed classes were introduced as a preview feature in JDK 15 and finalized in JDK 17. They provide developers with a way to define a restricted class hierarchy. This feature allows for better control over class inheritance, thereby simplifying code maintenance and comprehension.

Here's an example that illustrates the concept of sealed classes using various types of pets:

```
sealed abstract class Pet permits Mammal, Reptile {}

sealed abstract class Mammal extends Pet permits Cat, Dog, PygmyPossum {
   abstract boolean isAdult();
}

final class Cat extends Mammal {
    boolean isAdult() {
      return true;
    }
    // Cat-specific properties and methods
}

// Similar implementations for Dog and PygmyPossum

sealed abstract class Reptile extends Pet permits Snake, BeardedDragon {
    abstract boolean isHarmless();
}

final class Snake extends Reptile {
    boolean isHarmless() {
        return true;
    }
    // Snake-specific properties and methods
}

// Similar implementation for BeardedDragon
```

In this example, we have a sealed class `Pet` that permits two subclasses: `Mammal` and `Reptile`. Both of these subclasses are also sealed: `Mammal` permits `Cat`, `Dog`, and `PygmyPossum`, while `Reptile` permits `Snake` and `BeardedDragon`. Each of these classes has an `isAdult` or `isHarmless` method, indicating whether the pet is an adult or is harmless, respectively. By organizing the classes in this way, we establish a clear and logical hierarchy representing different categories of pets.

This well-structured hierarchy demonstrates the power and flexibility of sealed classes in Java, which enable developers to define restricted and logically organized class hierarchies. The JVM fully supports sealed classes and can enforce the class hierarchy defined by the developer at the runtime level, leading to robust and error-free code.

Records

Java 16 introduced records as a preview feature; they became a standard feature in Java 17. Records provide a compact syntax for declaring classes that are primarily meant to encapsulate data. They represent data as a set of properties or fields and automatically generate related methods such as accessors and constructors. By default, records are immutable, which means that their state can't be changed post creation.

Let's look at an example of `Pet` records:

```
List<Pet> pets = List.of(
    new Pet("Ruby", 2, "Great Pyrenees and Red Heeler Mix", true),
    new Pet("Bash", 1, "Maltese Mix", false),
    new Pet("Perl", 10, "Black Lab Mix", true)
);

List<Pet> adultPets = pets.stream()
    .filter(Pet::isAdult)
    .collect(Collectors.toList());

adultPets.forEach(pet -> System.out.println(pet.name() + " is an adult " + pet.breed()));
```

In this code, we create a list of `Pet` records and then use a stream to filter out the pets that are adults. The `filter` method uses the `Pet::isAdult` method reference to filter the stream. The `collect` method then gathers the remaining elements of the stream into a new list. Finally, we use a `forEach` loop to print out the names and breeds of the adult pets.

Incorporating records into your code can enhance its clarity and readability. By defining the components of your class as part of the record definition, you can avoid having to write boilerplate code for constructors, accessor methods, and `equals` and `hashCode` implementations. Additionally, because records are immutable by default, they can be safely shared between threads without the need for synchronization, which can further improve performance in multithreaded environments.

Beyond Java 17: Project Valhalla

Project Valhalla is a highly anticipated enhancement to Java's type system. It aims to revolutionize the way we understand and use the Java object model and generics. The main goals are to introduce inline classes (also known as value classes or types) and user-defined primitives. The project also aims to make generics more versatile to support these new types. These changes are expected to lead to improved performance characteristics, particularly for small, immutable objects and generic APIs. To fully appreciate the potential of these changes, we need to take a closer look at the current performance implications of Java's type system.

Performance Implications of the Current Type System

Java's type system distinguishes between primitive data types and reference types (objects). Primitive data types don't possess an identity; they represent pure data values. The JVM assigns identity only to aggregate objects, such as classes and arrays, such that each instance is distinct and can be independently tracked by the JVM.

This distinction has notable performance implications. Identity allows us to mutate the state of an object, but it comes with a cost. When considering an object's state, we need to account for the object header size and the locality implications of the pointers to this object. Even for an immutable object, we still need to account for its identity and provide support for operations like `System.identityHashCode`, as the JVM needs to be prepared for potential synchronization with respect to the object.

To better understand this, let's visualize a Java object (Figure 2.2). It consists of a header and a body of fields or, in the case of an array object, array elements. The object header includes a mark word and a `klass`, which points to the class's metadata (Figure 2.3). For arrays, the header also contains a length word.

Figure 2.2 A Java Object

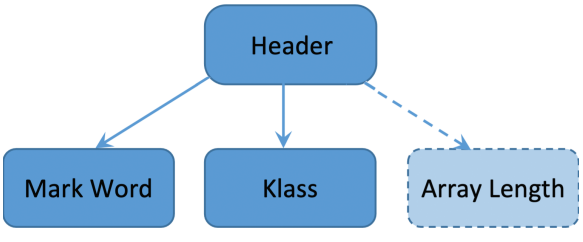

Figure 2.3 A Java Object's Header

NOTE The `klass` will be compressed if `–XX:+UseCompressedOops` is enabled.

Analyzing Object Memory Layout with Java Object Layout (JOL)

Java Object Layout (JOL)[4] is a powerful tool for understanding the memory allocation of objects. It provides a clear breakdown of the overhead associated with each object, including the object header and fields. Here's how we can analyze the layout of an 8-byte array using JOL:

```
import org.openjdk.jol.info.ClassLayout;
import org.openjdk.jol.vm.VM;
import static java.lang.System.out;

public class ArrayLayout {
    public static void main(String[] args) throws Exception {
        out.println(VM.current().details());
        byte[] ba = new byte[8];
        out.println(ClassLayout.parseInstance(ba).toPrintable());
    }
}
```

For more information on how to print internal details, refer to the following two `jol-core` files:[5]

```
org/openjdk/jol/info/ClassLayout.java
org/openjdk/jol/info/ClassData.java
```

For simplicity, let's run this code on 64-bit hardware with a 64-bit operating system and a 64-bit Java VM with `–XX:+UseCompressedOops` enabled. (This option has been enabled by default since Java 6, update 23. If you are unsure which options are enabled by default, use the `–XX:+PrintFlagsFinal` option to check.) Here's the output from JOL:

```
# Running 64-bit HotSpot VM.
# Using compressed oop with 3-bit shift.
# Using compressed klass with 3-bit shift.
# WARNING | Compressed references base/shifts are guessed by the experiment!
# WARNING | Therefore, computed addresses are just guesses, and ARE NOT RELIABLE.
# WARNING | Make sure to attach Serviceability Agent to get the reliable addresses.
# Objects are 8 bytes aligned.
# Field sizes by type: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
# Array element sizes: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
```

[4] OpenJDK. "Code Tools: jol." https://openjdk.org/projects/code-tools/jol/.

[5] Maven Repository. "Java Object Layout: Core." https://mvnrepository.com/artifact/org.openjdk.jol/jol-core.

```
[B object internals:
OFFSET SIZE TYPE DESCRIPTION     VALUE
   0     4       (object header) 01 00 00 00 (00000001 …) (1)
   4     4       (object header) 00 00 00 00 (00000000 …) (0)
   8     4       (object header) 48 68 00 00 (01001000 …) (26696)
  12     4       (object header) 08 00 00 00 (00001000 …) (8)
  16     8 byte  [B.<elements>          N/A
Instance size: 24 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total
```

Breaking down the output, we can see several sections: the JVM configuration, the byte array internals, the mark word, the `klass`, the array length, the array elements, and possible padding for alignment.

Now let's parse the output in sections.

Section 1: The first three lines

```
# Running 64-bit HotSpot VM.
# Using compressed oop with 3-bit shift.
# Using compressed klass with 3-bit shift.
```

These lines confirm that we are on a 64-bit Java VM with compressed ordinary object pointers (oops) enabled, and the `klass` is also compressed. The `klass` pointer is a critical component of the JVM as it accesses metadata for the object's class, such as its methods, fields, and superclass. This metadata is necessary for many operations, such as method invocation and field access.

Compressed oops[6] and `klass` pointers are employed to reduce memory usage on 64-bit JVMs. Their use can significantly improve performance for applications that have a large number of objects or classes.

Section 2: Byte array internals

```
[B object internals:
OFFSET SIZE TYPE DESCRIPTION     VALUE
   0     4       (object header) 01 00 00 00 (00000001 …) (1)
   4     4       (object header) 00 00 00 00 (00000000 …) (0)
```

The output also shows the internals of a byte array. Referring back to Figure 2.3, which shows the Java object's header, we can tell that the two lines after the title represent the mark word. The mark word is used by the JVM for object synchronization and garbage collection.

Section 3: `klass`

```
OFFSET SIZE TYPE DESCRIPTION     VALUE
   8     4       (object header) 48 68 00 00 (01001000 …) (26696)
```

[6] https://wiki.openjdk.org/display/HotSpot/CompressedOops

The `klass` section shows only 4 bytes because when compressed oops are enabled, the compressed `klass` is automatically enabled as well. You can try disabling compressed `klass` by using the following option: `-XX:-UseCompressedClassPointers`.

Section 4: Array length

```
OFFSET SIZE TYPE DESCRIPTION     VALUE
  12     4       (object header) 08 00 00 00 (00001000 …) (8)
```

The array length section shows a value of 8, indicating this is an 8-byte array. This value is the array length field of the header.

Section 5: Array elements

```
OFFSET SIZE TYPE DESCRIPTION     VALUE
  16     8 byte  [B.<elements>   N/A
```

The array elements section displays the body/field `array.elements`. Since this is an 8-byte array, the size is shown as 8 bytes in the clarified output. Array elements are stored in a contiguous block of memory, which can improve cache locality and performance.

Understanding the composition of an array object allows us to appreciate the overhead involved. For arrays with sizes of 1 to 8 bytes, we need 24 bytes (the required padding can range from 0 to 7 bytes, depending on the elements):

```
Instance size: 24 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total
```

The instance size and space losses are calculated to understand the memory usage of your objects better.

Now, let's examine a more complex example: an array of the `MorningPeople` class. First, we will consider the mutable version of the class:

```
public class MorningPeopleArray {

    public static void main(String[] args) throws Exception {
        out.println(VM.current().details());

        // Create an array of MorningPeople objects
        MorningPeople[] mornpplarray = new MorningPeople[8];

        // Print the layout of the MorningPeople class
        out.println(ClassLayout.parseClass(MorningPeople.class).toPrintable());

        // Print the layout of the mornpplarray instance
```

```
        out.println(ClassLayout.parseInstance(mornpplarray).toPrintable());
    }
}
class MorningPeople {
    String name;
    Boolean type;

    public MorningPeople(String name, Boolean type) {
        this.name = name;
        this.type = type;
    }
}
```

The output would now appear as follows:

```
# Running 64-bit HotSpot VM.
# Using compressed oop with 3-bit shift.
# Using compressed klass with 3-bit shift.
# WARNING | Compressed references base/shifts are guessed by the experiment!
# WARNING | Therefore, computed addresses are just guesses, and ARE NOT RELIABLE.
# WARNING | Make sure to attach Serviceability Agent to get the reliable addresses.
# Objects are 8 bytes aligned.
# Field sizes by type: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
# Array element sizes: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]

MorningPeople object internals:
 OFFSET  SIZE   TYPE               DESCRIPTION                VALUE
      0    12                      (object header)            N/A
     12     4   java.lang.String   MorningPeople.name         N/A
     16     4   java.lang.Boolean  MorningPeople.type         N/A
     20     4                      (loss due to the next object alignment)
Instance size: 24 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total

[LMorningPeople; object internals:
 OFFSET  SIZE   TYPE              DESCRIPTION               VALUE
      0     4                     (object header)           (1)
      4     4                     (object header)           (0)
      8     4                     (object header)           (13389376)
     12     4                     (object header)           (8)
     16    32   MorningPeople     MorningPeople;.<elements> N/A
Instance size: 48 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total
```

Inefficiency with Small, Immutable Objects

Consider the immutable `MorningPeople` class:

```
// An immutable class by declaring it final
final class MorningPeople {
    private final String name;
    private final Boolean type;

    public MorningPeople(String name, Boolean type) {
        this.name = name;
        this.type = type;
    }
    // Getter methods for name and type
    public String getName() {
         return name;
    }
    public Boolean getType() {
        return type;
    }
}
```

In this version of the `MorningPeople` class, the `name` and `type` fields are `private` and `final`, meaning they can't be modified once they're initialized in the constructor. The class is also declared as `final`, so it can't be subclassed. There are no setter methods, so the state of a `MorningPeople` object can't be changed after it's created. This makes the `MorningPeople` class immutable.

If we compile and run this code through JOL, the internals will still be the same as we saw with the mutable version. This outcome highlights an important point: The current Java type system is inefficient for certain types of data. This inefficiency is most evident in cases where we are dealing with large quantities of small, immutable data objects. For such cases, the memory overhead becomes significant, and the issue of locality of reference comes into play. Each `MorningPeople` object is a separate entity in memory, even though the latter is immutable. The references to these objects are scattered across the heap. This scattered layout can, in turn, adversely affect the performance of our Java programs due to poor hardware cache utilization.

The Emergence of Value Classes: Implications for Memory Management

Project Valhalla's planned introduction of value classes is poised to transform the JVM's approach to memory management. By reducing the need for object identity in certain cases, these classes are expected to lighten the garbage collector's workload, leading to fewer pauses and more efficient GC cycles. This change also aims to optimize storage and retrieval operations, especially for arrays, by leveraging the compact layout of value types. However, the path

to integrating these new classes poses challenges, particularly in managing the coexistence of value and reference types. The JVM must adapt its GC strategies and optimize cache usage to handle this new mixed model effectively.

Value classes represent a fundamental shift from traditional class structures. They lack identity, so they differ from objects in the conventional sense. They are proposed to be immutable, which would mean they cannot be synchronized or mutated post creation—attributes that are critical to maintaining state consistency. In the current Java type system, to create an immutable class, as in the `MorningPeople` example, you would typically declare the class as `final` (preventing extension), make all fields `private` (disallowing direct access), omit setter methods, make all mutable fields `final` (allowing only one-time assignment), initialize all fields via a constructor, and ensure exclusive access to any mutable components.

Value classes envision a streamlined process and promise a more memory-efficient storage method by eliminating the need for an object header, enabling inline storage akin to primitive types. This results in several key benefits:

- **Reduced memory footprint:** Inline storage significantly reduces the required memory footprint.
- **Improved locality of reference:** Inline storage ensures that all the fields of a value class are located close to each other in memory. This improved locality of reference can significantly enhance the performance of our programs by optimizing the CPU cache usage.
- **Versatile field types:** Value classes are predicted to support fields of any type, including primitives, references, and nested value classes. This flexibility to have user-defined primitive types allows for more expressive and efficient code.

Redefining Generics with Primitive Support

Recall our discussion of generics in Chapter 1, where we examined the generic type `FreshmenAdmissions<K, V>` using `String` and `Boolean` as type parameters for `K` and `V`, respectively. However, in current Java, generics are limited to reference types. Primitive types, if needed, are used via autoboxing with their wrapper classes, as demonstrated in the snippet from Chapter 1:

```
applicationStatus.admissionInformation("Monica", true);
```

Under Project Valhalla, the evolution of Java's type system aims to include support for generics over primitive types, eventually extending this support to value types. Consider the potential definition of the `FreshmenAdmissions` class in this new context:

```
public class FreshmenAdmissions<K, any V> {
    K key;
    V boolornumvalue;

    public void admissionInformation(K name, V value) {
```

```
        key = name;
        boolornumvalue = value;
    }
}
```

Here, the `any` keyword introduces a significant shift, allowing generics to directly accept primitive types without the need for boxing. For instance, in the modified `FreshmenAdmissions<K,any V>` class, `V` can now directly be a primitive type.[7] To demonstrate this enhancement, consider the `TestGenerics` class using the primitive type `boolean` instead of the wrapper class `Boolean`:

```
public class TestGenerics {
    public static void main(String[] args) {
        FreshmenAdmissions<String, boolean> applicationStatus = new FreshmenAdmissions<>();
        applicationStatus.admissionInformation("Monica", true);
    }
}
```

This example illustrates how the direct use of primitive types in generics could simplify code and improve performance, a key goal of Project Valhalla's advancements in the Java type system.

Exploring the Current State of Project Valhalla

Classes and Types

Project Valhalla introduces several novel concepts to the Java class hierarchy, such as inline (value) classes, primitive classes, and enhanced generics. Value classes, which were discussed in the preceding section, have a unique characteristic: When two instances of a value class are compared using the == operator, it checks for content equality rather than instance identity (because value classes do not have identity). Furthermore, synchronization on value classes is not possible. The JVM can use these features to save memory by avoiding unnecessary heap allocations.

Primitive classes are a special type of value class. They're not reference types, which means they can't be null. Instead, if you don't assign them a value, their values will default to zero. Primitive classes are stored directly in memory, rather than requiring a separate heap allocation. As a result, arrays of primitive classes don't need to point to different parts of the heap for each element, which makes programs run both more quickly and more efficiently. When necessary, the JVM can "box" a primitive class into a value class, and vice versa.[8]

[7] Nicolai Parlog. "Java's Evolution into 2022: The State of the Four Big Initiatives." Java Magazine, February 18, 2022. https://blogs.oracle.com/javamagazine/post/java-jdk-18-evolution-valhalla-panama-loom-amber.

[8] JEP 401: *Value Classes and Objects (Preview).* https://openjdk.org/jeps/401.

Significantly, Project Valhalla also introduces primitive classes for the eight basic types in the JVM: `byte`, `char`, `short`, `int`, `long`, `boolean`, `float`, and `double`. This makes the type system more elegant and easier to understand because it is no longer necessary to treat these types differently from other types. The introduction of these primitive classes for the basic types is a key aspect of Project Valhalla's efforts to optimize Java's performance and memory management, making the language more efficient for a wide range of applications.

Project Valhalla is working on improving generics so that they will work better with value classes. The goal is to help generic APIs perform better when used with value classes.

Memory Access Performance and Efficiency

One of the main goals of Project Valhalla is to improve memory access performance and efficiency. By allowing value classes and primitive classes to be stored directly in memory, and by improving generics so that they work better with these types, Project Valhalla aims to reduce memory usage and improve the performance of Java applications. This would be especially beneficial for lists or arrays, which can store their values directly in a block of memory, without needing separate heap pointers.

NOTE To stay updated on the latest advancements and track the ongoing progress of Project Valhalla, you have several resources at your disposal. The OpenJDK's Project Valhalla mailing list[9] is a great starting point. Additionally, you can export the GitHub repository[10] for in-depth insights and access to source codes. For hands-on experience, the latest early-access downloadable binaries[11] offer a practical way to experiment with OpenJDK prototypes, allowing you to explore many of these new features.

Early Access Release: Advancing Project Valhalla's Concepts

The early access release of Project Valhalla JEP 401: *Value Classes and Objects*[12] marks a significant step in realizing the project's ambitious goals. While previous sections discussed the theoretical framework of value classes, the early access (EA) release concretizes these concepts with specific implementations and functionalities. Key highlights include

- **Implementation of value objects:** In line with the Java Language Specification for Value Objects,[13] this release introduces value objects in Java, emphasizing their identity-free behavior and the ability to have inline, allocation-free encodings in local variables or method arguments.

[9] https://mail.openjdk.org/mailman/listinfo/valhalla-dev

[10] https://github.com/openjdk/valhalla/tree/lworld

[11] https://jdk.java.net/valhalla/

[12] https://openjdk.org/jeps/401

[13] https://cr.openjdk.org/~dlsmith/jep8277163/jep8277163-20220830/specs/value-objects-jls.html#jls-8.1.1.5

- **Enhancements and experimental features:** The EA release experiments with features like flattened fields and arrays, and developers can use the `.ref` suffix to refer to the corresponding reference type of a primitive class. This approach aims to provide null-free primitive value types, potentially transforming Java's approach to memory management and data representation.
- **Command-line options for experimentation:** Developers can use options like `-XDenablePrimitiveClasses` and `-XX:+EnablePrimitiveClasses` to unlock experimental support for *Primitive Classes*, while other options like `–XX:InlineFieldMaxFlatSize=n` and `–XX:FlatArrayElementMaxSize=n` allow developers to set limits on flattened fields and array components.
- **HotSpot optimizations:** Significant optimizations in HotSpot's C2 compiler specifically target the efficient handling of value objects, aiming to reduce the need for heap allocations.

With these enhancements, Project Valhalla steps out of the theoretical realm and into a more tangible form, showcasing the practical implications and potential of its features. As developers explore these features, it's important to note the compatibility implications. Refactoring an identity class to a value class may affect existing code. Additionally, value classes can now be declared as records and made serializable, expanding their functionality. Core reflection has also been updated to support these new modifiers. Finally, it's crucial for developers to understand that value class files generated by *javac* in this EA release are specifically tailored for this version. They may not be compatible with other JVMs or third-party tools, indicating the experimental nature of these features.

Use Case Scenarios: Bringing Theory to Practice

In high-performance computing and data processing applications, value classes can offer substantial benefits. For example, financial simulations that handle large volumes of immutable objects can leverage the memory efficiency and speed of value classes for more efficient processing.[14] Enhanced generics enable API designers to create more versatile yet less complex APIs, promoting broader applicability and easier maintenance.

Concurrent applications stand to benefit significantly from the inherent thread safety of value classes. The immutability of value classes aligns with thread safety, reducing the complexity associated with synchronization in multithreaded environments.

A Comparative Glance at Other Languages

Java's upcoming value classes, part of Project Valhalla, aim to optimize memory usage and performance similar to C#'s value types. However, Java's value classes are expected to offer enhanced features for methods and interfaces, diverging from C#'s more traditional approach with structs. While C#'s structs provide efficient memory management, they face performance costs associated with boxing when used as reference types. Java's approach is focused

[14] https://openjdk.java.net/projects/valhalla/

on avoiding such overhead and enhancing runtime efficiency, particularly in collections and arrays.[15]

Kotlin's data classes offer functional conveniences akin to Java's records, automating common methods like `equals()` and `hashCode()`, but they aren't tailored for memory optimization. Instead, Kotlin's inline classes are the choice for optimizing memory. Inline classes avoid overhead by embedding values directly in the code during compilation, which is comparable to the goals of Java's proposed value classes in Project Valhalla for reducing runtime costs.[16]

Scala's case classes are highly effective for representing immutable data structures, promoting functional programming practices. They inherently provide pattern matching, `equals()`, `hashCode()`, and `copy()` methods. However, unlike Java's anticipated value classes, Scala's case classes do not specialize in memory optimization or inline storage. Their strength lies in facilitating immutable, pattern-friendly data modeling, rather than in low-level memory management or performance optimization typical of value types.[17]

Conclusion

In this chapter, we've explored the significant strides made in Java's type system, leading up to the advancements now offered by Project Valhalla. From the foundational use of primitive and reference types to the ambitious introduction of value types and enhanced generics, Java's evolution reflects a consistent drive toward efficiency and performance.

Project Valhalla, in particular, marks a notable point in this journey, bringing forward concepts that promise to refine Java's capabilities further. These changes are pivotal for both new and experienced Java developers, offering tools for more efficient and effective coding. Looking ahead, Project Valhalla sets the stage for Java's continued relevance in the programming world—it represents a commitment to modernizing the language while maintaining its core strengths.

[15] https://docs.microsoft.com/en-us/dotnet/csharp/

[16] https://kotlinlang.org/docs/reference/data-classes.html

[17] https://docs.scala-lang.org/tour/case-classes.html

Chapter 3

From Monolithic to Modular Java: A Retrospective and Ongoing Evolution

Introduction

In the preceding chapters, we journeyed through the significant advancements in the Java language and its execution environment, witnessing the remarkable growth and transformation of these foundational elements. However, a critical aspect of Java's evolution, which has far-reaching implications for the entire ecosystem, is the transformation of the Java Development Kit (JDK) itself. As Java matured, it introduced a plethora of features and language-level enhancements, each contributing to the increased complexity and sophistication of the JDK. For instance, the introduction of the enumeration type in J2SE 5.0 necessitated the addition of the `java.lang.Enum` base class, the `java.lang.Class.getEnumConstants()` method, `EnumSet`, and `EnumMap` to the `java.util` package, along with updates to the *Serialized Form*. Each new feature or syntax addition required meticulous integration and robust support to ensure seamless functionality.

With every expansion of Java, the JDK began to exhibit signs of unwieldiness. Its monolithic structure presented challenges such as an increased memory footprint, slower start-up times, and difficulties in maintenance and updates. The release of JDK 9 marked a significant turning point in Java's history, as it introduced the Java Platform Module System (JPMS) and transitioned Java from a monolithic structure to a more manageable, modular one. This evolution continued with JDK 11 and JDK 17, with each bringing further enhancements and refinements to the modular Java ecosystem.

This chapter delves into the specifics of this transformation. We will explore the inherent challenges of the monolithic JDK and detail the journey toward modularization. Our discussion will extend to the benefits of modularization for developers, particularly focusing on those who have adopted JDK 11 or JDK 17. Furthermore, we'll consider the impact of these changes on JVM performance engineering, offering insights to help developers optimize their applications

and leverage the latest JDK innovations. Through this exploration, the goal is to demonstrate how Java applications can significantly benefit from modularization.

Understanding the Java Platform Module System

As just mentioned, the JPMS was a strategic response to the mounting complexity and unwieldiness of the monolithic JDK. The primary goal when developing it was to create a scalable platform that could effectively manage security risks at the API level while enhancing performance. The advent of modularity within the Java ecosystem empowered developers with the flexibility to select and scale modules based on the specific needs of their applications. This transformation allowed Java platform developers to use a more modular layout when managing the Java APIs, thereby fostering a system that was not only more maintainable but also more efficient. A significant advantage of this modular approach is that developers can utilize only those parts of the JDK that are necessary for their applications; this selective usage reduces the size of their applications and improves load times, leading to more efficient and performant applications.

Demystifying Modules

In Java, a module is a cohesive unit comprising packages, resources, and a module descriptor (`module-info.java`) that provides information about the module. The module serves as a container for these elements. Thus, a module

- **Encapsulates its packages:** A module can declare which of its packages should be accessible to other modules and which should be hidden. This encapsulation improves code maintainability and security by allowing developers to clearly express their code's intended usage.
- **Expresses dependencies:** A module can declare dependencies on other modules, making it clear which modules are required for that module to function correctly. This explicit dependency management simplifies the deployment process and helps developers identify problematic issues early in the development cycle.
- **Enforces strong encapsulation:** The module system enforces strong encapsulation at both compile time and runtime, making it difficult to break the encapsulation either accidentally or maliciously. This enforcement leads to better security and maintainability.
- **Boosts performance:** The module system allows the JVM to optimize the loading and execution of code, leading to improved start-up times, lower memory consumption, and faster execution.

The adoption of the module system has greatly improved the Java platform's maintainability, security, and performance.

Modules Example

Let's explore the module system by considering two example modules: `com.house.brickhouse` and `com.house.bricks`. The `com.house.brickhouse` module contains two classes, `House1` and `House2`, which calculate the number of bricks needed for houses with different levels. The `com.house.bricks` module contains a `Story` class that provides a method to count bricks based on the number of levels. Here's the directory structure for `com.house.brickhouse`:

```
src
── com.house.brickhouse
        ├── com
        │   └── house
        │       └── brickhouse
        │           ├── House1.java
        │           └── House2.java
        └── module-info.java
```

com.house.brickhouse:

module-info.java:

```
module com.house.brickhouse {
    requires com.house.bricks;
    exports com.house.brickhouse;
}
```

com/house/brickhouse/House1.java:

```
package com.house.brickhouse;
import com.house.bricks.Story;

public class House1 {
    public static void main(String[] args) {
        System.out.println("My single-level house will need " + Story.count(1) + " bricks");
    }
}
```

com/house/brickhouse/House2.java:

```
package com.house.brickhouse;
import com.house.bricks.Story;

public class House2 {
    public static void main(String[] args) {
        System.out.println("My two-level house will need " + Story.count(2) + " bricks");
    }
}
```

Now let's look at the directory structure for `com.house.bricks`:

```
src
└── com.house.bricks
    ├── com
    │   └── house
    │       └── bricks
    │           └── Story.java
    └── module-info.java

com.house.bricks:
   module-info.java:

module com.house.bricks {
    exports com.house.bricks;
}

com/house/bricks/Story.java:

package com.house.bricks;

public class Story {
    public static int count(int level) {
        return level * 18000;
    }
}
```

Compilation and Run Details

We compile the `com.house.bricks` module first:

```
$ javac -d mods/com.house.bricks src/com.house.bricks/module-info.java src/com.house.bricks/com/
house/bricks/Story.java
```

Next, we compile the `com.house.brickhouse` module:

```
$ javac --module-path mods -d mods/com.house.brickhouse
src/com.house.brickhouse/module-info.java
src/com.house.brickhouse/com/house/brickhouse/House1.java
src/com.house.brickhouse/com/house/brickhouse/House2.java
```

Now we run the `House1` example:

```
$ java --module-path mods -m com.house.brickhouse/com.house.brickhouse.House1
```

Output:

```
My single-level house will need 18000 bricks
```

Then we run the `House2` example:

```
$ java --module-path mods -m com.house.brickhouse/com.house.brickhouse.House2
```

Output:

```
My two-level house will need 36000 bricks
```

Introducing a New Module

Now, let's expand our project by introducing a new module that provides various types of bricks. We'll call this module `com.house.bricktypes`, and it will include different classes for different types of bricks. Here's the new directory structure for the `com.house.bricktypes` module:

```
src
└── com.house.bricktypes
    ├── com
    │   └── house
    │       └── bricktypes
    │           ├── ClayBrick.java
    │           └── ConcreteBrick.java
    └── module-info.java

com.house.bricktypes:
    module-info.java:

module com.house.bricktypes {
    exports com.house.bricktypes;
}
```

The `ClayBrick.java` and `ConcreteBrick.java` classes will define the properties and methods for their respective brick types.

`ClayBrick.java`:

```
package com.house.bricktypes;

public class ClayBrick {
    public static int getBricksPerSquareMeter() {
        return 60;
    }
}
```

ConcreteBrick.java:

```
package com.house.bricktypes;

public class ConcreteBrick {
    public static int getBricksPerSquareMeter() {
        return 50;
    }
}
```

With the new module in place, we need to update our existing modules to make use of these new brick types. Let's start by updating the `module-info.java` file in the `com.house.brickhouse` module:

```
module com.house.brickhouse {
    requires com.house.bricks;
    requires com.house.bricktypes;
    exports com.house.brickhouse;
}
```

We modify the `House1.java` and `House2.java` files to use the new brick types.

House1.java:

```
package com.house.brickhouse;
import com.house.bricks.Story;
import com.house.bricktypes.ClayBrick;

public class House1 {
    public static void main(String[] args) {
        int bricksPerSquareMeter = ClayBrick.getBricksPerSquareMeter();
        System.out.println("My single-level house will need "
                           + Story.count(1, bricksPerSquareMeter) + " clay bricks");
    }
}
```

House2.java:

```
package com.house.brickhouse;
import com.house.bricks.Story;
import com.house.bricktypes.ConcreteBrick;

public class House2 {
```

```
    public static void main(String[] args) {
        int bricksPerSquareMeter = ConcreteBrick.getBricksPerSquareMeter();
        System.out.println("My two-level house will need "
                           + Story.count(2, bricksPerSquareMeter) + " concrete bricks");
    }
}
```

By making these changes, we're allowing our `House1` and `House2` classes to use different types of bricks, which adds more flexibility to our program. Let's now update the `Story.java` class in the `com.house.bricks` module to accept the bricks per square meter:

```
package com.house.bricks;

public class Story {
    public static int count(int level, int bricksPerSquareMeter) {
        return level * bricksPerSquareMeter * 300;
    }
}
```

Now that we've updated our modules, let's compile and run them to see the changes in action:

- Create a new `mods` directory for the `com.house.bricktypes` module:

```
$ mkdir mods/com.house.bricktypes
```

- Compile the `com.house.bricktypes` module:

```
$ javac -d mods/com.house.bricktypes
src/com.house.bricktypes/module-info.java
src/com.house.bricktypes/com/house/bricktypes/*.java
```

- Recompile the `com.house.bricks` and `com.house.brickhouse` modules:

```
$ javac --module-path mods -d mods/com.house.bricks
src/com.house.bricks/module-info.java src/com.house.bricks/com/house/bricks/Story.java

$ javac --module-path mods -d mods/com.house.brickhouse
src/com.house.brickhouse/module-info.java
src/com.house.brickhouse/com/house/brickhouse/House1.java
src/com.house.brickhouse/com/house/brickhouse/House2.java
```

With these updates, our program is now more versatile and can handle different types of bricks. This is just one example of how the modular system in Java can make our code more flexible and maintainable.

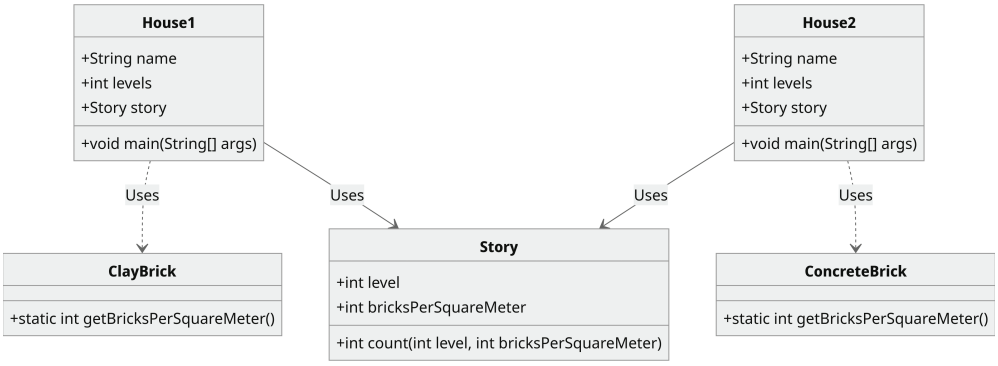

Figure 3.1 Class Diagram to Show the Relationships Between Modules

Let's now visualize these relationships with a class diagram. Figure 3.1 includes the new module `com.house.bricktypes`, and the arrows represent "Uses" relationships. `House1` uses `Story` and `ClayBrick`, whereas `House2` uses `Story` and `ConcreteBrick`. As a result, instances of `House1` and `House2` will contain references to instances of `Story` and either `ClayBrick` or `ConcreteBrick`, respectively. They use these references to interact with the methods and attributes of the `Story`, `ClayBrick`, and `ConcreteBrick` classes. Here are more details:

- `House1` and `House2`: These classes represent two different types of houses. Both classes have the following attributes:
  - `name`: A string representing the name of the house.
  - `levels`: An integer representing the number of levels in the house.
  - `story`: An instance of the `Story` class representing a level of the house.
  - `main(String[] args)`: The entry method for the class, which acts as the initial kickstarter for the application's execution.
- `Story`: This class represents a level in a house. It has the following attributes:
  - `level`: An integer representing the level number.
  - `bricksPerSquareMeter`: An integer representing the number of bricks per square meter for the level.
  - `count(int level, int bricksPerSquareMeter)`: A method that calculates the total number of bricks required for a given level and bricks per square meter.
- `ClayBrick` and `ConcreteBrick`: These classes represent two different types of bricks. Both classes have the following attributes:
  - `getBricksPerSquareMeter()`: A static method that returns the number of bricks per square meter. This method is called by the houses to obtain the value needed for calculations in the `Story` class.

Next, let's look at the use-case diagram of the Brick House Construction system with the House Owner as the actor and Clay Brick and Concrete Brick as the systems (Figure 3.2). This diagram illustrates how the House Owner interacts with the system to calculate the number of bricks required for different types of houses and choose the type of bricks for the construction.

Here's more information on the elements of the use-case diagram:

- **House Owner:** This is the actor who wants to build a house. The House Owner interacts with the Brick House Construction system in the following ways:
  - **Calculate Bricks for House 1:** The House Owner uses the system to calculate the number of bricks required to build House 1.
  - **Calculate Bricks for House 2:** The House Owner uses the system to calculate the number of bricks required to build House 2.
  - **Choose Brick Type:** The House Owner uses the system to select the type of bricks to be used for the construction.
- **Brick House Construction:** This system helps the House Owner in the construction process. It provides the following use cases:
  - **Calculate Bricks for House 1:** This use case calculates the number of bricks required for House 1. It interacts with both the Clay Brick and Concrete Brick systems to get the necessary data.
  - **Calculate Bricks for House 2:** This use case calculates the number of bricks required for House 2. It also interacts with both the Clay Brick and Concrete Brick systems to get the necessary data.
  - **Choose Brick Type:** This use case allows the House Owner to choose the type of bricks for the construction.
- **Clay Brick and Concrete Brick:** These systems provide the data (e.g., size, cost) to the Brick House Construction system that is needed to calculate the number of bricks required for the construction of the houses.

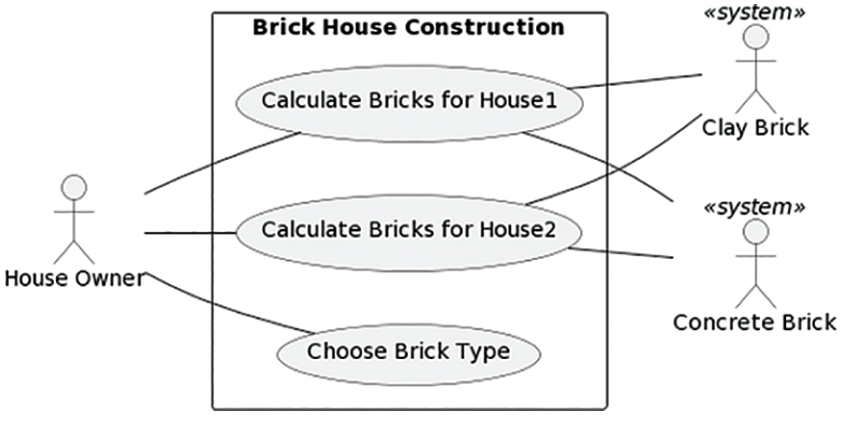

Figure 3.2 Use-Case Diagram of Brick House Construction

From Monolithic to Modular: The Evolution of the JDK

Before the introduction of the modular JDK, the bloating of the JDK led to overly complex and difficult-to-read applications. In particular, complex dependencies and cross-dependencies made it difficult to maintain and extend applications. JAR (Java Archives) hell (i.e., problems related to loading classes in Java) arose due to both the lack of simplicity and JARs' lack of awareness about the classes they contained.

The sheer footprint of the JDK also posed a challenge, particularly for smaller devices or other situations where the entire monolithic JDK wasn't needed. The modular JDK came to the rescue, transforming the JDK landscape.

Continuing the Evolution: Modular JDK in JDK 11 and Beyond

The Java Platform Module System (JPMS) was first introduced in JDK 9, and its evolution has continued in subsequent releases. JDK 11, the first long-term support (LTS) release after JDK 8, further refined the modular Java platform. Some of the notable improvements and changes made in JDK 11 are summarized here:

- **Removal of deprecated modules:** Some Java Enterprise Edition (EE) and Common Object Request Broker Architecture (CORBA) modules that had been deprecated in JDK 9 were finally removed in JDK 11. This change promoted a leaner Java platform and reduced the maintenance burden.
- **Matured module system:** The JPMS has matured over time, benefiting from the feedback of developers and real-world usage. Newer JDK releases have addressed issues, improved performance, and optimized the module system's capabilities.
- **Refined APIs:** APIs and features have been refined in subsequent releases, providing a more consistent and coherent experience for developers using the module system.
- **Continued enhancements:** JDK 11 and subsequent releases have continued to enhance the module system—for example, by offering better diagnostic messages and error reporting, improved JVM performance, and other incremental improvements that benefit developers.

Implementing Modular Services with JDK 17

With the JDK's modular approach, we can enhance the concept of services (introduced in Java 1.6) by decoupling modules that provide the service interface from their provider module, eventually creating a fully decoupled consumer. To employ services, the type is usually declared as an interface or an abstract class, and the service providers need to be clearly identified in their modules, enabling them to be recognized as providers. Lastly, consumer modules are required to utilize those providers.

To better explain the decoupling that occurs, we'll use a step-by-step example to build a `BricksProvider` along with its providers and consumers.

Service Provider

A service provider is a module that implements a service interface and makes it available for other modules to consume. It is responsible for implementing the functionalities defined in the service interface. In our example, we'll create a module called `com.example.bricksprovider`, which will implement the `BrickHouse` interface and provide the service.

Creating the `com.example.bricksprovider` Module

First, we create a new directory called `bricksprovider`; inside it, we create the `com/example/bricksprovider` directory structure. Next, we create a `module-info.java` file in the `bricksprovider` directory with the following content:

```
module com.example.bricksprovider {
    requires com.example.brickhouse;
    provides com.example.brickhouse.BrickHouse with com.example.bricksprovider.BricksProvider;
}
```

This `module-info.java` file declares that our module requires the `com.example.brickhouse` module and provides an implementation of the `BrickHouse` interface through the `com.example.bricksprovider.BricksProvider` class.

Now, we create the `BricksProvider.java` file inside the `com/example/bricksprovider` directory with the following content:

```
package com.example.bricksprovider;
import com.example.brickhouse.BrickHouse;

public class BricksProvider implements BrickHouse {
    @Override
    public void build() {
        System.out.println("Building a house with bricks...");
    }
}
```

Service Consumer

A service consumer is a module that uses a service provided by another module. It declares the service it requires in its `module-info.java` file using the `uses` keyword. The service consumer can then use the `ServiceLoader` API to discover and instantiate implementations of the required service.

Creating the `com.example.builder` Module

First, we create a new directory called `builder`; inside it, we create the `com/example/builder` directory structure. Next, we create a `module-info.java` file in the `builder` directory with the following content:

```
module com.example.builder {
    requires com.example.brickhouse;
    uses com.example.brickhouse.BrickHouse;
}
```

This `module-info.java` file declares that our module requires the `com.example.brickhouse` module and uses the `BrickHouse` service.

Now, we create a `Builder.java` file inside the `com/example/builder` directory with the following content:

```
package com.example.builder;
import com.example.brickhouse.BrickHouse;
import java.util.ServiceLoader;

public class Builder {
    public static void main(String[] args) {
        ServiceLoader<BrickHouse> loader = ServiceLoader.load(BrickHouse.class);
        loader.forEach(BrickHouse::build);
    }
}
```

A Working Example

Let's consider a simple example of a modular Java application that uses services:

- `com.example.brickhouse`: A module that defines the `BrickHouse` service interface that other modules can implement
- `com.example.bricksprovider`: A module that provides an implementation of the `BrickHouse` service and declares it in its `module-info.java` file using the `provides` keyword
- `com.example.builder`: A module that consumes the `BrickHouse` service and declares the required service in its `module-info.java` file using the `uses` keyword

The builder can then use the `ServiceLoader` API to discover and instantiate the `BrickHouse` implementation provided by the `com.example.bricksprovider` module.

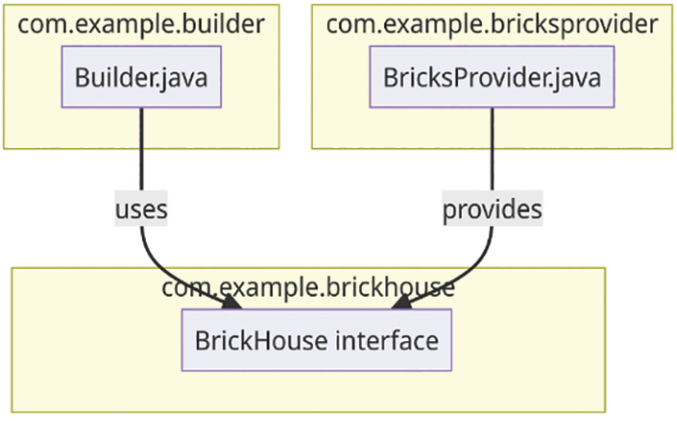

Figure 3.3 Modular Services

Figure 3.3 depicts the relationships between the modules and classes in a module diagram. The module diagram represents the dependencies and relationships between the modules and classes:

- The `com.example.builder` module contains the `Builder.java` class, which uses the `BrickHouse` interface from the `com.example.brickhouse` module.
- The `com.example.bricksprovider` module contains the `BricksProvider.java` class, which implements and provides the `BrickHouse` interface.

Implementation Details

The `ServiceLoader` API is a powerful mechanism that allows the `com.example.builder` module to discover and instantiate the `BrickHouse` implementation provided by the `com.example.bricksprovider` module at runtime. This allows for more flexibility and better separation of concerns between modules. The following subsections focus on some implementation details that can help us better understand the interactions between the modules and the role of the `ServiceLoader` API.

Discovering Service Implementations

The `ServiceLoader.load()` method takes a service interface as its argument—in our case, `BrickHouse.class`—and returns a `ServiceLoader` instance. This instance is an iterable object containing all available service implementations. The `ServiceLoader` relies on the information provided in the `module-info.java` files to discover the service implementations.

Instantiating Service Implementations

When iterating over the `ServiceLoader` instance, the API automatically instantiates the service implementations provided by the service providers. In our example, the `BricksProvider` class is instantiated, and its `build()` method is called when iterating over the `ServiceLoader` instance.

Encapsulating Implementation Details

By using the JPMS, the `com.example.bricksprovider` module can encapsulate its implementation details, exposing only the `BrickHouse` service that it provides. This allows the `com.example.builder` module to consume the service without depending on the concrete implementation, creating a more robust and maintainable system.

Adding More Service Providers

Our example can be easily extended by adding more service providers implementing the `BrickHouse` interface. As long as the new service providers are properly declared in their respective `module-info.java` files, the `com.example.builder` module will be able to discover and use them automatically through the `ServiceLoader` API. This allows for a more modular and extensible system that can adapt to changing requirements or new implementations.

Figure 3.4 is a use-case diagram that depicts the interactions between the service consumer and service provider. It includes two actors: Service Consumer and Service Provider.

- **Service Consumer:** This uses the services provided by the Service Provider. The Service Consumer interacts with the Modular JDK in the following ways:
  - **Discover Service Implementations:** The Service Consumer uses the Modular JDK to find available service implementations.
  - **Instantiate Service Implementations:** Once the service implementations are discovered, the Service Consumer uses the Modular JDK to create instances of these services.
  - **Encapsulate Implementation Details:** The Service Consumer benefits from the encapsulation provided by the Modular JDK, which allows it to use services without needing to know their underlying implementation.
- **Service Provider:** This implements and provides the services. The Service Provider interacts with the Modular JDK in the following ways:
  - **Implement Service Interface:** The Service Provider uses the Modular JDK to implement the service interface, which defines the contract for the service.
  - **Encapsulate Implementation Details:** The Service Provider uses the Modular JDK to hide the details of its service implementation, exposing only the service interface.
  - **Add More Service Providers:** The Service Provider can use the Modular JDK to add more providers for the service, enhancing the modularity and extensibility of the system.

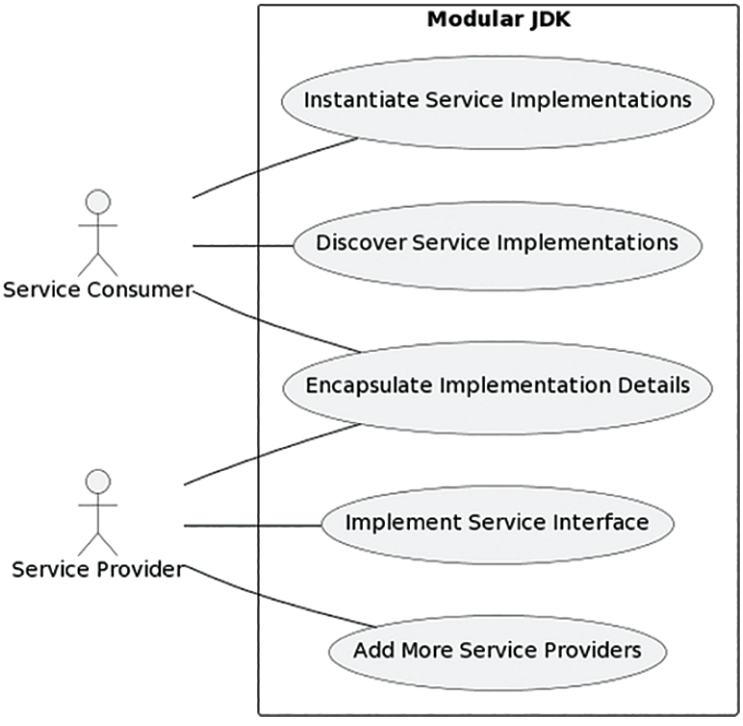

Figure 3.4 Use-Case Diagram Highlighting the Service Consumer and Service Provider

The Modular JDK acts as a robust facilitator for these interactions, establishing a comprehensive platform where service providers can effectively offer their services. Simultaneously, it provides an avenue for service consumers to discover and utilize these services efficiently. This dynamic ecosystem fosters a seamless exchange of services, enhancing the overall functionality and interoperability of modular Java applications.

JAR Hell Versioning Problem and Jigsaw Layers

Before diving into the details of the JAR hell versioning problem and Jigsaw layers, I'd like to introduce Nikita Lipski, a fellow JVM engineer and an expert in the field of Java modularity. Nikita has provided valuable insights and a comprehensive write-up on this topic, which we will be discussing in this section. His work will help us better understand the JAR hell versioning problem and how Jigsaw layers can be utilized to address this issue in JDK 11 and JDK 17.

Java's backward compatibility is one of its key features. This compatibility ensures that when a new version of Java is released, applications built for older versions can run on the new version without any changes to the source code, and often even without recompilation. The same principle applies to third-party libraries—applications can work with updated versions of the libraries without modifications to the source code.

However, this compatibility does not extend to versioning at the source level, and the JPMS does not introduce versioning at this level, either. Instead, versioning is managed at the artifact level, using artifact management systems like Maven or Gradle. These systems handle versioning and dependency management for the libraries and frameworks used in Java projects, ensuring that the correct versions of the dependencies are included in the build process. But what happens when a Java application depends on multiple third-party libraries, which in turn may depend on different versions of another library? This can lead to conflicts and runtime errors if multiple versions of the same library are present on the classpath.

So, although JPMS has certainly improved modularity and code organization in Java, the "JAR hell" problem can still be relevant when dealing with versioning at the artifact level. Let's look at an example (shown in Figure 3.5) where an application depends on two third-party libraries (Foo and Bar), which in turn depend on different versions of another library (Baz).

If both versions of the Baz library are placed on the classpath, it becomes unclear which version of the library will be used at runtime, resulting in unavoidable version conflicts. To address this issue, JPMS prohibits such situations by detecting split packages, which are not allowed in JPMS, in support of its "reliable configuration" goal (Figure 3.6).

While detecting versioning problems early is useful, JPMS does not provide a recommended way to resolve them. One approach to address these problems is to use the latest version of the conflicting library, assuming it is backward compatible. However, this might not always be possible due to introduced incompatibilities.

To address such cases, JPMS offers the *ModuleLayer* feature, which allows for the installation of a module sub-graph into the module system in an isolated manner. When different versions of the conflicting library are placed into separate layers, both of those versions can be loaded by JPMS. Although there is no direct way to access a module of the child layer from the parent layer, this can be achieved indirectly—by implementing a service provider in the child layer module, which the parent layer module can then use. (See the earlier discussion of "Implementing Modular Services with JDK 17" for more details.)

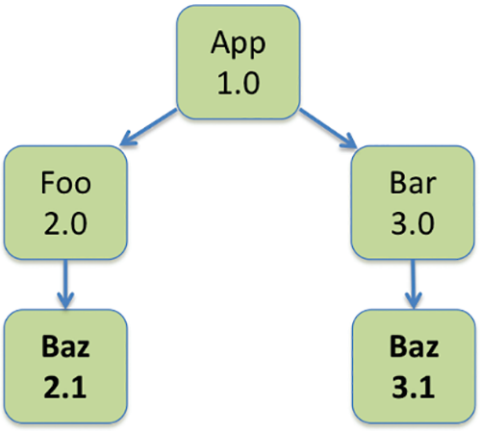

Figure 3.5 Modularity and Version Conflicts

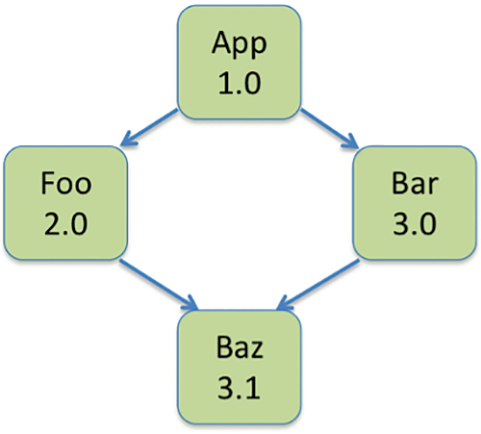

Figure 3.6 Reliable Configuration with JPMS

Working Example: JAR Hell

In this section, a working example is provided to demonstrate the use of module layers in addressing the JAR hell problem in the context of JDK 17 (this strategy is applicable to JDK 11 users as well). This example builds upon Nikita's explanation and the house service provider implementation we discussed earlier. It demonstrates how you can work with different versions of a library (termed basic and high-quality implementations) within a modular application.

First, let's take a look at the sample code provided by Java SE 9 documentation:[1]

```
1 ModuleFinder finder = ModuleFinder.of(dir1, dir2, dir3);
2 ModuleLayer parent = ModuleLayer.boot();
3 Configuration cf = parent.configuration().resolve(finder, ModuleFinder.of(),
  Set.of("myapp"));
4 ClassLoader scl = ClassLoader.getSystemClassLoader();
5 ModuleLayer layer = parent.defineModulesWithOneLoader(cf, scl);
```

In this example:

- At line 1, a `ModuleFinder` is set up to locate modules from specific directories (`dir1`, `dir2`, and `dir3`).
- At line 2, the boot layer is established as the parent layer.
- At line 3, the boot layer's configuration is resolved as the parent configuration for the modules found in the directories specified in line 1.
- At line 5, a new layer with the resolved configuration is created, using a single class loader with the system class loader as its parent.

[1] https://docs.oracle.com/javase/9/docs/api/java/lang/ModuleLayer.html

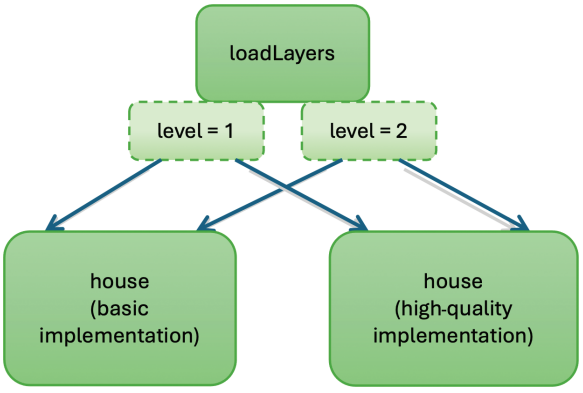

Figure 3.7 JPMS Example Versions and Layers

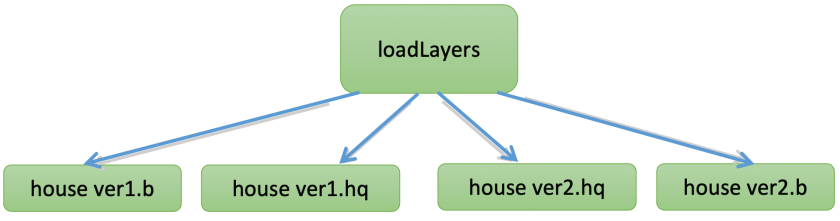

Figure 3.8 JPMS Example Version Layers Flattened

Now, let's extend our house service provider implementation. We'll have basic and high-quality implementations provided in the `com.codekaram.provider` modules. You can think of the "basic implementation" as version 1 of the house library and the "high-quality implementation" as version 2 of the house library (Figure 3.7).

For each level, we will reach out to both the libraries. So, our combinations would be level 1 + basic implementation provider, level 1 + high-quality implementation provider, level 2 + basic implementation provider, and level 2 + high-quality implementation provider. For simplicity, let's denote the combinations as *house ver1.b*, *house ver1.hq*, *house ver2.b*, and *house ver2.hq*, respectively (Figure 3.8).

Implementation Details

Building upon the concepts introduced by Nikita in the previous section, let's dive into the implementation details and understand how the layers' structure and program flow work in practice. First, let's look at the source trees:

```
ModuleLayer
├── basic
│   └── src
│       └── com.codekaram.provider
```

```
│           ├── classes
│           │   ├── com
│           │   │   └── codekaram
│           │   │       └── provider
│           │   │           └── House.java
│           │   └── module-info.java
│           └── tests
├── high-quality
│   └── src
│       └── com.codekaram.provider
│           ├── classes
│           │   ├── com
│           │   │   └── codekaram
│           │   │       └── provider
│           │   │           └── House.java
│           │   └── module-info.java
│           └── tests
└── src
    └── com.codekaram.brickhouse
        ├── classes
        │   ├── com
        │   │   └── codekaram
        │   │       └── brickhouse
        │   │           ├── loadLayers.java
        │   │           └── spi
        │   │               └── BricksProvider.java
        │   └── module-info.java
        └── tests
```

Here's the module file information and the module graph for `com.codekaram.provider`. Note that these look exactly the same for both the basic and high-quality implementations.

```
module com.codekaram.provider {
    requires com.codekaram.brickhouse;
    uses com.codekaram.brickhouse.spi.BricksProvider;
    provides com.codekaram.brickhouse.spi.BricksProvider with com.codekaram.provider.House;
}
```

The module diagram (shown in Figure 3.9) helps visualize the dependencies between modules and the services they provide, which can be useful for understanding the structure of a modular Java application:

- The `com.codekaram.provider` module depends on the `com.codekaram.brickhouse` module and implicitly depends on the `java.base` module, which is the foundational module of every Java application. This is indicated by the arrows pointing from `com`.

`codekaram.provider` to `com.codekaram.brickhouse` and the assumed arrow to `java.base`.

- The `com.codekaram.brickhouse` module also implicitly depends on the `java.base` module, as all Java modules do.

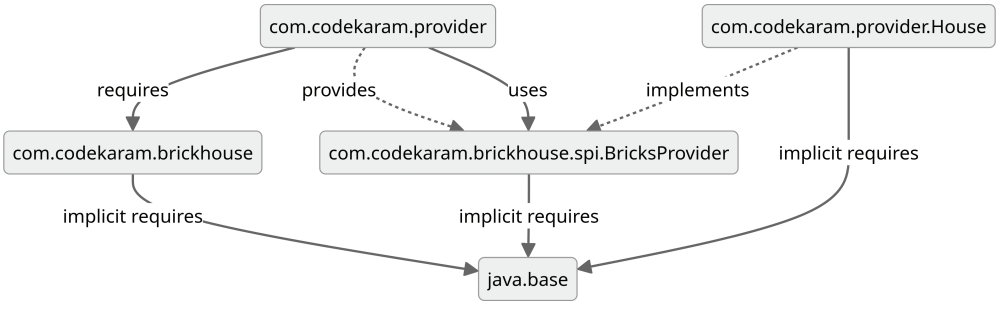

Figure 3.9 A Working Example with Services and Layers

- The `java.base` module does not depend on any other module and is the core module upon which all other modules rely.
- The `com.codekaram.provider` module provides the service `com.codekaram.brickhouse.spi.BricksProvider` with the implementation `com.codekaram.provider.House`. This relationship is represented in the graph by a dashed arrow from `com.codekaram.provider` to `com.codekaram.brickhouse.spi.BricksProvider`.

Before diving into the code for these providers, let's look at the module file information for the `com.codekaram.brickhouse` module:

```
module com.codekaram.brickhouse {
    uses com.codekaram.brickhouse.spi.BricksProvider;
    exports com.codekaram.brickhouse.spi;
}
```

The `loadLayers` class will not only handle forming layers, but also be able to load the service providers for each level. That's a bit of a simplification, but it helps us to better understand the flow. Now, let's examine the `loadLayers` implementation. Here's the creation of the layers' code based on the sample code from the "Working Example: JAR Hell" section:

```
static ModuleLayer getProviderLayer(String getCustomDir) {
   ModuleFinder finder = ModuleFinder.of(Paths.get(getCustomDir));
   ModuleLayer parent = ModuleLayer.boot();
   Configuration cf = parent.configuration().resolve(finder,
     ModuleFinder.of(), Set.of("com.codekaram.provider"));
   ClassLoader scl = ClassLoader.getSystemClassLoader();
   ModuleLayer layer = parent.defineModulesWithOneLoader(cf, scl);
```

```
   System.out.println("Created a new layer for " + layer);
   return layer;
}
```

If we simply want to create two layers, one for house version `basic` and another for house version `high-quality`, all we have to do is call `getProviderLayer()` (from the `main` method):

```
doWork(Stream.of(args)
       .map(getCustomDir -> getProviderLayer(getCustomDir)));
```

If we pass the two directories `basic` and `high-quality` as runtime parameters, the `getProviderLayer()` method will look for `com.codekaram.provider` in those directories and then create a layer for each. Let's examine the output (the line numbers have been added for the purpose of clarity and explanation):

```
1  $ java --module-path mods -m com.codekaram.brickhouse/
   com.codekaram.brickhouse.loadLayers basic high-quality

2  Created a new layer for com.codekaram.provider

3  I am the basic provider

4  Created a new layer for com.codekaram.provider

5  I am the high-quality provider
```

- Line 1 is our command-line argument with `basic` and `high-quality` as directories that provide the implementation of the `BrickProvider` service.
- Lines 2 and 4 are outputs indicating that `com.codekaram.provider` was found in both the directories and a new layer was created for each.
- Lines 3 and 5 are the output of `provider.getName()` as implemented in the `doWork()` code:

```
private static void doWork(Stream<ModuleLayer> myLayers){
myLayers.flatMap(moduleLayer -> ServiceLoader
   .load(moduleLayer, BricksProvider.class)
    .stream().map(ServiceLoader.Provider::get))
  .forEach(eachSLProvider -> System.out.println("I am the " + eachSLProvider.getName() +
" provider"));}
```

In `doWork()`, we first create a service loader for the `BricksProvider` service and load the provider from the module layer. We then print the return `String` of the `getName()` method for that provider. As seen in the output, we have two module layers and we were successful in printing the `I am the basic provider` and `I am the high-quality provider` outputs, where `basic` and `high-quality` are the return strings of the `getName()` method.

Now, let's visualize the workings of the four layers that we discussed earlier. To do so, we'll create a simple problem statement that builds a quote for basic and high-quality bricks for both levels of the house. First, we add the following code to our `main()` method:

```
int[] level = {1,2};
IntStream levels = Arrays.stream(level);
```

Next, we stream `doWork()` as follows:

```
levels.forEach(levelcount -> loadLayers
         .doWork(…
```

We now have four layers similar to those mentioned earlier (*house ver1.b*, *house ver1.hq*, *house ver2.b*, and *house ver2.hq*). Here's the updated output:

```
Created a new layer for com.codekaram.provider
My basic 1 level house will need 18000 bricks and those will cost me $6120
Created a new layer for com.codekaram.provider
My high-quality 1 level house will need 18000 bricks and those will cost me $9000
Created a new layer for com.codekaram.provider
My basic 2 level house will need 36000 bricks and those will cost me $12240
Created a new layer for com.codekaram.provider
My high-quality 2 level house will need 36000 bricks and those will be over my budget of $15000
```

NOTE The return string of the `getName()` methods for our providers has been changed to return just the `"basic"` and `"high-quality"` strings instead of an entire sentence.

The variation in the last line of the updated output serves as a demonstration of how additional conditions can be applied to service providers. Here, a budget constraint check has been integrated into the high-quality provider's implementation for a two-level house. You can, of course, customize the output and conditions as per your requirements.

Here's the updated `doWork()` method to handle both the level and the provider, along with the relevant code in the main method:

```
private static void doWork(int level, Stream<ModuleLayer> myLayers){
    myLayers.flatMap(moduleLayer -> ServiceLoader
        .load(moduleLayer, BricksProvider.class)
        .stream().map(ServiceLoader.Provider::get))
    .forEach(eachSLProvider -> System.out.println("My " + eachSLProvider.getName()
    + " " + level + " level house will need " + eachSLProvider.getBricksQuote(level)));
}
```

```
public static void main(String[] args) {
    int[] levels = {1, 2};
    IntStream levelStream = Arrays.stream(levels);

    levelStream.forEach(levelcount -> doWork(levelcount, Stream.of(args)
        .map(getCustomDir -> getProviderLayer(getCustomDir))));
}
```

Now, we can calculate the number of bricks and their cost for different levels of the house using the `basic` and `high-quality` implementations, with a separate module layer being devoted to each implementation. This demonstrates the power and flexibility that module layers provide, by enabling you to dynamically load and unload different implementations of a service without affecting other parts of your application.

Remember to adjust the service providers' code based on your specific use case and requirements. The example provided here is just a starting point for you to build on and adapt as needed.

In summary, this example illustrates the utility of Java module layers in creating applications that are both adaptable and scalable. By using the concepts of module layers and the Java `ServiceLoader`, you can create extensible applications, allowing you to adapt those applications to different requirements and conditions without affecting the rest of your codebase.

Open Services Gateway Initiative

The Open Services Gateway Initiative (OSGi) has been an alternative module system available to Java developers since 2000, long before the introduction of Jigsaw and Java module layers. As there was no built-in standard module system in Java at the time of OSGi's emergence, it addressed many modularity problems differently compared to Project Jigsaw. In this section, with insights from Nikita, whose expertise in Java modularity encompasses OSGi, we will compare Java module layers and OSGi, highlighting their similarities and differences.

OSGi Overview

OSGi is a mature and widely used framework that provides modularity and extensibility for Java applications. It offers a dynamic component model, which allows developers to create, update, and remove modules (called bundles) at runtime without restarting the application.

Similarities

- **Modularity:** Both Java module layers and OSGi promote modularity by enforcing a clear separation between components, making it easier to maintain, extend, and reuse code.
- **Dynamic loading:** Both technologies support dynamic loading and unloading of modules or bundles, allowing developers to update, extend, or remove components at runtime without affecting the rest of the application.

- **Service abstraction:** Both Java module layers (with the `ServiceLoader`) and OSGi provide service abstractions that enable loose coupling between components. This allows for greater flexibility when switching between different implementations of a service.

Differences

- **Maturity:** OSGi is a more mature and battle-tested technology, with a rich ecosystem and tooling support. Java module layers, which were introduced in JDK 9, are comparatively newer and may not have the same level of tooling and library support as OSGi.
- **Integration with Java platform:** Java module layers are a part of the Java platform, providing a native solution for modularity and extensibility. OSGi, by contrast, is a separate framework that builds on top of the Java platform.
- **Complexity:** OSGi can be more complex than Java module layers, with a steeper learning curve and more advanced features. Java module layers, while still providing powerful functionality, may be more straightforward and easier to use for developers who are new to modularity concepts.
- **Runtime environment:** OSGi applications run inside an OSGi container, which manages the life cycle of the bundles and enforces modularity rules. Java module layers run directly on the Java platform, with the module system handling the loading and unloading of modules.
- **Versioning:** OSGi provides built-in support for multiple versions of a module or bundle, allowing developers to deploy and run different versions of the same component concurrently. This is achieved by qualifying modules with versions and applying "uses constraints" to ensure safe class namespaces exist for each module. However, dealing with versions in OSGi can introduce unnecessary complexity for module resolution and for end users. In contrast, Java module layers do not natively support multiple versions of a module, but you can achieve similar functionality by creating separate module layers for each version.
- **Strong encapsulation:** Java module layers, as first-class citizens in the JDK, provide strong encapsulation by issuing error messages when unauthorized access to non-exported functionality occurs, even via reflection. In OSGi, non-exported functionality can be "hidden" using class loaders, but the module internals are still available for reflection access unless a special security manager is set. OSGi was limited by pre-JPMS features of Java SE and could not provide the same level of strong encapsulation as Java module layers.

When it comes to achieving modularity and extensibility in Java applications, developers typically have two main options: Java module layers and OSGi. Remember, the choice between Java module layers and OSGi is not always binary and can depend on many factors. These include the specific requirements of your project, the existing technology stack, and your team's familiarity with the technologies. Also, it's worth noting that Java module layers and OSGi are not the only options for achieving modularity in Java applications. Depending on your specific needs and context, other solutions might be more appropriate. It is crucial to thoroughly evaluate the pros and cons of all available options before making a decision for your project. Your choice should be based on the specific demands and restrictions of your project to ensure optimal outcomes.

On the one hand, if you need advanced features like multiple version support and a dynamic component model, OSGi may be the better option for you. This technology is ideal for complex applications that require both flexibility and scalability. However, it can be more difficult to learn and implement than Java module layers, so it may not be the best choice for developers who are new to modularity.

On the other hand, Java module layers offer a more straightforward solution for achieving modularity and extensibility in your Java applications. This technology is built into the Java platform itself, which means that developers who are already familiar with Java should find it relatively easy to use. Additionally, Java module layers offer strong encapsulation features that can help prevent dependencies from bleeding across different modules.

Introduction to Jdeps, Jlink, Jdeprscan, and Jmod

This section covers four tools that aid in the development and deployment of modular applications: *jdeps*, *jlink*, *jdeprscan*, and *jmod*. Each of these tools serves a unique purpose in the process of building, analyzing, and deploying Java applications.

Jdeps

Jdeps is a tool that facilitates analysis of the dependencies of Java classes or packages. It's particularly useful when you're trying to create a module file for JAR files. With *jdeps*, you can create various filters using regular expressions (*regex*); a regular expression is a sequence of characters that forms a search pattern. Here's how you can use *jdeps* on the `loadLayers` class:

```
$ jdeps mods/com.codekaram.brickhouse/com/codekaram/brickhouse/loadLayers.class
loadLayers.class -> java.base
loadLayers.class -> not found
   com.codekaram.brickhouse -> com.codekaram.brickhouse.spi   not found
   com.codekaram.brickhouse -> java.io                        java.base
   com.codekaram.brickhouse -> java.lang                      java.base
   com.codekaram.brickhouse -> java.lang.invoke               java.base
   com.codekaram.brickhouse -> java.lang.module               java.base
   com.codekaram.brickhouse -> java.nio.file                  java.base
   com.codekaram.brickhouse -> java.util                      java.base
   com.codekaram.brickhouse -> java.util.function             java.base
   com.codekaram.brickhouse -> java.util.stream               java.base
```

The preceding command has the same effect as passing the option `-verbose:package` to *jdeps*. The `-verbose` option by itself will list all the dependencies:

```
$ jdeps -v mods/com.codekaram.brickhouse/com/codekaram/brickhouse/loadLayers.class
loadLayers.class -> java.base
loadLayers.class -> not found
   com.codekaram.brickhouse.loadLayers -> com.codekaram.brickhouse.spi.BricksProvider  not found
   com.codekaram.brickhouse.loadLayers -> java.io.PrintStream                  java.base
   com.codekaram.brickhouse.loadLayers -> java.lang.Class                      java.base
```

```
com.codekaram.brickhouse.loadLayers -> java.lang.ClassLoader                java.base
com.codekaram.brickhouse.loadLayers -> java.lang.ModuleLayer                java.base
com.codekaram.brickhouse.loadLayers -> java.lang.NoSuchMethodException      java.base
com.codekaram.brickhouse.loadLayers -> java.lang.Object                     java.base
com.codekaram.brickhouse.loadLayers -> java.lang.String                     java.base
com.codekaram.brickhouse.loadLayers -> java.lang.System                     java.base
com.codekaram.brickhouse.loadLayers -> java.lang.invoke.CallSite            java.base
com.codekaram.brickhouse.loadLayers -> java.lang.invoke.LambdaMetafactory   java.base
com.codekaram.brickhouse.loadLayers -> java.lang.invoke.MethodHandle        java.base
com.codekaram.brickhouse.loadLayers -> java.lang.invoke.MethodHandles       java.base
com.codekaram.brickhouse.loadLayers -> java.lang.invoke.MethodHandles$Lookup java.base
com.codekaram.brickhouse.loadLayers -> java.lang.invoke.MethodType          java.base
com.codekaram.brickhouse.loadLayers -> java.lang.invoke.StringConcatFactory java.base
com.codekaram.brickhouse.loadLayers -> java.lang.module.Configuration       java.base
com.codekaram.brickhouse.loadLayers -> java.lang.module.ModuleFinder        java.base
com.codekaram.brickhouse.loadLayers -> java.nio.file.Path                   java.base
com.codekaram.brickhouse.loadLayers -> java.nio.file.Paths                  java.base
com.codekaram.brickhouse.loadLayers -> java.util.Arrays                     java.base
com.codekaram.brickhouse.loadLayers -> java.util.Collection                 java.base
com.codekaram.brickhouse.loadLayers -> java.util.ServiceLoader              java.base
com.codekaram.brickhouse.loadLayers -> java.util.Set                        java.base
com.codekaram.brickhouse.loadLayers -> java.util.Spliterator                java.base
com.codekaram.brickhouse.loadLayers -> java.util.function.Consumer          java.base
com.codekaram.brickhouse.loadLayers -> java.util.function.Function          java.base
com.codekaram.brickhouse.loadLayers -> java.util.function.IntConsumer       java.base
com.codekaram.brickhouse.loadLayers -> java.util.function.Predicate         java.base
com.codekaram.brickhouse.loadLayers -> java.util.stream.IntStream           java.base
com.codekaram.brickhouse.loadLayers -> java.util.stream.Stream              java.base
com.codekaram.brickhouse.loadLayers -> java.util.stream.StreamSupport       java.base
```

Jdeprscan

Jdeprscan is a tool that analyzes the usage of deprecated APIs in modules. Deprecated APIs are older APIs that the Java community has replaced with newer ones. These older APIs are still supported but are marked for removal in future releases. *Jdeprscan* helps developers maintain their code by suggesting alternative solutions to these deprecated APIs, aiding them in transitioning to newer, supported APIs.

Here's how you can use *jdeprscan* on the `com.codekaram.brickhouse` module:

```
$ jdeprscan --for-removal mods/com.codekaram.brickhouse
No deprecated API marked for removal found.
```

In this example, *jdeprscan* is used to scan the `com.codekaram.brickhouse` module for deprecated APIs that are marked for removal. The output indicates that no such deprecated APIs are found.

You can also use `--list` to see all deprecated APIs in a module:

```
$ jdeprscan --list mods/com.codekaram.brickhouse
No deprecated API found.
```

In this case, no deprecated APIs are found in the `com.codekaram.brickhouse` module.

Jmod

Jmod is a tool used to create, describe, and list JMOD files. JMOD files are an alternative to JAR files for packaging modular Java applications, which offer additional features such as native code and configuration files. These files can be used for distribution or to create custom runtime images with *jlink*.

Here's how you can use *jmod* to create a JMOD file for the `brickhouse` example. Let's first compile and package the module specific to this example:

```
$ javac --module-source-path src -d build/modules $(find src -name "*.java")
$ jmod create --class-path build/modules/com.codekaram.brickhouse com.codekaram.brickhouse.jmod
```

Here, the `jmod create` command is used to create a JMOD file named `com.codekaram.brickhouse.jmod` from the `com.codekaram.brickhouse` module located in the `build/modules` directory. You can then use the `jmod describe` command to display information about the JMOD file:

```
$ jmod describe com.codekaram.brickhouse.jmod
```

This command will output the module descriptor and any additional information about the JMOD file.

Additionally, you can use the `jmod list` command to display the contents of the created JMOD file:

```
$ jmod list com.codekaram.brickhouse.jmod
com/codekaram/brickhouse/
com/codekaram/brickhouse/loadLayers.class
com/codekaram/brickhouse/loadLayers$1.class
…
```

The output lists the contents of the `com.codekaram.brickhouse.jmod` file, showing the package structure and class files.

By using *jmod* to create JMOD files, describe their contents, and list their individual files, you can gain a better understanding of your modular application's structure and streamline the process of creating custom runtime images with *jlink*.

Jlink

Jlink is a tool that helps link modules and their transitive dependencies to create custom modular runtime images. These custom images can be packaged and deployed without needing the entire Java Runtime Environment (JRE), which makes your application lighter and faster to start.

To use the *jlink* command, this tool needs to be added to your path. First, ensure that the `$JAVA_HOME/bin` is in the path. Next, type `jlink` on the command line:

```
$ jlink
Error: --module-path must be specified
Usage: jlink <options> --module-path <modulepath> --add-modules <module>[,<module>...]
Use --help for a list of possible options
```

Here's how you can use *jlink* for the code shown in "Implementing Modular Services with JDK 17":

```
$ jlink --module-path $JAVA_HOME/jmods:build/modules --add-modules com.example.builder --output
consumer.services --bind-services
```

A few notes on this example:

- The command includes a directory called `$JAVA_HOME/jmods` in the `module-path`. This directory contains the `java.base.jmod` needed for all application modules.
- Because the module is a consumer of services, it's necessary to link the service providers (and their dependencies). Hence, the `–bind-services` option is used.
- The runtime image will be available in the `consumer.services` directory as shown here:

```
$ ls consumer.services/
bin conf      include   legal     lib       release
```

Let's now run the image:

```
$ consumer.services/bin/java -m com.example.builder/com.example.builder.Builder
Building a house with bricks...
```

With *jlink*, you can create lightweight, custom, stand-alone runtime images tailored to your modular Java applications, thereby simplifying the deployment and reducing the size of your distributed application.

Conclusion

This chapter has undertaken a comprehensive exploration of Java modules, tools, and techniques to create and manage modular applications. We have delved into the Java Platform Module System (JPMS), highlighting its benefits such as reliable configuration and strong encapsulation. These features contribute to more maintainable and scalable applications.

We navigated the intricacies of creating, packaging, and managing modules, and explored the use of module layers to enhance application flexibility. These practices can help address common challenges faced when migrating to newer JDK versions (e.g., JDK 11 or JDK 17), including updating project structures and ensuring dependency compatibility.

Performance Implications

The use of modular Java carries significant performance implications. By including only the necessary modules in your application, the JVM loads fewer classes, which improves start-up performance and reduces the memory footprint. This is particularly beneficial in resource-limited environments such as microservices running in containers. However, it is important to note that while modularity can improve performance, it also introduces a level of complexity. For instance, improper module design can lead to cyclic dependencies,[2] negatively impacting performance. Therefore, careful design and understanding of modules are essential to fully reap the performance benefits.

Tools and Future Developments

We examined the use of powerful tools like *jdeps*, *jdeprscan*, *jmod*, and *jlink*, which are instrumental in identifying and addressing compatibility issues, creating custom runtime images, and streamlining the deployment of modular applications. Looking ahead, we can anticipate more advanced options for creating custom runtime images with *jlink*, and more detailed and accurate dependency analysis with *jdeps*.

As more developers adopt modular Java, new best practices and patterns will emerge, alongside new tools and libraries designed to work with JPMS. The Java community is continuously improving JPMS, with future Java versions expected to refine and expand its capabilities.

Embracing the Modular Programming Paradigm

Transitioning to modular Java can present unique challenges, especially in understanding and implementing modular structures in large-scale applications. Compatibility issues may arise with third-party libraries or frameworks that may not be fully compatible with JPMS. These challenges, while part of the journey toward modernization, are often outweighed by the benefits of modular Java, such as improved performance, enhanced scalability, and better maintainability.

In conclusion, by leveraging the knowledge gained from this chapter, you can confidently migrate your projects and fully harness the potential of modular Java applications. The future of modular Java is exciting, and embracing this paradigm will equip you to meet the evolving needs of the software development landscape. It's an exciting time to be working with modular Java, and we look forward to seeing how it evolves and shapes the future of robust and efficient Java applications.

[2] https://openjdk.org/projects/jigsaw/spec/issues/#CyclicDependences

This page intentionally left blank

Chapter 4

The Unified Java Virtual Machine Logging Interface

The Java HotSpot Virtual Machine (VM) is a treasure trove of features and flexibility. Command-line options allow for enabling its experimental or diagnostic features and exploring various VM and garbage collection activities. Within the HotSpot VM, there are distinct builds tailored for different purposes. The primary version is the *product* build, optimized for performance and used in production environments. In addition, there are *debug* builds, which include the *fast-debug* and the *slow-debug* variants.

> **NOTE** The asserts-enabled *fast-debug* build offers additional debugging capabilities with minimal performance overhead, making it suitable for development environments where some level of diagnostics is needed without significantly compromising performance. On the other hand, the *slow-debug* build includes comprehensive debugging tools and checks, ideal for in-depth analysis with native debuggers but with a significant performance trade-off.

Until JDK 9, the logging interface of the HotSpot VM was fragmented, lacking unification. Developers had to combine various print options to retrieve detailed logging information with appropriate timestamps and context. This chapter guides you through the evolution of this feature, focusing on the unified logging system introduced in JDK 9, which was further enhanced in JDK 11 and JDK 17.

The Need for Unified Logging

The Java HotSpot VM is rich in options, but its lack of a unified logging interface was a significant shortcoming. Various command-line options were necessary to retrieve specific logging details, which could have been clearer and more efficient. A brief overview of these options illustrates the complexity (many more options were available, aside from those listed here):

- Garbage collector (GC) event timestamp: `-XX:+PrintGCTimeStamps` or `-XX:+PrintGCDateStamps`
- GC event details: `-XX:+PrintGCDetails`
- GC event cause: `-XX:+PrintGCCause`

- GC adaptiveness: `-XX:+PrintAdaptiveSizePolicy`
- GC-specific options (e.g., for the Garbage First [G1] GC): `-XX:+G1PrintHeapRegions`
- Compilation levels: `-XX:+PrintCompilation`
- Inlining information: `-XX:+PrintInlining`
- Assembly code: `-XX:+PrintAssembly` (requires `-XX:+UnlockDiagnosticVMOptions`)
- Class histogram: `-XX:+PrintClassHistogram`
- Information on `java.util.concurrent` locks: `-XX:+PrintConcurrentLocks`

Memorizing each option and determining whether the information is at the info, debug, or tracing level is challenging. Thus, JDK Enhancement Proposal (JEP) 158: *Unified JVM Logging*[1] was introduced in JDK 9. It was followed by JEP 271: *Unified GC Logging*,[2] which offered a comprehensive logging framework for GC activities.

Unification and Infrastructure

The Java HotSpot VM has a unified logging interface that provides a familiar look and approach to all logging information for the JVM. All log messages are characterized using tags. This enables you to request the level of information necessary for your level of investigation, from "error" and "warning" logs to "info," "trace," and "debug" levels.

To enable JVM logging, you need to provide the `-Xlog` option. By default (i.e., when no `-Xlog` is specified), only warnings and errors are logged to *stderr*. So, the default configuration looks like this:

```
-Xlog:all=warning:stderr:uptime,level,tags
```

Here, we see `all`, which is simply an alias for all tags.

The unified logging system is an all-encompassing system that provides a consistent interface for all JVM logging. It is built around four main components:

- **Tags:** Used to categorize log messages, with each message assigned one or more tags. Tags make it easier to filter and find specific log messages. Some of the predefined tags include `gc`, `compiler`, and `threads`.
- **Levels:** Define the severity or importance of a log message. Six levels are available: `debug`, `error`, `info`, `off`, `trace`, and `warning`.
- **Decorators:** Provide additional context to a log message, such as timestamps, process IDs, and more.
- **Outputs:** The destinations where the log messages are written, such as *stdout*, *stderr*, or a file.

[1] https://openjdk.org/jeps/158

[2] https://openjdk.org/jeps/271

Performance Metrics

In the context of JVM performance engineering, the unified logging system provides valuable insights into various performance metrics. These metrics include GC activities, just-in-time (JIT) compilation events, and system resource usage, among others. By monitoring these metrics, developers can identify potential performance issues and preempt necessary actions to optimize the performance of their Java applications.

Tags in the Unified Logging System

In the unified logging system, tags are crucial for segregating and identifying different types of log messages. Each tag corresponds to a particular system area or a specific type of operation. By enabling specific tags, users can tailor the logging output to focus on areas of interest.

Log Tags

The unified logging system provides a myriad of tags to capture granular information. Tags such as `gc` for garbage collection, `thread` for thread operations, `class` for class loading, `cpu` for CPU-related events, `os` for operating system interactions, and many more as highlighted here:

```
add, age, alloc, annotation, arguments, attach, barrier, biasedlocking, blocks, bot, breakpoint,
bytecode, cds, census, class, classhisto, cleanup, codecache, compaction, compilation, condy,
constantpool, constraints, container, coops, cpu, cset, data, datacreation, dcmd, decoder,
defaultmethods, director, dump, dynamic, ergo, event, exceptions, exit, fingerprint, free,
freelist, gc, handshake, hashtables, heap, humongous, ihop, iklass, indy, init, inlining, install,
interpreter, itables, jfr, jit, jni, jvmci, jvmti, lambda, library, liveness, load, loader,
logging, malloc, map, mark, marking, membername, memops, metadata, metaspace, methodcomparator,
methodhandles, mirror, mmu, module, monitorinflation, monitormismatch, nestmates, nmethod,
nmt, normalize, numa, objecttagging, obsolete, oldobject, oom, oopmap, oops, oopstorage, os,
owner, pagesize, parser, patch, path, perf, periodic, phases, plab, placeholders, preorder,
preview, promotion, protectiondomain, ptrqueue, purge, record, redefine, ref, refine, region,
reloc, remset, resolve, safepoint, sampling, scavenge, sealed, setting, smr, stackbarrier,
stackmap, stacktrace, stackwalk, start, startup, startuptime, state, stats, streaming,
stringdedup, stringtable, subclass, survivor, suspend, sweep, symboltable, system, table, task,
thread, throttle, time, timer, tlab, tracking, unload, unshareable, update, valuebasedclasses,
verification, verify, vmmutex, vmoperation, vmthread, vtables, vtablestubs, workgang
```

Specifying `all` instead of a tag combination matches all tag combinations.

If you want to run your application with all log messages, you could execute your program with `-Xlog`, which is equivalent to `-Xlog:all`. This option will log all messages to *stdout* with the level set at `info`, while warnings and errors will go to *stderr*.

Specific Tags

By default, running your application with `-Xlog` but without specific tags will produce a large volume of log messages from various system parts. If you want to focus on specific areas, you can do so by specifying the tags.

For instance, if you are interested in log messages related to garbage collection, the heap, and compressed oops, you could specify the `gc` tag. Running your application in JDK 17 with the `-Xlog:gc*` option would produce log messages related to garbage collection:

```
$ java -Xlog:gc* LockLoops
[0.006s][info][gc] Using G1
[0.007s][info][gc,init] Version: 17.0.8+9-LTS-211 (release)
[0.007s][info][gc,init] CPUs: 16 total, 16 available
…
[0.057s][info][gc,metaspace] Compressed class space mapped at: 0x0000000132000000
-0x0000000172000000, reserved size: 1073741824
[0.057s][info][gc,metaspace] Narrow klass base: 0x0000000131000000, Narrow klass shift: 0, Narrow
klass range: 0x100000000
```

Similarly, you might be interested in the `thread` tag if your application involves heavy multi-threading. Running your application with the `-Xlog:thread*` option would produce log messages related to thread operations:

```
$ java -Xlog:thread* LockLoops
[0.019s][info][os,thread] Thread attached (tid: 7427, pthread id: 123145528827904).
[0.030s][info][os,thread] Thread "GC Thread#0" started (pthread id: 123145529888768, attributes:
stacksize: 1024k, guardsize: 4k, detached).
…
```

Identifying Missing Information

In some situations, enabling a tag doesn't produce the expected log messages. For instance, running your application with `-Xlog:monitorinflation*` might produce scant output, as shown here:

```
…
[11.301s][info][monitorinflation] deflated 1 monitors in 0.0000023 secs
[11.301s][info][monitorinflation] end deflating: in_use_list stats: ceiling=11264, count=2, max=3
…
```

In such cases, you may need to adjust the log level—a topic that we'll explore in the next section.

Diving into Levels, Outputs, and Decorators

Let's take a closer look at the available log levels, examine the use of decorators, and discuss ways to redirect outputs.

Levels

The JVM's unified logging system provides various logging levels, each corresponding to a different amount of detail. To understand these levels, you can use the `-Xlog:help` command, which offers information on all available options:

```
Available log levels:
 off, trace, debug, info, warning, error
```

Here,

- **`off`:** Disable logging.
- **`error`:** Critical issues within the JVM.
- **`warning`:** Potential issues that might warrant attention.
- **`info`:** General information about JVM operations.
- **`debug`:** Detailed information useful for debugging. This level provides in-depth insights into the JVM's behavior and is typically used when troubleshooting specific issues.
- **`trace`:** The most verbose level, providing extremely detailed logging. Trace level is often used for the most granular insights into JVM operations, especially when a detailed step-by-step account of events is required.

As shown in our previous examples, the default log level is set to `info` if no specific log level is given for a tag. When we ran our test, we observed very little information for the `monitorinflation` tag when we used the default `info` log level. We can explicitly set the log level to `trace` to access additional information, as shown here:

```
$ java -Xlog:monitorinflation*=trace LockLoops
```

The output logs now contain detailed information about the various locks, as shown in the following log excerpt:

```
…
[3.073s][trace][monitorinflation      ] Checking in_use_list:
[3.073s][trace][monitorinflation      ] count=3, max=3
[3.073s][trace][monitorinflation      ] in_use_count=3 equals ck_in_use_count=3
[3.073s][trace][monitorinflation      ] in_use_max=3 equals ck_in_use_max=3
[3.073s][trace][monitorinflation      ] No errors found in in_use_list checks.
[3.073s][trace][monitorinflation      ] In-use monitor info:
[3.073s][trace][monitorinflation      ] (B -> is_busy, H -> has hash code, L -> lock status)
```

```
[3.073s][trace][monitorinflation      ]          monitor  BHL            object       object type
[3.073s][trace][monitorinflation      ] ================  ===  ================  =================
[3.073s][trace][monitorinflation      ] 0x00006000020ec680  100  0x000000070fe190e0  java.
lang.ref.ReferenceQueue$Lock (is_busy: waiters=1, contentions=0owner=0x0000000000000000,
cxq=0x0000000000000000, EntryList=0x0000000000000000)
…
```

In the unified logging system, log levels are hierarchical. Hence, setting the log level to `trace` will include messages from all lower levels (`debug`, `info`, `warning`, `error`) as well. This ensures a comprehensive logging output, capturing a wide range of details, as can be seen further down in the log:

```
…
[11.046s][trace][monitorinflation      ] deflate_monitor: object=0x000000070fe70a58,
mark=0x00006000020e00d2, type='java.lang.Object'
[11.046s][debug][monitorinflation      ] deflated 1 monitors in 0.0000157 secs
[11.046s][debug][monitorinflation      ] end deflating: in_use_list stats: ceiling=11264, count=2,
max=3
[11.046s][debug][monitorinflation      ] begin deflating: in_use_list stats: ceiling=11264,
count=2, max=3
[11.046s][debug][monitorinflation      ] deflated 0 monitors in 0.0000004 secs
…
```

This hierarchical logging is a crucial aspect of the JVM's logging system, allowing for flexible and detailed monitoring based on your requirements.

Decorators

Decorators are another critical aspect of JVM logging. Decorators enrich log messages with additional context, making them more informative. Specifying `-Xlog:help` on the command line will yield the following decorators:

```
…
Available log decorators:
 time (t), utctime (utc), uptime (u), timemillis (tm), uptimemillis (um), timenanos (tn),
uptimenanos (un), hostname (hn), pid (p), tid (ti), level (l), tags (tg)
 Decorators can also be specified as 'none' for no decoration.
…
```

These decorators provide specific information in the logs. By default, `uptime`, `level`, and `tags` are selected, which is why we saw these three decorators in the log output in our earlier examples. However, you might sometimes want to use different decorators, such as

- **`pid` (`p`):** Process ID. Useful in distinguishing logs from different JVM instances.

- **`tid` (`ti`):** Thread ID. Crucial for tracing actions of specific threads, aiding in debugging concurrency issues or thread-specific behavior.
- **`uptimemillis` (`um`):** JVM uptime in milliseconds. Helps in correlating events with the timeline of the application's operation, especially in performance monitoring.

Here's how we can add the decorators:

```
$ java -Xlog:gc*::uptimemillis,pid,tid LockLoops
[0.006s][32262][8707] Using G1
[0.008s][32262][8707] Version: 17.0.8+9-LTS-211 (release)
[0.008s][32262][8707] CPUs: 16 total, 16 available [0.023s][32033][9987] Memory: 16384M
…
[0.057s][32262][8707] Compressed class space mapped at: 0x000000012a000000-0x000000016a000000,
reserved size: 1073741824
[0.057s][32262][8707] Narrow klass base: 0x0000000129000000, Narrow klass shift: 0, Narrow klass
range: 0x100000000
```

Decorators like `pid` and `tid` are particularly valuable in debugging. When multiple JVMs are running, `pid` helps identify which JVM instance generated the log. `tid` is crucial in multi-threaded applications for tracking the behavior of individual threads, which is vital in diagnosing issues like deadlocks or race conditions.

Use `uptime` or `uptimemillis` for a time reference for events, enabling a clear timeline of operations. `uptimemillis` is essential in performance analysis to understand when specific events occur relative to the application's start time. Similarly, `hostname` can be valuable in distributed systems to identify the source machine of log entries.

Decorators enhance log readability and analysis in several ways:

- **Contextual clarity:** By providing additional details such as timestamps, process IDs, and thread IDs, decorators make logs self-contained and more understandable.
- **Filtering and searching:** With relevant decorators, log data can be filtered more effectively, enabling quicker isolation of issues.
- **Correlation and causation analysis:** Decorators allow for correlating events across different parts of the system, aiding in root-cause analysis and system-wide performance optimization.

In summary, decorators in the unified logging system play a pivotal role in enhancing the debugging and performance monitoring capabilities of JVM logs. They add necessary context and clarity, making log data more actionable and insightful for developers.

Outputs

In JVM logging, outputs determine where your log information is directed. The JVM provides flexibility in controlling the output of the logs, allowing users to redirect the output to *stdout*, *stderr*, or a specific file. The default behavior is to route all warnings and errors to *stderr*, while

all `info`, `debug`, and `trace` levels are directed to *stdout*. However, users can adjust this behavior to suit their needs. For instance, a user can specify a particular file to write the log data, which can be useful when the logs need to be analyzed later or when *stdout*/*stderr* is not convenient or desired for log data.

A file name can be passed as a parameter to the `-Xlog` command to redirect the log output to that file. For example, suppose you want to log the garbage collection (`gc`) activities at the `info` level and want these logs to be written to the `gclog.txt` file. Here's an example for the `gc*` tag and level set to `info`:

```
$ java -Xlog:gc*=info:file=gclog.txt LockLoops
```

In this example, all log information about the garbage collection process at the `info` level will be written to the `gclog.txt` file. The same can be done for any tag or log level, providing tremendous flexibility to system administrators and developers.

You can even direct different log levels to different outputs, as in the following example:

```
$ java -Xlog:monitorinflation*=trace:file=lockinflation.txt -Xlog:gc*=info:file=gclog.txt
LockLoops
```

In this case, the `info`-level logs of the garbage collection process are written into `gclog.txt`, and the `trace`-level logs for `monitorinflation` are directed to `lockinflation.txt`. This allows a fine-grained approach to managing log data and enables the segregation of log data based on its verbosity level, which can be extremely helpful during troubleshooting.

> **NOTE** Directing logs to a file rather than *stdout*/*stderr* can have performance implications, particularly for I/O-intensive applications. Writing to a file may be slower and could impact application performance. To mitigate this risk, asynchronous logging can be an effective solution. We will dive into asynchronous logging a bit later in this chapter.

Understanding and effectively managing the outputs in JVM logging can help you take full advantage of the flexibility and power of the logging system. Also, given that the JVM does not automatically handle log rotation, managing large log files becomes crucial for long-running applications. Without proper log management, log files can grow significantly, consuming disk space and potentially affecting system performance. Implementing log rotation, through either external tools or custom scripts, can help maintain log file sizes and prevent potential issues. Effective log output management not only makes application debugging and monitoring tasks more manageable, but also ensures that logging practices align with the performance and maintenance requirements of your application.

Practical Examples of Using the Unified Logging System

To demonstrate the versatility of the unified logging system, let's start by understanding how to configure it. A comprehensive grasp of this system is crucial for Java developers, particularly for efficient debugging and performance optimization. The `-Xlog` option, a central component of this system, offers extensive flexibility to tailor logging to specific needs. Let's explore its usage through various scenarios:

- **Configuring the logging system using the `-Xlog` option:** The `-Xlog` option is a powerful tool used to configure the logging system. Its flexible syntax, `-Xlog:tags:output:decorators:level`, allows you to customize what is logged, where it is logged, and how it is presented.
- **Enabling logging for specific tags:** If you want to enable logging for the `gc` and `compiler` tags at the `info` level, you can do so with the following command:

  ```
  $ java -Xlog:gc,compiler:info MyApp
  ```

- **Specifying different levels for different tags:** It's also possible to specify different levels for different tags. For instance, if you want `gc` logs at the `info` level, but `compiler` logs at the `warning` level, you can use this command:

  ```
  $ java -Xlog:gc=info,compiler=warning MyApp
  ```

- **Logging to different outputs:** By default, logs are written to *stdout*. However, you can specify a different output, such as a file. For example:

  ```
  $ java -Xlog:gc:file=gc.log MyApp
  ```

  This will write all `gc` logs to the file `gc.log`.

- **Using decorators to include additional information:** You can use decorators to include more context in your log messages. For instance, to include timestamps with your `gc` logs, you can use this command:

  ```
  $ java -Xlog:gc*:file=gc.log:time MyApp
  ```

  This will write `gc` logs to `gc.log`, with each log message being prefixed with the timestamp.

- **Enabling all logs:** For comprehensive logging, which is particularly useful during debugging or detailed analysis, you can enable all log messages across all levels with the following command:

  ```
  $ java -Xlog:all=trace MyApp
  ```

  This command ensures that every possible log message from the JVM, from the `trace` to `error` level, is captured. While this provides a complete picture of JVM operations, it can result in a large volume of log data, so it should be used judiciously.

- **Disabling logging:** If you want to turn off logging, you can do so with the following command:

  ```
  $ java -Xlog:off MyApp
  ```

Benchmarking and Performance Testing

The unified logging system is crucial in benchmarking and performance testing scenarios. By thoroughly analyzing the logs, developers can understand how their applications perform under different workloads and configurations. This information can be used to fine-tune those applications for optimal performance.

For instance, the garbage collection logs can provide insights into an application's memory usage patterns, which can guide decisions on adjusting heap size and fine-tuning other GC parameters for efficiency. In addition, other types of logs, such as JIT compilation logs, thread activity logs, or the `monitorinflation` logs discussed earlier can contribute valuable information. JIT compilation logs, for example, can help identify performance-critical methods,[3] whereas the `thread` and `monitorinflation` logs can be used to diagnose concurrency issues or inefficiencies in thread utilization.

In addition to utilizing them during performance testing, integrating these logs into regression testing frameworks is vital. It ensures that new changes or updates do not adversely affect application performance. By doing so, continuous integration pipelines can significantly streamline the process of identifying and addressing performance bottlenecks in a continuous delivery environment.

However, it's important to note that interpreting these logs often requires a nuanced understanding of JVM behavior. For instance, GC logs can be complex and necessitate a deep understanding of memory management in Java. Similarly, deciphering JIT compilation logs requires knowledge of how the JVM optimizes code execution. Balancing log analysis with other performance testing methodologies ensures a comprehensive evaluation of application performance.

Tools and Techniques

Analyzing JVM logs can be a complex endeavor due to the potential influx of data into such logs. However, the complexity of this task is significantly reduced with the aid of specialized tools and techniques. Log analysis tools are adept at parsing logs and presenting the data in a more manageable and digestible format. These tools can filter, search, and aggregate log data, making it easier to pinpoint relevant information. Similarly, visualization tools can help with pattern identification and detection of trends within the log data. They can turn textual log data into graphical representations, making it easier to spot anomalies or performance bottlenecks.

Moreover, based on historical data, machine learning techniques can predict future performance trends. This predictive analysis can be crucial in proactive system maintenance and optimization. However, effective use of these techniques requires a substantial body of historical data and a foundational understanding of data science principles.

These tools are not only instrumental in problem identification but also play a role in operational responses, such as triggering alerts or automated actions based on log analysis results. We will learn about performance techniques in Chapter 5, "End-to-End Java Performance Optimization: Engineering Techniques and Micro-benchmarking with JMH."

[3] We covered performance-critical methods in Chapter 1: "The Performance Evolution of Java: The Language and the Virtual Machine."

Optimizing and Managing the Unified Logging System

Although the unified logging system is an indispensable tool for diagnostics and monitoring, its use demands careful consideration, particularly in regard to applications' performance. Excessive logging can lead to performance degradation, as the system spends additional resources writing to the log files. Moreover, extensive logging can consume significant amounts of disk space, which could be a concern in environments with limited storage. To help manage disk space usage, log files can be compressed, rotated, or moved to secondary storage as necessary.

Thus, balancing the need for detailed logging information and the potential performance impacts is crucial. One approach to do so is to adjust the logging level to include only the most important messages in production environments. For instance, logging some of the messages at the `error` and `warning` levels might be more appropriate than logging all messages at the `info`, `debug`, or `trace` level in high-traffic scenarios.

Another consideration is the output destination for the log messages. During the application's development stages, it's recommended that log messages be directly output on *stdout*—which is why it's the default for the logging system. However, writing log messages to a file might be more suitable in a production environment, so that these messages can be archived and analyzed later as needed.

One advanced feature of the unified logging system is its ability to adjust logging levels at runtime dynamically. This capability is particularly beneficial for troubleshooting issues in live applications. For example, you might typically run your application with standard logging to maximize performance. However, during traffic surges or when you observe an increase in specific patterns, you can temporarily change the logging tag levels to gather more relevant information. Once the issue is resolved and normalcy is restored, you can revert to the original logging level without restarting the application.

This dynamic logging feature is facilitated by the `jcmd` utility, which allows you to send commands to a running JVM. For instance, to increase the logging level for the `gc*` tag to `trace`, and to add some useful decorators, you can use the following command:

```
$ jcmd <pid> VM.log what=gc*=trace decorators=uptimemillis,tid,hostname
```

Here, `<pid>` is the process ID of the running JVM. After executing this command, the JVM will start logging all garbage collection activities at the `trace` level, including the uptime in milliseconds, thread ID, and hostname decorators. This provides us with comprehensive diagnostic information.

To explore the full range of configurable options, you can use the following command:

```
$ jcmd <pid> help VM.log
```

Although the unified logging system provides a wealth of information, it's essential to use it wisely. You can effectively manage the trade-off between logging detail and system performance by adjusting the logging level, choosing an appropriate output, and leveraging the dynamic logging feature.

Asynchronous Logging and the Unified Logging System

The evolution of Java's unified logging system has resulted in another notable advanced (optional) feature: asynchronous logging. Unlike traditional synchronous logging, which (under certain conditions) can become a bottleneck in application performance, asynchronous logging delegates log-writing tasks to a separate thread. Log entries are placed in a staging buffer before being transferred to the final output. This method minimizes the impacts of logging on the main application threads—a consideration that is crucial for modern, large-scale, and latency-sensitive applications.

Benefits of Asynchronous Logging

- **Reduced latency:** By offloading logging activities, asynchronous logging significantly lowers the latency in application processing, ensuring that key operations aren't delayed by log writing.
- **Improved throughput:** In I/O-intensive environments, asynchronous logging enhances application throughput by handling log activities in parallel, freeing up the main application to handle more requests.
- **Consistency under load:** One of the most significant advantages of asynchronous logging is its ability to maintain consistent performance, even under substantial application loads. It effectively mitigates the risk of logging-induced bottlenecks, ensuring that performance remains stable and predictable.

Implementing Asynchronous Logging in Java

Using Existing Libraries and Frameworks

Java offers several robust logging frameworks equipped with asynchronous capabilities, such as Log4j2 and Logback:

- **Log4j2[4]:** Log4j2's asynchronous logger leverages the LMAX Disruptor, a high-performance interthread messaging library, to offer low-latency and high-throughput logging. Its configuration involves specifying *AsyncLogger* in the configuration file, which then directs the logging events to a ring buffer, reducing the logging overhead.
- **Logback[5]:** In Logback, asynchronous logging can be enabled through the *AsyncAppender* wrapper, which buffers log events and dispatches them to the designated appender. Configuration entails wrapping an existing appender with *AsyncAppender* and tweaking the buffer size and other parameters to balance performance and resource use.

In both frameworks, configuring the asynchronous logger typically involves XML or JSON configuration files, where various parameters such as buffer size, blocking behavior, and underlying appenders can be defined.

[4] https://logging.apache.org/log4j/2.x/manual/async.html

[5] https://logback.qos.ch/manual/appenders.html#AsyncAppender

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.

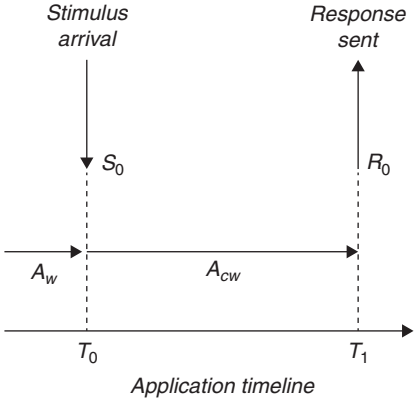

Figure 5.1 An Application Timeline Showing the Arrival of a Stimulus and the Departure of the Response

[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.

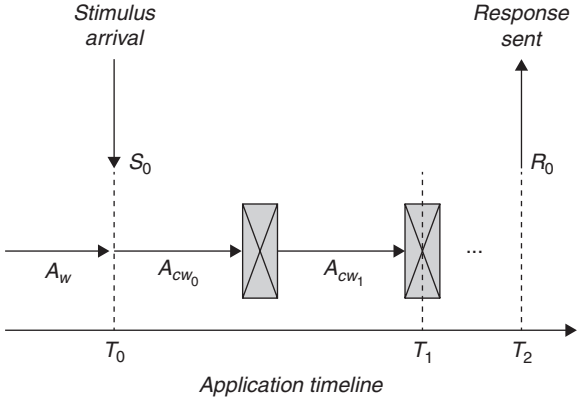

Figure 5.2 An Application Timeline with Stop-the-World Pauses

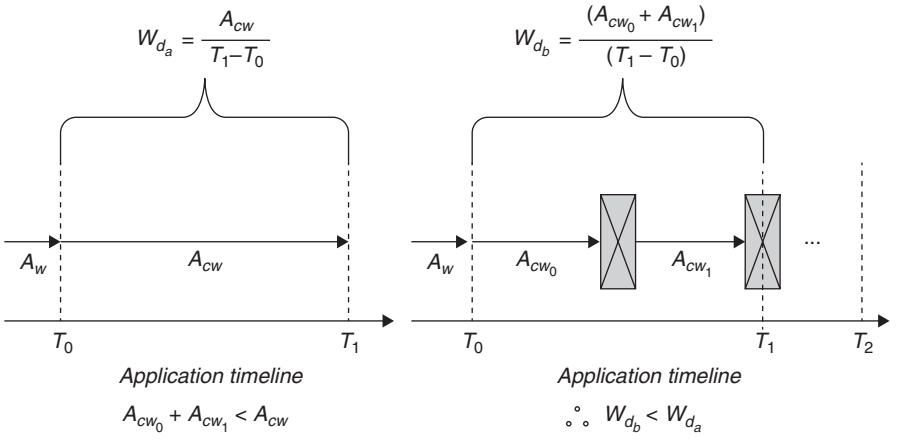

Figure 5.3 Side-by-Side Comparison to Highlight Work Done and Throughput

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.

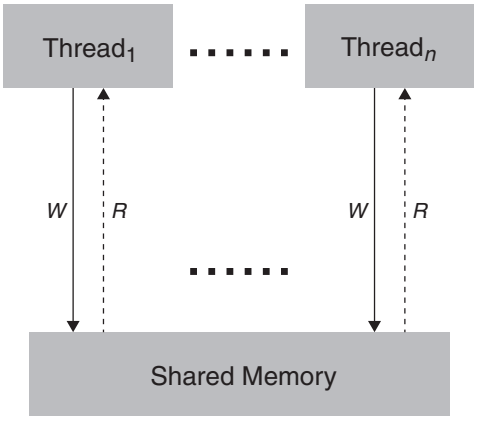

Figure 5.4 Threads 1 to *n* accessing a Shared Memory Structure Without Hindrance (Inspired by Luc Maranget, Susmit Sarkar, and Peter Sewell's *A Tutorial Introduction to the ARM and POWER Relaxed Memory Models*)

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.

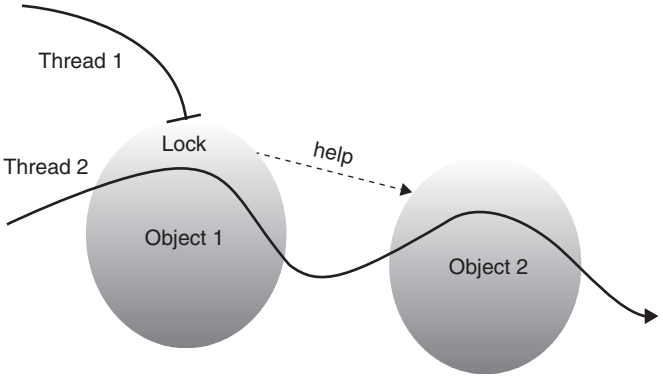

Figure 5.5 Thread 2 Locks Object 1 and Moves to the Helper in Object 2 (Inspired by Doug Lea's *Concurrent Programming in Java: Design Principles and Patterns*, 2nd ed.)

[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.

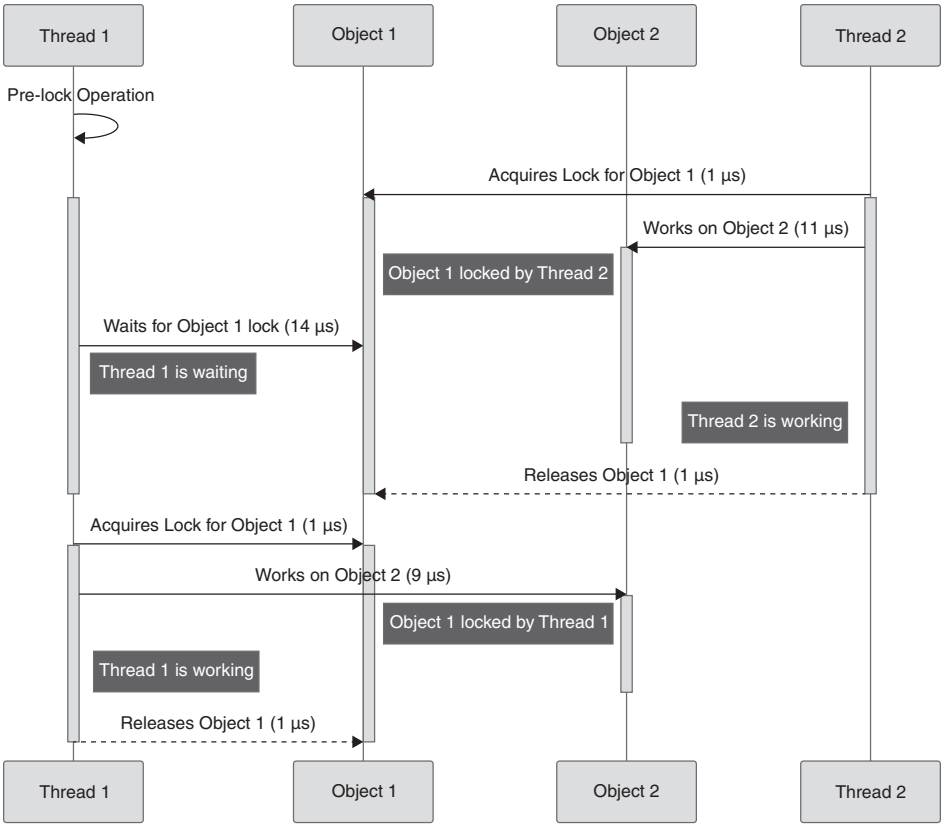

Figure 5.6 Sequence of Thread Operation in a Sequentially Consistent Shared Memory Model

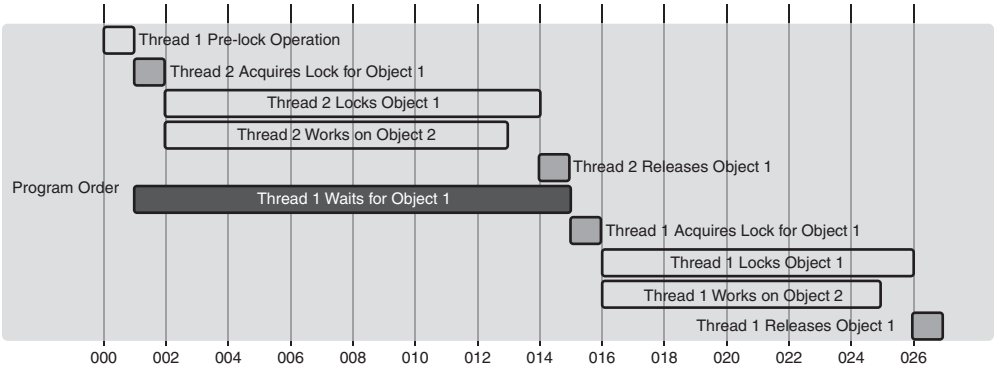

Figure 5.7 Global Execution Timeline of Thread Interactions

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

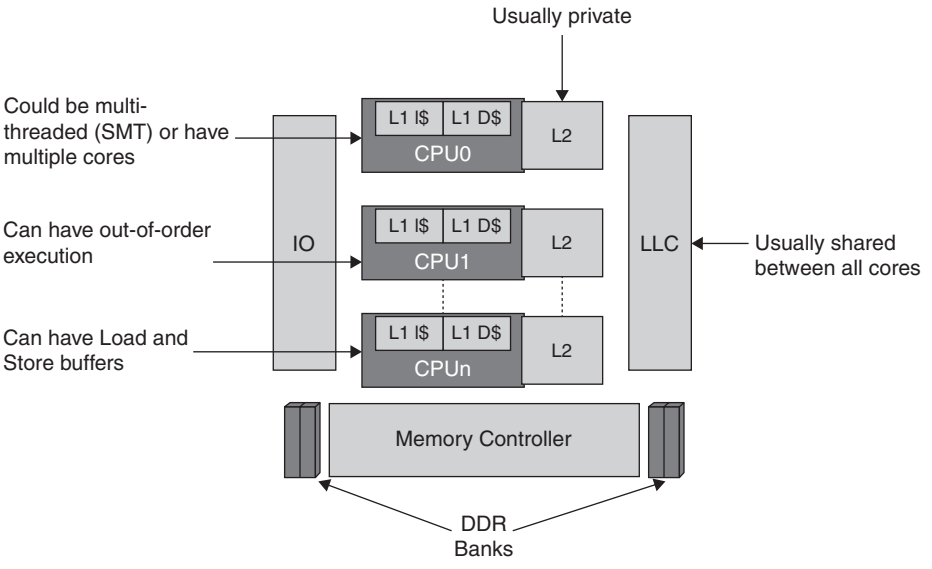

Figure 5.8 A Modern Hardware Configuration

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.

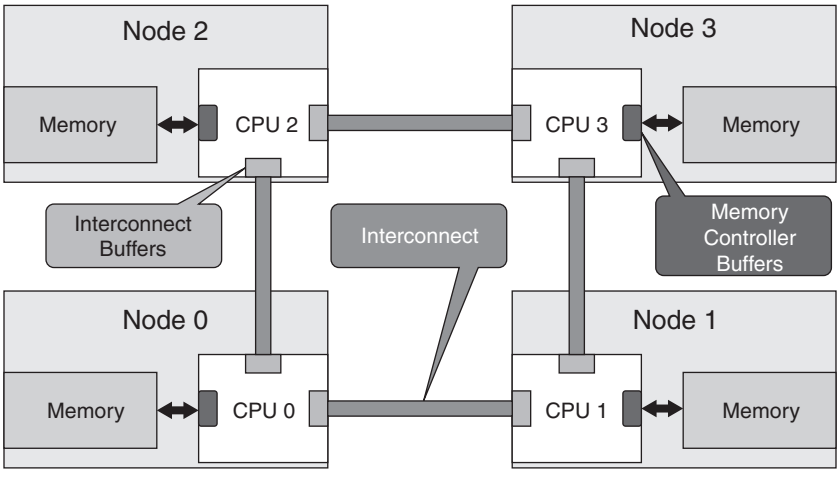

Figure 5.9 NUMA Nodes in Modern Systems

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.

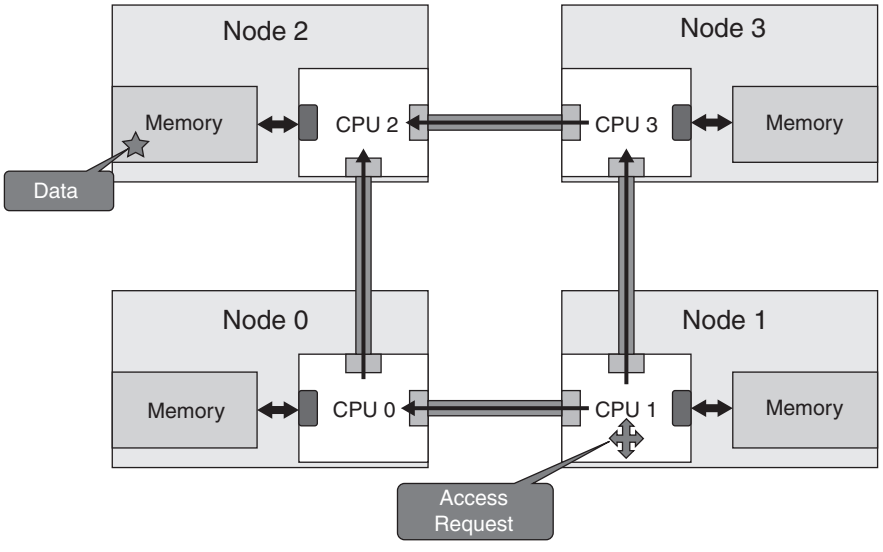

Figure 5.10 An Illustration of NUMA Access

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.

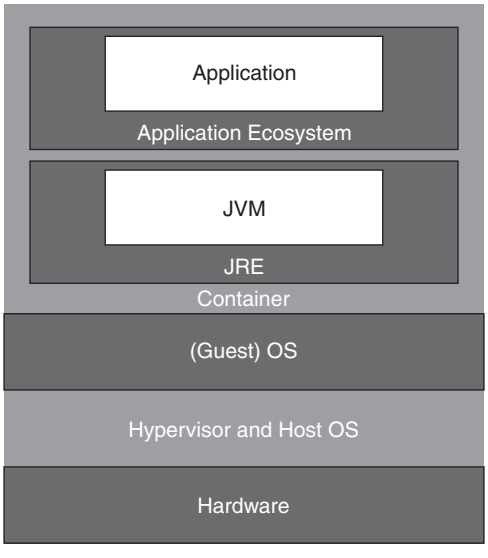

Figure 5.11 A Simplified Illustration of Different Layers of a Typical System Stack

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.

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

For each category, we will gather the following information:

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

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

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

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

JVM and Runtime Environment

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

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

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

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

Subsequently, we will gather the following data:

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

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

Garbage Collectors

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

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

Containerized Environments

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

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

OS Considerations

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

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

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

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

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

Host OS and Hypervisor Dynamics

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

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

Hardware Subsystem

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

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

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

System Profilers and Performance Monitoring Units

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

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

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

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

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

Java Application Profilers

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

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

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

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

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

Key Takeaways

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

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

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

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

The Importance of Performance Benchmarking

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

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

Key Performance Metrics

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

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

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

The Performance Benchmark Regime: From Planning to Analysis

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

The Performance Benchmarking Process

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

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

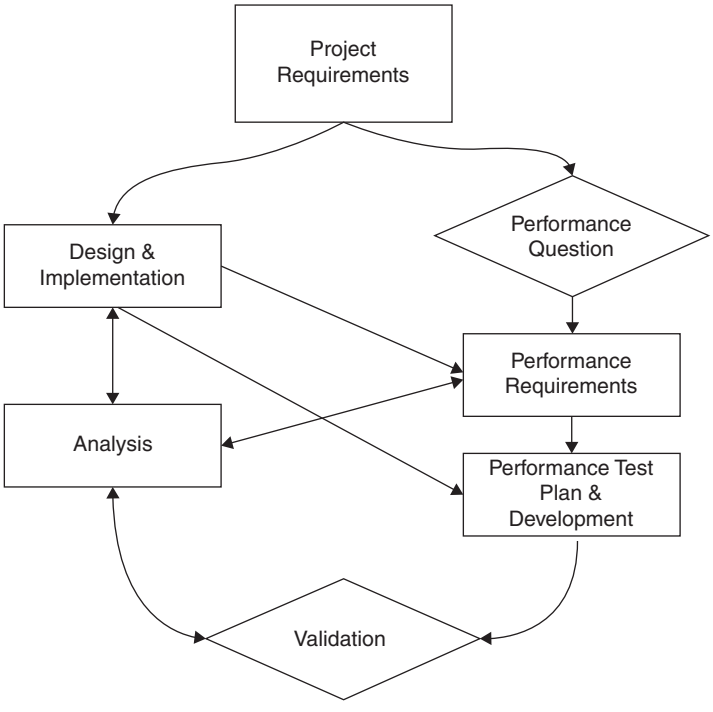

Figure 5.12 Benchmarking Process

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

Iterative Approach to Benchmarking

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

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

Benchmarking JVM Memory Management: A Comprehensive Guide

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

Setting Clear Objectives

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

Key Metrics to Benchmark

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

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

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

Choosing the Right Tools

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

Set Up Your Benchmarking Environment

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

Benchmark Execution Protocol

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

Analyze Results

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

The Refinement Cycle

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

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

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

Why Do We Need a Benchmarking Harness?

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

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

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

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

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

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

Key Features of JMH for Performance Optimization

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

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

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

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

Getting Started with Maven

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

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

Setting Up JMH and Understanding Maven Archetype

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

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

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

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

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

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

```
package org.jmh.suite;

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

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

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

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

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

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

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

}
```

The directory structure of the project would look like this:

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

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

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

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

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

The JMH project then presents the aggregate information after completion:

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

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

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

Benchmark Phases: Warm-Up and Measurement

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

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

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

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

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

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

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

Loop Optimizations and @OperationsPerInvocation

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

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

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

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

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

Benchmarking Modes in JMH

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

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

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

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

Understanding the Profilers in JMH

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

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

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

Key Annotations in JMH

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

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

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

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

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

JVM Benchmarking with JMH

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Profiling JMH Benchmarks with perfasm

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

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

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

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

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

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

Conclusion

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

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

This page intentionally left blank

Chapter 6

Advanced Memory Management and Garbage Collection in OpenJDK

Introduction

In this chapter, we will dive into the world of JVM performance engineering, examining advanced memory management and garbage collection techniques in the OpenJDK Hotspot VM. Our focus will be on the evolution of optimizations and algorithms from JDK 11 to JDK 17.

Automatic memory management in modern OpenJDK is a crucial aspect of the JVM. It empowers developers to leverage optimized algorithms that provide efficient allocation paths and adapt to the applications' demands. Operating within a managed runtime environment minimizes the chances of memory leaks, dangling pointers, and other memory-related bugs that can be challenging to identify and resolve.

My journey with GCs started during my tenure at Advanced Micro Devices (AMD) and then gained wings at Sun Microsystems and Oracle, where I was deeply involved in the evolution of garbage collection technologies. These experiences provided me with a unique perspective on the development and optimization of garbage collectors, which I'll share with you in this chapter.

Among the many enhancements to the JVM, JDK 11 introduced significant improvements to garbage collection, such as abortable mixed collections, which improved GC responsiveness, and the Z Garbage Collector, which aimed to achieve sub-millisecond pauses.

This chapter explores these developments, including optimizations like Thread-Local Allocation Buffers and Non-Uniform Memory Architecture–aware GC. We will also delve into various transactional workloads, examining their interplay with in-memory databases and their impact on the Java heap.

By the end of this chapter, you'll have a deeper understanding of advanced memory management in OpenJDK and be equipped with the knowledge to effectively optimize GC for your Java applications. So, let's dive in and explore the fascinating world of advanced memory management and garbage collection!

Overview of Garbage Collection in Java

Java's GC is an automatic, adaptive memory management system that recycles heap memory used by objects no longer needed by the application. For most GCs in OpenJDK, the JVM segments heap memory into different generations, with most objects allocated in the young generation, which is further divided into the eden and survivor spaces. Objects move from the eden space to the survivor space, and potentially to the old generation, as they survive subsequent minor GC cycles.

This generational approach leverages the "weak generational hypothesis," which states that most objects become unreachable quickly, and there are fewer references from old to young objects. By focusing on the young generation during minor GC cycles, the GC process becomes more efficient.

In the GC process, the collector first traces the object graph starting from root nodes—objects directly accessible from an application. This phase, also known as marking, involves explicitly identifying all live objects. Concurrent collectors might also implicitly consider certain objects as live, contributing to the GC footprint and leading to a phenomenon known as "floating garbage." This occurs when the GC retains references that may no longer be live. After completing the tracing, any objects not reachable during this traversal are considered "garbage" and can be safely deallocated.

OpenJDK HotSpot VM offers several built-in garbage collectors, each designed with specific goals, from increasing throughput and reducing latency to efficiently handling larger heap sizes. In our exploration, we'll focus on two of these collectors—the G1 Garbage Collector and Z Garbage Collector (ZGC). My direct involvement in algorithmic refinement and performance optimization of the G1 collector has given me in-depth insights into its advanced memory management techniques. Along with the G1 GC, we'll also explore ZGC, particularly focusing on their recent enhancements in OpenJDK.

The G1 GC, a revolutionary feature fully supported since JDK 7 update 4, is designed to maximize the capabilities of modern multi-core processors and large memory banks. More than just a garbage collector, G1 is a comprehensive solution that balances responsiveness with throughput. It consistently meets GC pause-time goals, ensuring smooth operation of applications while delivering impressive processing capacity. G1 operates in the background, concurrently with the application, traversing the live object graph and building the collection set for different regions of the heap. It minimizes pauses by splitting the heap into smaller, more manageable pieces, known as regions, and processing each as an independent entity, thereby enhancing the overall responsiveness of the application.

The concept of regions is central to the operation of both G1 and ZGC. The heap is divided into multiple, independently collectible regions. This approach allows the garbage collectors

to focus on collecting those regions that will yield substantial free space, thereby improving the efficiency of the GC process.

ZGC, first introduced as an experimental feature in JDK 11, is a low-latency GC designed for scalability. This state-of-the-art GC ensures that GC pause times are virtually independent of the size of the heap. It achieves this by employing a concurrent copying technique that relocates objects from one region of the memory to another while the application is still running. This concurrent relocation of objects significantly reduces the impact of GC pauses on application performance.

ZGC's design allows Java applications to scale more predictably in terms of memory and latency, making it an ideal choice for applications with large heap sizes and stringent latency requirements. ZGC works alongside the application, operating concurrently to keep pauses to a minimum and maintain a high level of application responsiveness.

In this chapter, we'll delve into the details of these GCs, their characteristics, and their impact on various application types. We'll explore their intricate mechanisms for memory management, the role of thread-local and Non-Uniform Memory Architecture (NUMA)–aware allocations, and ways to tune these collectors for optimal performance. We'll also explore practical strategies for assessing GC performance, from initial measurement to fine-tuning. Understanding this iterative process and recognizing workload patterns will enable developers to select the most suitable GC strategy, ensuring efficient memory management in a wide range of applications.

Thread-Local Allocation Buffers and Promotion-Local Allocation Buffers

Thread-Local Allocation Buffers (TLABs) are an optimization technique employed in the OpenJDK HotSpot VM to improve allocation performance. TLABs reduce synchronization overhead by providing separate memory regions in the heap for individual threads, allowing each thread to allocate memory without the need for locks. This lock-free allocation mechanism is also known as the "fast-path."

In a TLAB-enabled environment, each thread is assigned its buffer within the heap. When a thread needs to allocate a new object, it can simply bump a pointer within its TLAB. This approach makes the allocation process go much faster than the process of acquiring locks and synchronizing with other threads. It is particularly beneficial for multithreaded applications, where contention for a shared memory pool can cause significant performance degradation.

TLABs are resizable and can adapt to the allocation rates of their respective threads. The JVM monitors the allocation behavior of each thread and can adjust the size of the TLABs accordingly. This adaptive resizing helps strike a balance between reducing contention and making efficient use of the heap space.

To fine-tune TLAB performance, the JVM provides several options for configuring TLAB behavior. These options can be used to optimize TLAB usage for specific application workloads and requirements. Some of the key tuning options include

- **`UseTLAB`:** Enables or disables TLABs. By default, TLABs are enabled in the HotSpot VM. To disable TLABs, use `-XX:-UseTLAB`.
- **`-XX:TLABSize=<value>`:** Sets the initial size of TLABs in bytes. Adjusting this value can help balance TLAB size with the allocation rate of individual threads. For example, `-XX:TLABSize=16384` sets the initial TLAB size to 16 KB.
- **`ResizeTLAB`:** Controls whether TLABs are resizable. By default, TLAB resizing is enabled. To disable TLAB resizing, use `-XX:-ResizeTLAB`.
- **`-XX:TLABRefillWasteFraction=<value>`:** Specifies the maximum allowed waste for TLABs, as a fraction of the TLAB size. This value influences the resizing of TLABs, with smaller values leading to more frequent resizing. For example, `-XX:TLABRefillWasteFraction=64` sets the maximum allowed waste to 1/64th of the TLAB size.

By experimenting with these tuning options and monitoring the effects on your application's performance, you can optimize TLAB usage to better suit your application's allocation patterns and demands, ultimately improving its overall performance. To monitor TLAB usage, you can use tools such as Java Flight Recorder (JFR) and JVM logging.

For Java Flight Recorder in JDK 11 and later, you can start recording using this command-line option:

```
-XX:StartFlightRecording=duration=300s,jdk.ObjectAllocationInNewTLAB#enabled=true,
jdk.ObjectAllocationOutsideTLAB#enabled=true,filename=myrecording.jfr
```

This command will record a 300-second profile of your application and save it as `myrecording.jfr`. You can then analyze the recording with tools like JDK Mission Control[1] to visualize TLAB-related statistics.

For JVM logging, you can enable TLAB-related information with the Unified Logging system using the following option:

```
-Xlog:gc*,gc+tlab=debug:file=gc.log:time,tags
```

This configuration logs all GC-related information (`gc*`) along with TLAB information (`gc+tlab=debug`) to a file named `gc.log`. The time and tags decorators are used to include timestamps and tags in the log output. You can then analyze the log file to study the effects of TLAB tuning on your application's performance.

Similar to TLABs, the OpenJDK HotSpot VM employs Promotion-Local Allocation Buffers (PLABs) for GC threads to promote objects. Many GC algorithms perform depth-first copying into these PLABs to enhance co-locality. However, not all promoted objects are copied into PLABs; some objects might be directly promoted to the old generation. Like TLABs, PLABs are automatically resized based on the application's promotion rate.

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

For generational GCs in HotSpot, it is crucial to ensure proper sizing of the generations and allow for appropriate aging of the objects based on the application's allocation rate. This approach ensures that mostly (and possibly only) long-lived static and transient objects are tenured ("promoted"). Occasionally, spikes in the allocation rate may unnecessarily increase the promotion rate, which could trigger an old-generation collection. A properly tuned GC will have enough headroom that such spikes can be easily accommodated without triggering a full stop-the-world (STW) GC event, which is a fail-safe or fallback collection for some of the newer GCs.

To tune PLABs, you can use the following options, although these options are less frequently used because PLABs are usually self-tuned by the JVM:

- **`-XX:GCTimeRatio=<value>`:** Adjusts the ratio of GC time to application time. Lower values result in larger PLABs, potentially reducing GC time at the cost of increased heap usage.
- **`ResizePLAB`:** Enables or disables the adaptive resizing of PLABs. By default, it is enabled.
- **`-XX:PLABWeight=<value>`:** Sets the percentage of the current PLAB size added to the average PLAB size when computing the new PLAB size. The default value is 75.

PLABs can help improve GC performance by reducing contention and providing a faster promotion path for objects. Properly sized PLABs can minimize the amount of memory fragmentation and allow for more efficient memory utilization. However, just as with TLABs, it is essential to strike a balance between reducing contention and optimizing memory usage. If you suspect that PLAB sizing might be an issue in your application, you might consider adjusting the `-XX:PLABWeight` and `-XX:ResizePLAB` options, but be careful and always test their impact on application performance before applying changes in a production environment.

Optimizing Memory Access with NUMA-Aware Garbage Collection

In OpenJDK HotSpot, there is an architecture-specific allocator designed for Non-Uniform Memory Access (NUMA) systems called "NUMA-aware." As discussed in Chapter 5, "End-to-End Java Performance Optimization: Engineering Techniques and Micro-benchmarking with JMH," NUMA is a type of computer memory design that is utilized in multiprocessing. The speed of memory access is dependent on the location of the memory in relation to the processor. On these systems, the operating system allocates physical memory based on the "first-touch" principle, which states that memory is allocated closest to the processor that first accesses it. Hence, to ensure allocations occur in the memory area nearest to the allocating thread, the eden space is divided into segments for each node (where a node is a combination of a CPU, memory controller, and memory bank). Using a NUMA-aware allocator can help improve performance and reduce memory access latency in multiprocessor systems.

To better understand how this works, let's take a look at an example. In Figure 6.1, you can see four nodes labeled Node 0 to Node 3. Each node has its own processing unit and a memory controller plus memory bank combination. A process/thread that is running on Node 0 but

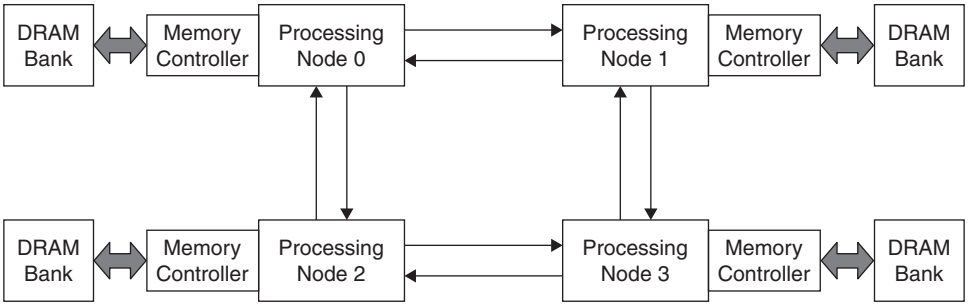

Figure 6.1 NUMA Nodes with Processors, Memory Controllers, and DRAM Banks

wants to access memory on Node 3 will have to reach Node 3 either via Node 2 or via Node 1. This is considered two hops. If a process/thread on Node 0 wants to access Node 3 or Node 2's memory bank, then it's just one hop away. A local memory access needs no hops. An architecture like that depicted in Figure 6.1 is called a NUMA.

If we were to divide the eden space of the Java heap so that we have areas for each node and also have threads running on a particular node allocate out of those node-allocated areas, then we would be able to have hop-free access to node-local memory (provided the scheduler doesn't move the thread to a different node). This is the principle behind the NUMA-aware allocator—basically, the TLABs are allocated from the closest memory areas, which allows for faster allocations.

In the initial stages of an object's life cycle, it resides in the eden space allocated specifically from its NUMA node. Once a few of the objects have survived or when the objects need to be promoted to the old generation, they might be shared between threads across different nodes. At this point, allocation occurs in a node-interleaved manner, meaning memory chunks are sequentially and evenly distributed across nodes, from Node 0 to the highest-numbered node, until all of the memory space has been allocated.

Figure 6.2 illustrates a NUMA-aware eden. Thread 0 and Thread 1 allocate memory out of the eden area for Node 0 because they are both running on Node 0. Meanwhile, Thread 2 allocates out of the eden area for Node 1 because it's running on Node 1.

To enable NUMA-aware GC in JDK 8 and later, use the following command-line option:

```
-XX:+UseNUMA
```

This option enables NUMA-aware memory allocations, allowing the JVM to optimize memory access for NUMA architectures.

To check if your JVM is running with NUMA-aware GC, you can use the following command-line option:

```
-XX:+PrintFlagsFinal
```

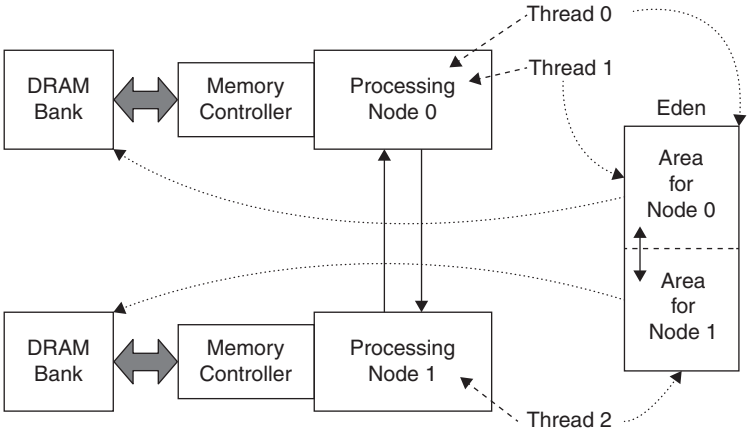

Figure 6.2 The Inner Workings of a NUMA-Aware GC

This option outputs the final values of the JVM flags, including the `UseNUMA` flag, which indicates if NUMA-aware GC is enabled.

In summary, the NUMA-aware allocator is designed to optimize memory access for NUMA architectures by dividing the eden space of the Java heap into areas for each node and creating TLABs from the closest memory areas. This can result in faster allocations and improved performance. However, when enabling NUMA-aware GC, it's crucial to monitor your application's performance, as the impacts might differ based on your system's hardware and software configurations.

Some garbage collectors, such as G1 and the Parallel GC, benefit from NUMA optimizations. G1 was made NUMA-aware in JDK 14,[2] enhancing its efficiency in managing large heaps on NUMA architectures. ZGC is another modern garbage collector designed to address the challenges of large heap sizes, low-latency requirements, and better overall performance. ZGC became NUMA-aware in JDK 15.[3] In the following sections, we will delve into the key features and enhancements of G1 and ZGC, as well as their performance implications and tuning options in the context of modern Java applications.

Exploring Garbage Collection Improvements

As Java applications continue to evolve in complexity, the need for efficient memory management and GC becomes increasingly important. Two significant garbage collectors in the OpenJDK HotSpot VM, G1 and ZGC, aim to improve garbage collection performance, scalability, and predictability, particularly for applications with substantial heaps sizes and stringent low-latency demands.

[2] https://openjdk.org/jeps/345

[3] https://wiki.openjdk.org/display/zgc/Main#Main-EnablingNUMASupport

This section highlights the salient features of each garbage collector, listing the key advancements and optimizations made from JDK 11 to JDK 17. We will explore G1's enhancements in mixed collection performance, pause-time predictability, and its ability to better adapt to different workloads. As ZGC is a newer garbage collector, we will also dive deeper into its unique capabilities, such as concurrent garbage collection, reduced pause times, and its suitability for large heap sizes. By understanding these advances in GC technology, you will gain valuable insights into their impact on application performance and memory management, and the trade-offs to consider when choosing a garbage collector for your specific use case.

G1 Garbage Collector: A Deep Dive into Advanced Heap Management

Over the years, there has been a significant shift in the design of GCs. Initially, the focus was on throughput-oriented GCs, which aimed to maximize the overall efficiency of memory management processes. However, with the increasing complexity of applications and heightened user expectations, the emphasis has shifted towards latency-oriented GCs. These GCs aim to minimize the pause times that can disrupt application performance, even if it means a slight reduction in overall throughput. G1, which is the default GC for JDK 11 LTS and beyond, is a prime example of this shift toward latency-oriented GCs. In the subsequent sections, we'll examine the intricacies of the G1 GC, its heap management, and its advantages.

Regionalized Heap

G1 divides the heap into multiple equal-size regions, allowing each region to play the role of eden, survivor, or old space dynamically. This flexibility aids in efficient memory management.

Adaptive Sizing

- **Generational sizing:** G1's generational framework is complemented by its adaptive sizing capability, allowing it to adjust the sizes of the young and old generations based on runtime behavior. This ensures optimal heap utilization.
- **Collection set (CSet) sizing:** G1 dynamically selects a set of regions for collection to meet the pause-time target. The CSet is chosen based on the amount of garbage those regions contain, ensuring efficient space reclamation.
- **Remembered set (RSet) coarsening:** G1 maintains remembered sets to track cross-region references. Over time, if these RSets grow too large, G1 can coarsen them, reducing their granularity and overhead.

Pause-Time Predictability

- **Incremental collection:** G1 breaks down garbage collection work into smaller chunks, processing these chunks incrementally to ensure that pause time remains predictable.
- **Concurrent marking enhancement:** Although CMS introduced concurrent marking, G1 enhances this phase with advanced techniques like snapshot-at-the-beginning (SATB)

marking.[4] G1 employs a multiphase approach that is intricately linked with its region-based heap management, ensuring minimal disruptions and efficient space reclamation.

- **Predictions in G1:** G1 uses a sophisticated set of predictive mechanisms to optimize its performance. These predictions, grounded in G1's history-based analysis, allow it to adapt dynamically to the application's behavior and optimize performance:
  - **Single region copy time:** G1 forecasts the time it will take to copy/evacuate the contents of a single region by drawing on historical data for past evacuations. This prediction is pivotal in estimating the total pause time of a GC cycle and in dividing the number of regions to include in a mixed collection to adhere to the desired pause time.
  - **Concurrent marking time:** G1 anticipates the time required to complete the concurrent marking phase. This foresight is critical in timing the initiation of the marking phase by dynamically adjusting the IHOP—the initiating heap occupancy percent (`-XX:InitiatingHeapOccupancyPercent`). The adjustment of IHOP is based on real-time analysis of the old generation's occupancy and the application's memory allocation behavior.
  - **Mixed garbage collection time:** G1 predicts the time for mixed garbage collections to strategically select a set of old regions to collect along with the young generation. This estimation helps G1 in maintaining a balance between reclaiming sufficient memory space and adhering to pause-time targets.
  - **Young garbage collection time:** G1 calculates the time required for young-only collections to guide the resizing of the young generation. G1 adjusts the size of the young generation based on these predictions to optimize pause times based on the desired target (`-XX:MaxGCPauseTimeMillis`) and adapt to the application's changing allocation patterns.
  - **Amount of reclaimable space:** G1 estimates the space that will be reclaimed by collecting specific old regions. This estimate informs the decision-making process for including old regions in mixed collections, optimizing memory utilization, and minimizing the risk of heap fragmentation.

Core Scaling

G1's design inherently supports multithreading, allowing it to scale with the number of available cores. This has the following implications:

- **Parallelized young-generation collections:** Ensuring quick reclamation of short-lived objects.
- **Parallelized old-generation collections:** Enhancing the efficiency of space reclamation in the old generation.
- **Concurrent threads during the marking phase:** Speeding up the identification of live objects.

[4] Charlie Hunt, Monica Beckwith, Poonam Parhar, and Bengt Rutisson. *Java Performance Companion*. Boston, MA: Addison-Wesley Professional, 2016.

Pioneering Features

Beyond the regionalized heap and adaptive sizing, G1 introduced several other features:

- **Humongous objects handling:** G1 has specialized handling for large objects that span multiple regions, ensuring they don't cause fragmentation or extended pause times.
- **Adaptive threshold tuning (IHOP):** As mentioned earlier, G1 dynamically adjusts the initiation threshold of the concurrent marking phase based on runtime metrics. This tuning is particularly focused on IHOP's dynamic adjustment to determine the right time to start the concurrent marking phase.

In essence, G1 is a culmination of innovative techniques and adaptive strategies, ensuring efficient memory management while providing predictable responsiveness.

Advantages of the Regionalized Heap

In G1, the division of the heap into regions creates a more flexible unit of collection compared to traditional generational approaches. These regions serve as G1's unit of work (UoW), enabling its predictive logic to efficiently tailor the collection set for each GC cycle by adding or removing regions as needed. By adjusting which regions are included in the collection set, G1 can respond dynamically to the application's current memory usage patterns.

A regionalized heap facilitates incremental compaction, enabling the collection set to include all young regions and at least one old region. This is a significant improvement over the monolithic old-generation collection, such as full GC in Parallel GC or the fallback full GC for CMS, which are less adaptable and efficient.

Traditional Versus Regionalized Java Heap Layouts

To better understand the benefits of a regionalized heap, let's compare it with a traditional Java heap layout. In a traditional Java heap (shown in Figure 6.3), there are separate areas designated for young and old generations. In contrast, a regionalized Java heap layout (shown in Figure 6.4) divides the heap into regions, which can be classified as either free or occupied. When a young region is released back to the free list, it can be reclaimed as an old region based on the promotion needs. This noncontiguous generation structure, coupled with the flexibility to resize the generations as needed, allows G1's predictive logic to achieve its service level objective (SLO) for responsiveness.

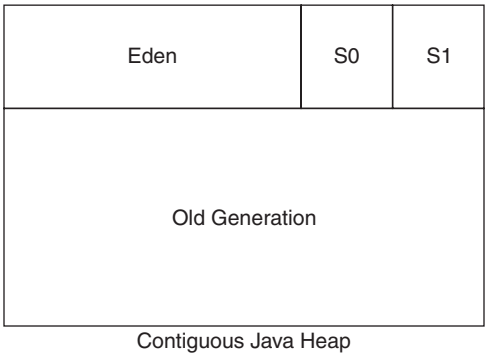

Figure 6.3 A Traditional Java Heap Layout

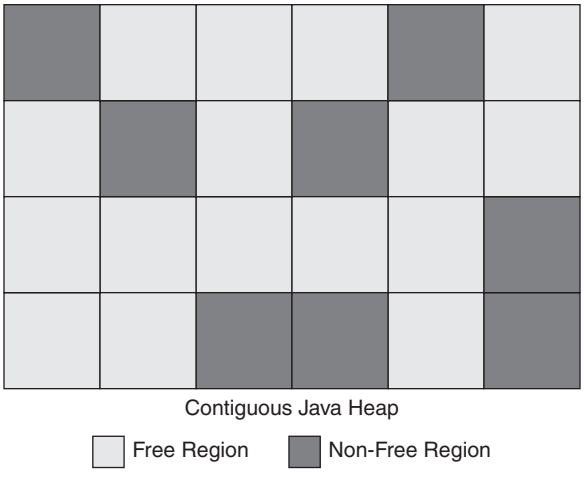

Figure 6.4 A Regionalized Java Heap Layout

Key Aspects of the Regionalized Java Heap Layout

One important aspect of G1 is the introduction of humongous objects. An object is considered humongous if it spans 50% or more of a single G1 region (Figure 6.5). G1 handles the allocation of humongous objects differently than the allocation of regular objects. Instead of following the fast-path (TLABs) allocation, humongous objects are allocated directly out of the old generation into designated humongous regions. If a humongous object spans more than one region size, it will require contiguous regions. (For more about humongous regions and objects, please refer to the *Java Performance Companion* book.[5])

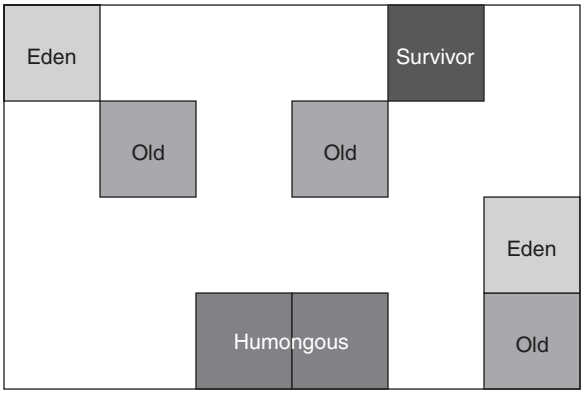

Figure 6.5 A Regionalized Heap Showing Old, Humongous, and Young Regions

[5] Charlie Hunt, Monica Beckwith, Poonam Parhar, and Bengt Rutisson. *Java Performance Companion*. Boston, MA: Addison-Wesley Professional, 2016: chapters 2 and 3.

Balancing Scalability and Responsiveness in a Regionalized Heap

A regionalized heap aims to enhance scalability while maintaining the target responsiveness and pause time. However, the command-line option `–XX:MaxGCPauseMillis` provides only soft-real time goals for G1. Although the prediction logic strives to meet the required goal, the collection pause doesn't have a hard-stop when the pause-time goal is reached. If your application experiences high GC tail latencies due to allocation (rate) spikes or increased mutation rates, and the prediction logic struggles to keep up, you may not consistently meet your responsiveness SLOs. In such cases, manual GC tuning (covered in the next section) may be necessary.

NOTE The mutation rate is the rate at which your application is updating references.

NOTE Tail latency refers to the slowest response times experienced by the end users. These latencies can significantly impact the user experience in latency-sensitive applications and are usually represented by higher percentiles (for example, 4-9s, 5-9s percentiles) of the latency distribution.

Optimizing G1 Parameters for Peak Performance

Under most circumstances, G1's automatic tuning and prediction logic work effectively when applications follow a predictable pattern, encompassing a warm-up (or ramp-up) phase, a steady-state, and a cool-down (or ramp-down) phase. However, challenges may arise when multiple spikes in the allocation rate occur, especially when these spikes result from bursty, transient, humongous objects allocations. Such situations can lead to premature promotions in the old generation and necessitate accommodating transient humongous objects/regions in the old generation. These factors can disrupt the prediction logic and place undue pressure on the marking and mixed collection cycles. Before recovery can occur, the application may encounter evacuation failures due to insufficient space for promotions or a lack of contiguous regions for humongous objects as a result of fragmentation. In these cases, intervention whether manual or via automated/AI systems becomes necessary.

G1's regionalized heap, collection set, and incremental collections offer numerous tuning knobs for adjusting internal thresholds, and they support targeted tuning when needed. To make well-informed decisions about modifying these parameters, it's essential to understand these tuning knobs and their potential impact on G1's ergonomics.

Optimizing GC Pause Responsiveness in JDK 11 and Beyond

To effectively tune G1, analyzing the pause histogram obtained from a GC log file can provide valuable insights into your application's performance. Consider the pause-time series plot from a JDK 11 GC log file, as shown in Figure 6.6. This plot reveals the following GC pauses:

- 4 young GC pauses followed by ...
- 1 initial mark pause when the IHOP threshold is crossed
- Next, 1 young GC pause during the concurrent mark phase ...
- And then 1 remark pause and 1 cleanup pause
- Finally, another young GC pause before 7 mixed GCs to incrementally reclaim the old-generation space

Assuming the system has an SLO requiring 100% of GC pauses to be less than or equal to 100 ms, it is evident that the last mixed collection did not meet this requirement. Such one-off pauses can potentially be mitigated by the abortable mixed collection feature introduced in JDK 12—which we will discuss in a bit. However, in cases where automated optimizations don't suffice or aren't applicable, manual tuning becomes necessary (Figure 6.7).

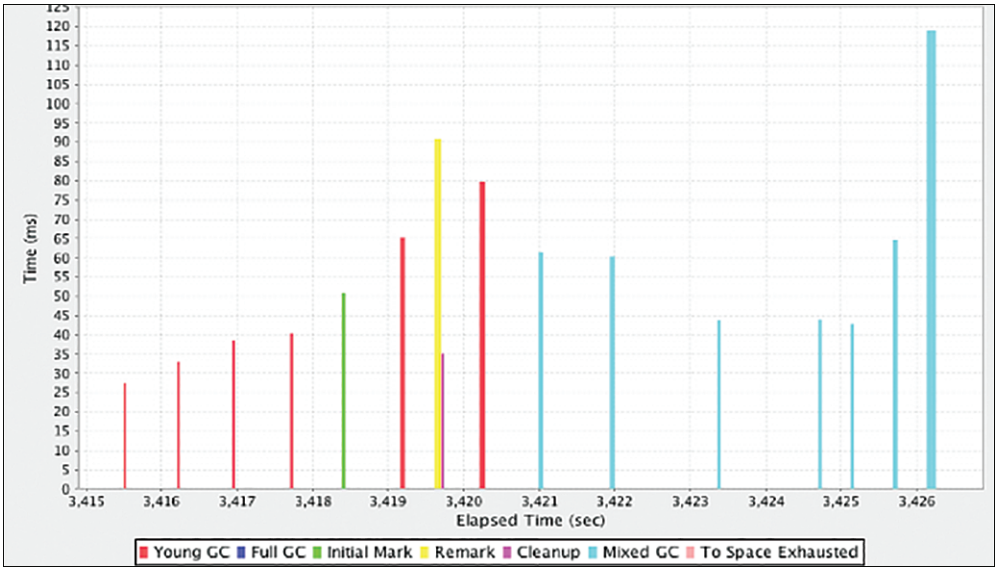

Figure 6.6 G1 Garage Collector's Pause-Time Histogram

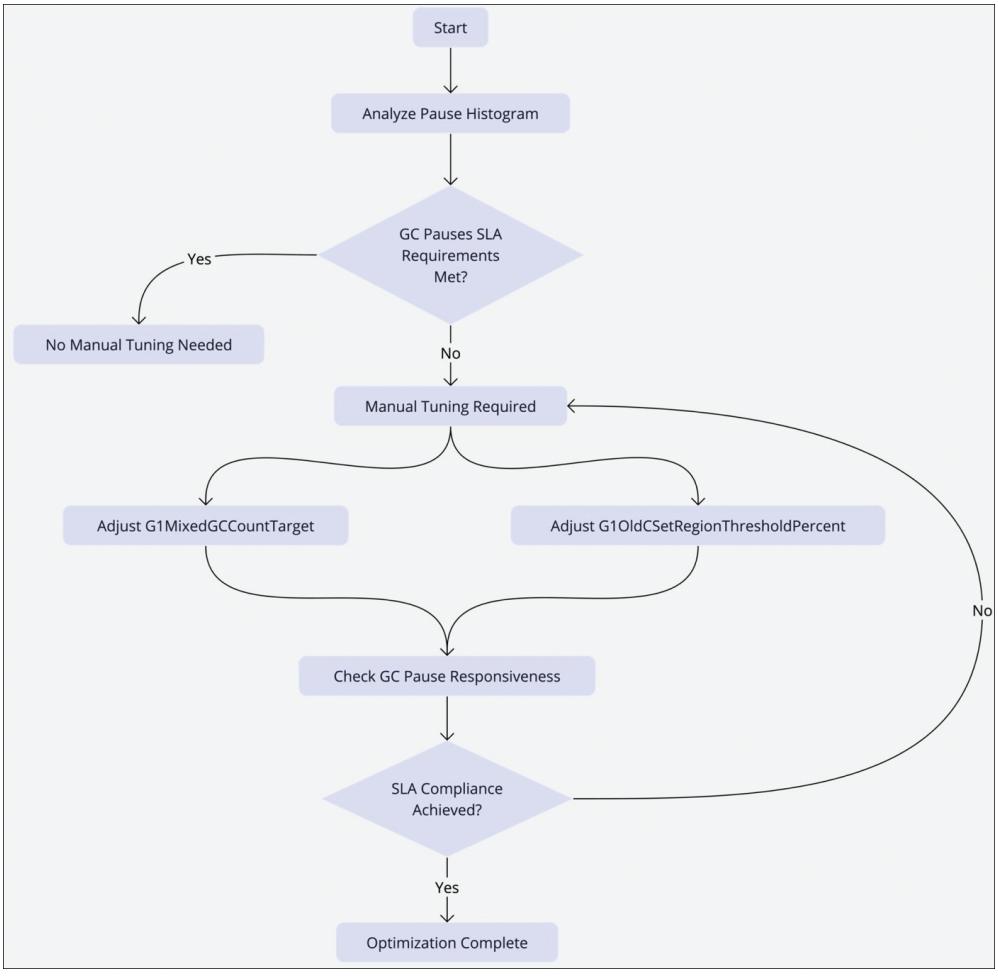

Figure 6.7 Optimizing G1 Responsiveness

Two thresholds can help control the number of old regions included in mixed collections:

- Maximum number of old regions to be included in the mixed collection set:`-XX:G1OldCSetRegionThresholdPercent=<p>`
- Minimum number of old regions to be included in the mixed collection set: `-XX:G1MixedGCCountTarget=<n>`

Both tunables help determine how many old regions are added to the young regions, forming the mixed collection set. Adjusting these values can improve pause times, ensuring SLO compliance.

In our example, as shown in the timeline of pause events (Figure 6.6), most young collections and mixed collections are below the SLO requirements except for the last mixed collection. A simple solution is to reduce the maximum number of old regions included in the mixed collection set by using the following command-line option:

```
-XX:G1OldCSetRegionThresholdPercent=<p>
```

where p is a percentage of the total Java heap. For example, with a heap size of 1 GB and a default value of 10% (i.e., `–Xmx1G – Xms1G -XX:G1OldCSetRegionThresholdPercent=10`), the count of old regions added to the collection set would be 10% of 1 GB (which is rounded up), so we get a count of 103. By reducing the percentage to 7%, the maximum count drops to 72, which would be enough to bring the last mixed-collection pause within the acceptable GC response time SLO. However, this change alone may increase the total number of mixed collections, which might be a concern.

G1 Responsiveness: Enhancements in Action

With advancements in JDK versions, several new features and enhancements have been introduced to G1. These features, when utilized correctly, can significantly improve the responsiveness and performance of applications without the need for extensive tuning. Although the official documentation provides a comprehensive overview of these features, there's immense value in understanding their practical applications. Here's how field experiences translate these enhancements into real-world performance gains:

- **Parallelizing reference processing (JDK 11):** Consider an e-commerce platform using `SoftReference` objects to cache product images. During high-traffic sales events, the number of references skyrockets. The parallel reference processing capability keeps the system agile under heavy loads, maintaining a smooth user experience even as traffic spikes.
- **Abortable mixed collections (JDK 12):** Picture a real-time multiplayer gaming server during intense gameplay, serving thousands of concurrent online players. This server can have medium-lived (transient) data (player, game objects). Here, real-time constraints with low tail latencies are the dominant SLO requirement. The introduction of abortable mixed collections ensures that gameplay stays fluid, even as in-game actions generate transient data at a frenzied pace.
- **Promptly returning unused committed memory (JDK 12):** A cloud-based video rendering service, which performs postproduction of 3D scenes, often sits on idle resources. G1 can return unused committed memory to the operating system more quickly based on the system load. With this ability, the service can optimize resource utilization, thereby ensuring that other applications on the same server have access to memory when needed, and in turn leading to cost savings in cloud hosting fees. To further fine-tune this behavior, two G1 tuning options can be utilized:
  - **`G1PeriodicGCInterval`**: This option specifies the interval (in milliseconds) at which periodic GC cycles are triggered. By setting this option, you can control how frequently G1 checks and potentially returns unused memory to the OS. The default is 0, which means that periodic GC cycles are disabled by default.

- **`G1PeriodicGCSystemLoadThreshold`:** This option sets a system load threshold. If the system load is below this threshold, a periodic GC cycle is triggered even if the `G1PeriodicGCInterval` has not been reached. This option can be particularly useful in scenarios where the system is not under heavy load, and it's beneficial to return memory more aggressively. The default is 0, meaning that the system load threshold check is disabled.

CAUTION Although these tuning options provide greater control over memory reclamation, it's essential to exercise caution. Overly aggressive settings might lead to frequent and unnecessary GC cycles, which could negatively impact the application's performance. Always monitor the system's behavior after making changes and adjust it accordingly to strike a balance between memory reclamation and application performance.

- **Improved concurrent marking termination (JDK 13):** Imagine a global financial analytics platform that processes billions of market signals for real-time trading insights. The improved concurrent marking termination in JDK 13 can significantly shrink GC pause times, ensuring that high-volume data analyses and reporting are not interrupted, and thereby providing traders with consistent, timely market intelligence.
- **G1 NUMA-awareness (JDK 14):** Envision a high-performance computing (HPC) environment tasked with running complex simulations. With G1's NUMA-awareness, the JVM is optimized for multi-socket systems, leading to notable improvements in GC pause times and overall application performance. For HPC workloads, where every millisecond counts, this means faster computation and more efficient use of hardware resources.
- **Improved concurrent refinement (JDK 15):** Consider a scenario in which an enterprise search engine must handle millions of queries across vast indexes. The improved concurrent refinement in JDK 15 reduces the number of log buffer entries that will be made, which translates into shorter GC pauses. This refinement allows for rapid query processing, which is critical for maintaining the responsiveness users expect from a modern search engine.
- **Improved heap management (JDK 17):** Consider the case of a large-scale Internet of Things (IoT) platform that aggregates data from millions of devices. JDK 17's enhanced heap management within G1 can lead to more efficient distribution of heap regions, which minimizes GC pause times and boosts application throughput. For an IoT ecosystem, this translates into faster data ingestion and processing, enabling real-time responses to critical sensor data.

Building on these enhancements, G1 also includes other optimizations that further enhance responsiveness and efficiency:

- **Optimizing evacuation:** This optimization streamlines the evacuation process, potentially reducing GC pause times. In transactional applications where quick response

times are critical, such as financial processing or real-time data analytics, optimizing evacuation can ensure faster transaction processing. This results in a system that remains responsive even under high load conditions.

- **Building remembered sets on the fly:** By constructing remembered sets during the marking cycle, G1 minimizes the time spent in the remark phase. This can be particularly beneficial in systems with massive data sets, such as a big data analytics platform, where quicker remark phases can lead to reduced processing times.
- **Improvements in remembered sets scanning:** These improvements facilitate shorter GC pause times—an essential consideration for applications such as live-streaming services, where uninterrupted data flow is critical.
- **Reduce barrier overhead:** Lowering the overhead associated with barriers can lead to shorter GC pause times. This enhancement is valuable in cloud-based software as a service (SaaS) applications, where efficiency directly translates into reduced operational costs.
- **Adaptive IHOP:** This feature allows G1 to adaptively adjust the IHOP value based on the current application behavior. For example, in a dynamic content delivery network, adaptive IHOP can improve responsiveness by efficiently managing memory under varying traffic loads.

By understanding these enhancements and the available tuning options, you can better optimize G1 to meet the responsiveness requirements for your specific application. Implementing these improvements thoughtfully can lead to substantial performance gains, ensuring that your Java applications remain efficient and responsive under a wide range of conditions.

Optimizing GC Pause Frequency and Overhead with JDK 11 and Beyond

Meeting stringent SLOs that require specific throughput goals and minimal GC overhead is critical for many applications. For example, an SLO might dictate that GC overhead must not exceed 5% of the total execution time. Analysis of pause-time histograms may reveal that an application is struggling to meet the SLO's throughput requirement due to frequent mixed GC pauses, resulting in excessive GC overhead. Addressing this challenge involves strategically tuning of G1 to improve its efficiency. Here's how:

- **Strategically eliminating expensive regions:** Tailor the mixed collection set by omitting costly old regions. This action, while accepting some uncollected garbage, or "heap waste," ensures the heap accommodates the live data, transient objects, humongous objects, and additional space for waste. Increasing the total Java heap size might be necessary to achieve this goal.
- **Setting live data thresholds:** Utilize `-XX:G1MixedGCLiveThresholdPercent=<p>` to determine a cutoff for each old region based on the percentage of live data in that region. Regions exceeding this threshold are not included in the mixed collection set to avoid costly collection cycles.
- **Heap waste management:** Apply `-XX:G1HeapWastePercent=<p>` to permit a certain percentage of the total heap to remain uncollected (that is, "wasted"). This tactic targets the costly regions at the end of the sorted array established during the marking phase.

Figure 6.8 presents an overview of this process.

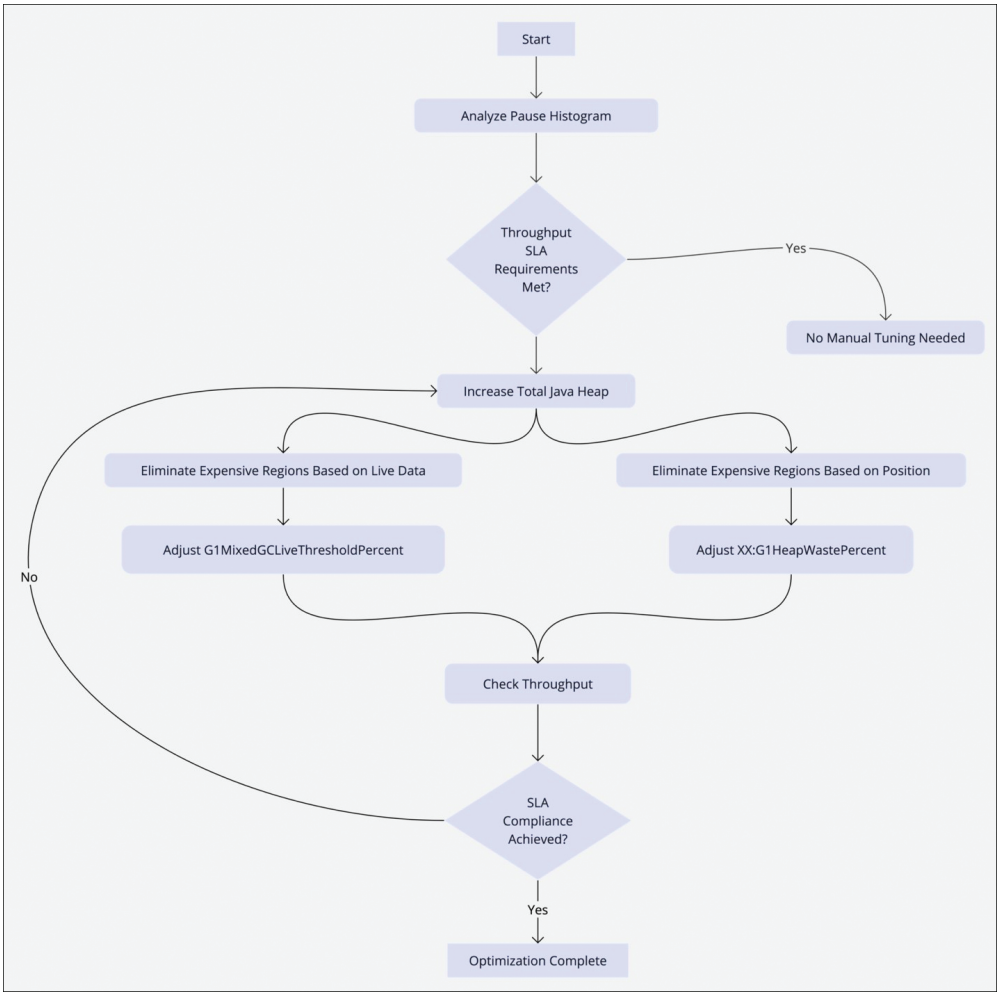

Figure 6.8 Optimizing G1's Throughput

G1 Throughput Optimizations: Real-World Applications

G1 has undergone a series of improvements and optimizations since Java 11, including the introduction of adaptive sizing and thresholds, that can help optimize its use to meet your specific application's throughput needs. These enhancements may reduce the need for fine-tuning for overhead reduction because they can lead to more efficient memory usage, thereby improving application performance. Let's look at these updates:

- **Eager reclaim of humongous objects (JDK 11):** In a social-media analytics application, large data sets representing user interactions are processed quite frequently. The capability to eagerly reclaim humongous objects prevents heap exhaustion and keeps analysis workflows running smoothly, especially during high-traffic events such as viral marketing campaigns.

- **Improved `G1MixedGCCountTarget` and abortable mixed collections (JDK 12):** Consider a high-volume transaction system in a stock trading application. By fine-tuning the number of mixed GCs and allowing abortable collections, G1 helps maintain consistent throughput even during market surges, ensuring traders experience minimal latency.
- **Adaptive heap sizing (JDK 15):** Cloud-based services, such as a document storage and collaboration platform, experience variable loads throughout the day. Adaptive heap sizing ensures that during periods of low activity, the system minimizes resource usage, reducing operational costs while still delivering high throughput during peak usage.
- **Concurrent humongous object allocation and young generation sizing improvements (JDK 17):** In a large-scale IoT data-processing service, concurrent allocation of large objects allows for more efficient handling of incoming sensor data streams. Improved young-generation sizing dynamically adjusts to the workload, preventing potential bottlenecks and ensuring steady throughput for real-time analytics.

In addition to these enhancements, the following optimizations contribute to improving the G1 GC's throughput:

- **Remembered sets space reductions:** In database management systems, reducing the space for remembered sets means more memory is available for caching query results, leading to faster response times and higher throughput for database operations.
- **Lazy threads and remembered sets initialization:** For an on-demand video streaming service, lazy initialization allows for quick scaling of resources in response to user demand, minimizing start-up delays and improving viewer satisfaction.
- **Uncommit at remark:** Cloud platforms can benefit from this feature by dynamically scaling down resource allocations during off-peak hours, optimizing cost-effectiveness without sacrificing throughput.
- **Old generation on NVDIMM:** By placing the old generation on nonvolatile DIMM (NVDIMM), G1 can achieve faster access times and improved performance. This can lead to more efficient use of the available memory and thus increasing throughput.
- **Dynamic GC threads with a smaller heap size per thread:** Microservices running in a containerized environment can utilize dynamic GC threads to optimize resource usage. With a smaller heap size per thread, the GC can adapt to the specific needs of each microservice, leading to more efficient memory management and improved service response times.
- **Periodic GC:** For IoT platforms that process continuous data streams, periodic garbage collection can help maintain a consistent application state by periodically cleaning up unused objects. This strategy can reduce the latency associated with garbage collection cycles, ensuring timely data processing and decision making for time-sensitive IoT operations.

By understanding these enhancements and the available tuning options, you can better optimize G1 to meet your specific application's throughput requirements.

Tuning the Marking Threshold

As mentioned earlier, Java 9 introduced an adaptive marking threshold, known as the IHOP. This feature should benefit most applications. However, certain pathological cases, such as bursty allocations of short-lived (possibly humongous) objects, often seen when building a short-lived transactional cache, could lead to suboptimal behavior, In such scenarios, a low adaptive IHOP might result in a premature concurrent marking phase, which completes without triggering the needed mixed collections—especially if the bursty allocations occur afterward. Alternatively, a high adaptive IHOP that doesn't decrease quickly enough could delay the concurrent marking phase initiation, leading to regular evacuation failures.

To address these issues, you can disable the adaptive marking threshold and set it to a steady value by adding the following option to your command line:

```
-XX:-G1UseAdaptiveIHOP -XX:InitiatingHeapOccupancyPercent=<p>
```

where p is the occupancy percentage of the total heap at which you would like the marking cycle to start.

In Figure 6.9, after the mixed GC pause at approximately 486.4 seconds, there is a young GC pause followed by the start of a marking cycle (the line after the young GC pause is the initiating mark pause). After the initial mark pause, we observe 23 young GC pauses, a "to"-space exhausted pause, and finally a full GC pause. This pattern, in which a mixed collection cycle has just been completed and a new concurrent marking cycle begins but is unable to finish while the size of the young generation is being reduced (hence the young collections), indicates issues with the adaptive marking threshold. The threshold may be too high, delaying the marking cycle to a point where back-to-back concurrent marking and mixed collection cycles are needed, or the concurrent cycle may be taking too long to complete due to a low number of concurrent marking threads that can't keep up with the workload.

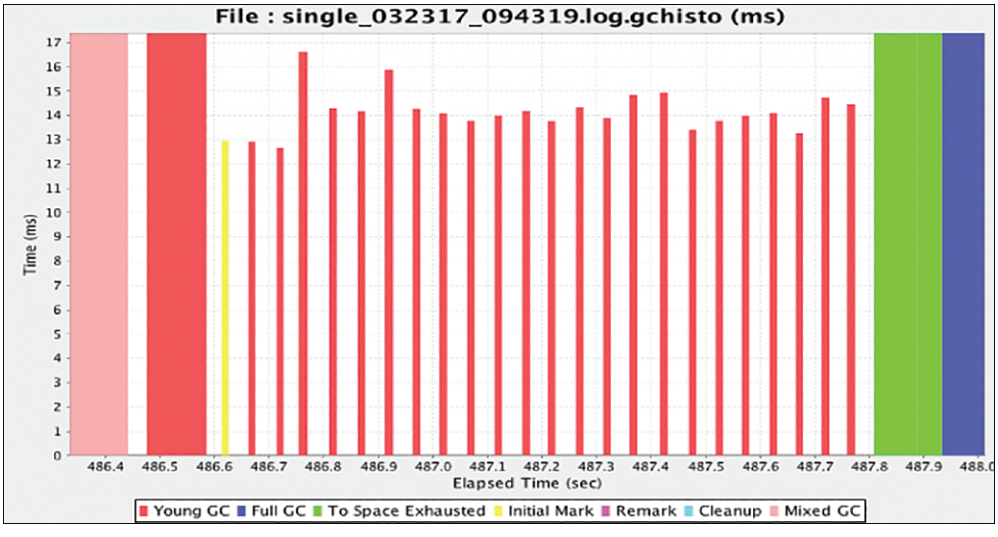

Figure 6.9 A G1 Garbage Collector Pause Timeline Showing Incomplete Marking Cycle

In such cases, you can stabilize the IHOP value to a lower level to accommodate the static and transient live data set size (LDS) (and humongous objects) and increase your concurrent threads count by setting `n` to a higher value: `-XX:ConcGCThreads=<n>`. The default value for `n` is one-fourth of your GC threads available for parallel (STW) GC activity.

G1 represents an important milestone in the evolution of GCs in the Java ecosystem. It introduced a regionalized heap and enabled users to specify realistic pause-time goals and maximize desired heap sizes. G1 predicts the total number of regions per collection (young and mixed) to meet the target pause time. However, the actual pause time might still vary due to factors such as the density of data structures, the popularity of regions or objects in the collected regions, and the amount of live data per region. This variability can make it challenging for applications to meet their already defined SLOs and creates a demand for a near-real-time, predictable, scalable, and low-latency garbage collector.

Z Garbage Collector: A Scalable, Low-Latency GC for Multi-terabyte Heaps

To meet the need for real-time responsiveness in applications, the Z Garbage Collector (ZGC) was introduced, advancing OpenJDK's garbage collection capabilities. ZGC incorporates innovative concepts into the OpenJDK universe—some of which have precedents in other collectors like Azul's C4 collector.[6] Among ZGC's pioneering features are concurrent compaction, colored pointers, off-heap forwarding tables, and load barriers, which collectively enhance performance and predictability.

ZGC was first introduced as an experimental collector in JDK 11 and reached production readiness with JDK 15. In JDK 16, concurrent thread stack processing was enabled, allowing ZGC to guarantee that pause times would remain below the millisecond threshold, independent of LDS and heap size. This advancement enables ZGC to support a wide spectrum of heap sizes, ranging from as little as 8 MB to as much as 16 TB, allowing it to cater to a diverse array of applications from lightweight microservices to memory-intensive domains like big data analytics, machine learning, and graphics rendering.

In this section, we'll take a look at some of the core concepts of ZGC that have enabled this collector to achieve ultra-low pauses and high predictability.

ZGC Core Concepts: Colored Pointers

One of the key innovations of ZGC is its use of colored pointers, a form of metadata tagging that allows the garbage collector to store additional information directly within the pointers. This technique, known as virtual address mapping or tagging, is achieved through multi-mapping on the x86-64 and aarch64 architectures. The colored pointers can indicate the state of an object, such as whether it is known to be marked, whether it is pointing into the relocation set, or whether it is reachable only through a *Finalizer* (Figure 6.10).

[6] www.azul.com/c4-white-paper/

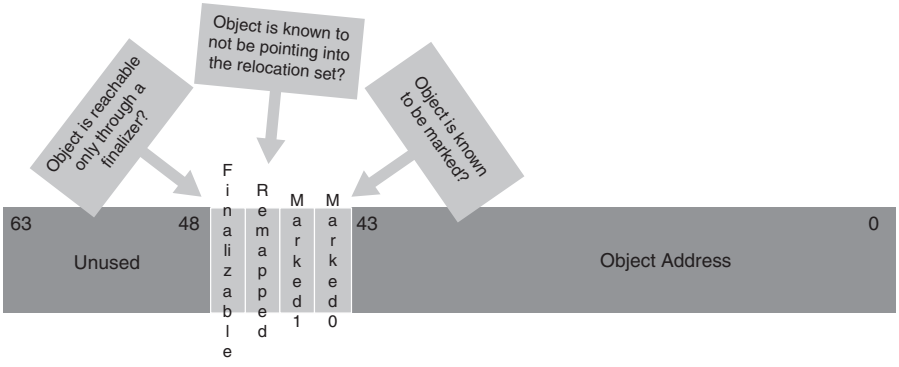

Figure 6.10 Colored Pointers

These tagged pointers allow ZGC to efficiently handle complex tasks such as concurrent marking and relocation without needing to pause the application. This leads to overall shorter GC pause times and improved application performance, making ZGC an effective solution for applications that require low latency and high throughput.

ZGC Core Concepts: Utilizing Thread-Local Handshakes for Improved Efficiency

ZGC is notable for its innovative approach to garbage collection, especially in how it utilizes thread-local handshakes.[7] Thread-local handshakes enable the JVM to execute actions on individual threads without requiring a global STW pause. This approach reduces the impact of STW events and contributes to ZGC's ability to deliver low pause times—a crucial consideration for latency-sensitive applications.

ZGC's adoption of thread-local handshakes is a significant departure from the traditional garbage collection STW techniques (the two are compared in Figure 6.11). In essence, they enable ZGC to adapt and scale the collections, thereby making it suitable for applications with larger heap sizes and stringent latency requirements. However, while ZGC's concurrent processing approach optimizes for lower latency, it could have implications for an application's maximum throughput. ZGC is designed to balance these aspects, ensuring that the throughput degradation remains within acceptable limits.

ZGC Core Concepts: Phases and Concurrent Activities

Like other OpenJDK garbage collectors, ZGC is a tracing collector, but it stands out because of its ability to perform both concurrent marking and compaction. It addresses the challenge of moving live objects from the "from" space to the "to" space within a running application through a set of specialized phases, innovative optimization techniques, and a unique approach to garbage collection.

ZGC refines the concept of a regionalized heap, pioneered by G1, with the introduction of *ZPages*—dynamically sized regions that are allocated and managed according to their memory usage patterns. This flexibility in region sizing is part of ZGC's strategy to optimize memory management and accommodate various allocation rates efficiently.

[7] https://openjdk.org/jeps/312

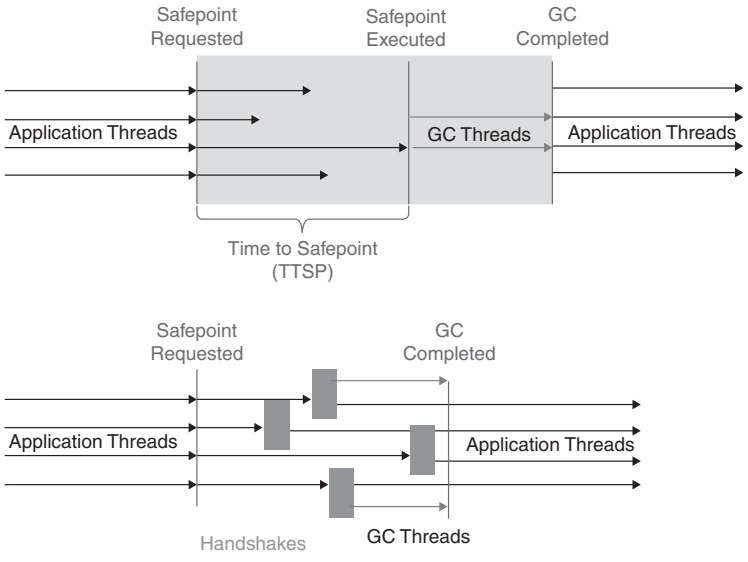

Figure 6.11 Thread-Local Handshakes Versus Global Stop-the-World Events

ZGC also improves on the concurrent marking phase by employing the unique approach of dividing the heap into logical stripes. Each GC thread works on its own stripe, thus reducing contention and synchronization overhead. This design allows ZGC to perform concurrent, parallel processing while marking, reducing phase times and improving overall application performance.

Figure 6.12 provides a visual representation of the ZGC's heap organization into logical stripes. The stripes are shown as separate segments within the heap space, numbered from 0 to *n*. Corresponding to each stripe is a GC thread—"GC Thread 0" through "GC Thread *n*"—tasked with garbage collection duties for its assigned segment.

Let's now look at the phases and concurrent activities of ZGC:

1. **STW Initial Mark (Pause Mark Start):** ZGC initiates the concurrent mark cycle with a brief pause to mark root objects. Recent optimizations, such as moving thread-stack scanning to the concurrent phase, have reduced the duration of this STW phase.

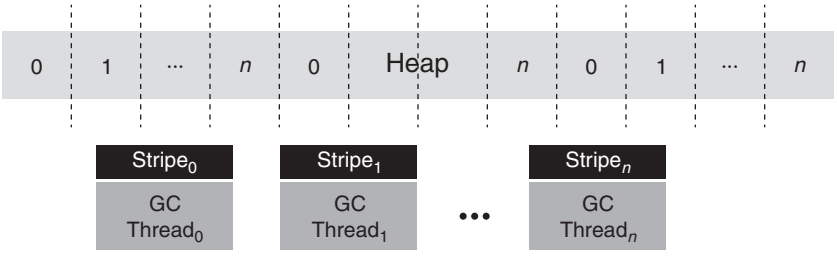

Figure 6.12 ZGC Logical Stripes

2. **Concurrent Mark/Remap:** During this phase, ZGC traces the application's live object graph and marks non-root objects. It employs a load barrier to assist in loading unmarked object pointers. Concurrent reference processing is also carried out, and thread-local handshakes are used.
3. **STW Remark (Pause Mark End):** ZGC executes a quick pause to complete the marking of objects that were in-flight during the concurrent phase. The duration of this phase is tightly controlled to minimize pause times. As of JDK 16, the amount of mark work is capped at 200 microseconds. If marking work exceeds this time limit, ZGC will continue marking concurrently, and another "pause mark end" activity will be attempted until all work is complete.
4. **Concurrent Prepare for Relocation:** During this phase, ZGC identifies the regions that will form the collection/relocation set. It also processes weak roots and non-strong references and performs class unloading.
5. **STW Relocate (Pause Relocate Start):** In this phase, ZGC finishes unloading any leftover metadata and *nmethods*.[8] It then prepares for object relocation by setting up the necessary data structures, including the use of a remap mask. This mask is part of ZGC's pointer forwarding mechanism, which ensures that any references to relocated objects are updated to point to their new locations. Application threads are briefly paused to capture roots into the collection/relocation set.
6. **Concurrent Relocate:** ZGC concurrently performs the actual work of relocating live objects from the "from" space to the "to" space. This phase can employ a technique called *self-healing*, in which the application threads themselves assist in the relocation process when they encounter an object that needs to be relocated. This approach reduces the amount of work that needs to be done during the STW pause, leading to shorter GC pause times and improved application performance.

Figure 6.13 depicts the interplay between Java application threads and ZGC background threads during the garbage collection process. The broad horizontal arrow represents the continuum of application execution over time, with Java application threads running continuously. The vertical bars symbolize STW pauses, where application threads are briefly halted to allow certain GC activities to take place.

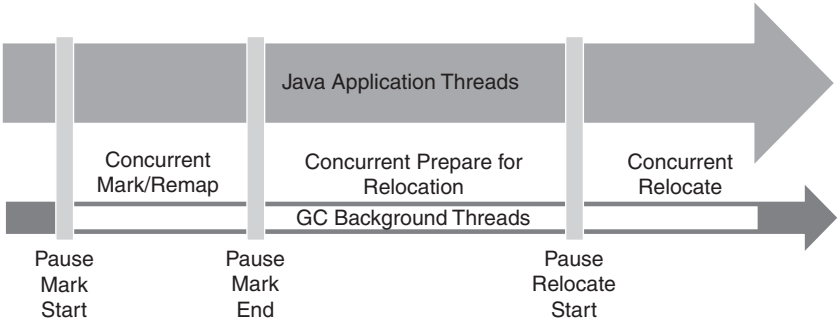

Figure 6.13 ZGC Phases and Concurrent Activities

[8] https://openjdk.org/groups/hotspot/docs/HotSpotGlossary.html

Beneath the application thread timeline is a dotted line representing the ongoing activity of GC background threads. These threads run concurrently with the application threads, performing garbage collection tasks such as marking, remapping, and relocating objects.

In Figure 6.13, the STW pauses are short and infrequent, reflecting ZGC's design goal to minimize their impact on application latency. The periods between these pauses represent phases of concurrent garbage collection activity where ZGC operates alongside the running application without interrupting it.

ZGC Core Concepts: Off-Heap Forwarding Tables

ZGC introduces off-heap forwarding tables—that is, data structures that store the new locations of relocated objects. By allocating these tables outside the heap space, ZGC ensures that the memory used by the forwarding tables does not impact the available heap space for the application. This allows for the immediate return and reuse of both virtual and physical memory, leading to more efficient memory usage. Furthermore, off-heap forwarding tables enable ZGC to handle larger heap sizes, as the size of the forwarding tables is not constrained by the size of the heap.

Figure 6.14 is a diagrammatic representation of the off-heap forwarding tables. In this figure,

- Live objects reside in heap memory.
- These live objects are relocated, and their new locations are recorded in forwarding tables.
- The forwarding tables are allocated in off-heap memory.
- The relocated objects, now in their new locations, can be accessed by application threads.

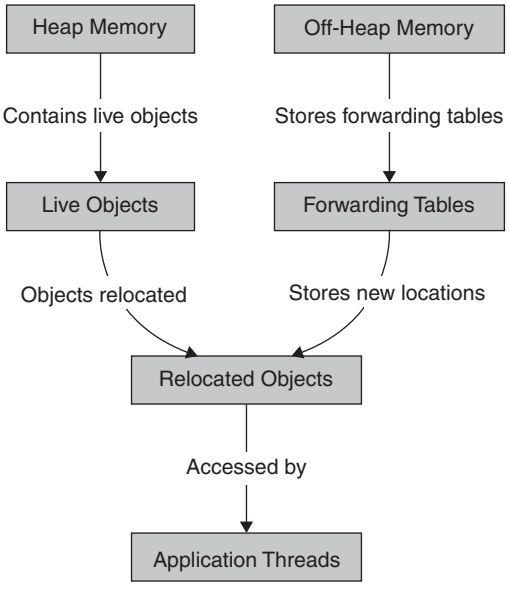

Figure 6.14 ZGC's Off-Heap Forwarding Tables

ZGC Core Concepts: *ZPages*

A *ZPage* is a contiguous chunk of memory within the heap, designed to optimize memory management and the GC process. It can have three different dimensions—small, medium, and large.[9] The exact sizes of these pages can vary based on the JVM configuration and the platform on which it's running. Here's a brief overview:

- **Small pages:** These are typically used for small object allocations. Multiple small pages can exist, and each page can be managed independently.
- **Medium pages:** These are larger than small pages and are used for medium-sized object allocations.
- **Large pages:** These are reserved for large object allocations. Large pages are typically used for objects that are too big to fit into small or medium pages. An object allocated in a large page occupies the entire page.

Figures 6.15 and 6.16 show the usage of *ZPages* in a benchmark, highlighting the frequency and volume of each page type. Figure 6.15 shows a predominance of small pages and Figure 6.16 is a zoomed-in version of the same graph to focus on the medium and large page types.

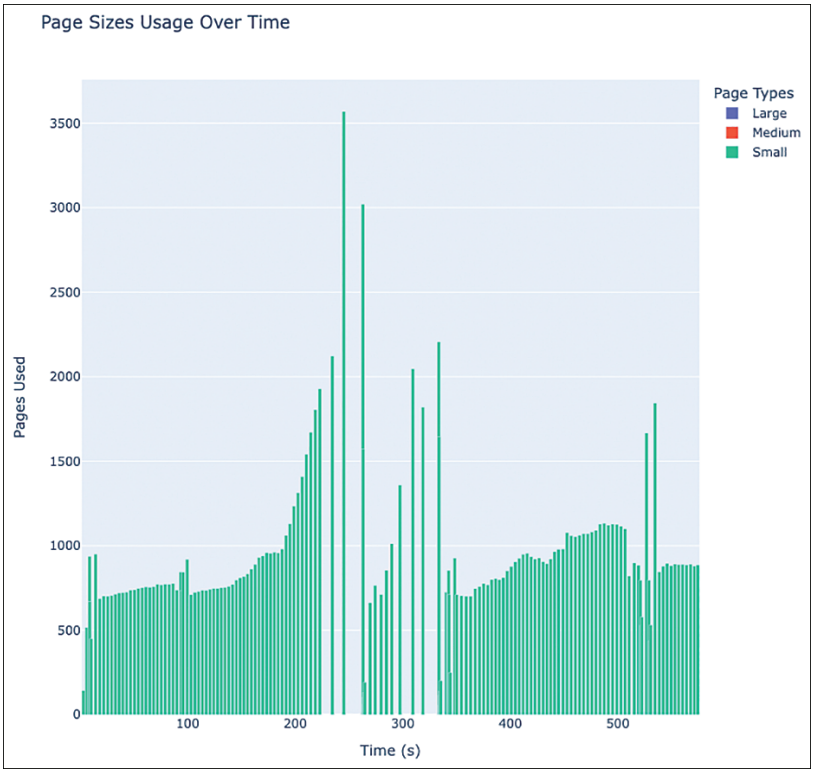

Figure 6.15 Page Sizes Usage Over Time

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

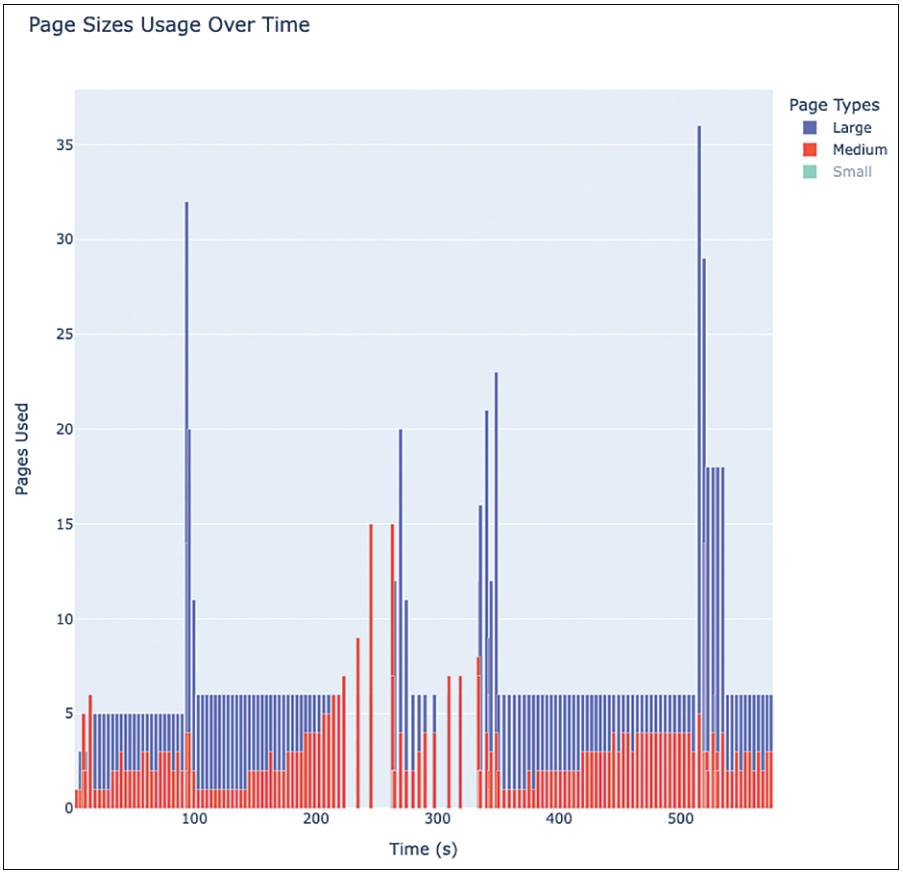

Figure 6.16 Medium and Large ZPages Over Time

Purpose and Functionality

ZGC uses pages to facilitate its multithreaded, concurrent, and region-based approach to garbage collection.

- **Region spanning:** A single *ZPage* (for example, a large *ZPage*) can cover numerous regions. When ZGC allocates a *ZPage*, it marks the corresponding regions that the *ZPage* spans as being used by that *ZPage*.
- **Relocation:** During the relocation phase, live objects are moved from one *ZPage* to another. The regions that these objects previously occupied in the source *ZPage* are then made available for future allocations.
- **Concurrency:** *ZPages* enable ZGC to concurrently operate on distinct pages, enhancing scalability and minimizing pause durations.

- **Dynamic nature:** While the size of regions is fixed, *ZPages* are dynamic. They can span multiple regions based on the size of the allocation request.

Allocation and Relocation

The *ZPage* size is chosen based on the allocation need and can be a small, medium, or large page. The chosen *ZPage* will then cover the necessary number of fixed-size regions to accommodate its size. During the GC cycle, live objects might be relocated from one page to another. The concept of pages allows ZGC to move objects around efficiently and reclaim entire pages when they become empty.

Advantages of ZPages

Using pages provides several benefits:

- **Concurrent relocation:** Since objects are grouped into pages, ZGC can relocate the contents of entire pages concurrently.
- **Efficient memory management:** Empty pages can be quickly reclaimed, and memory fragmentation issues are reduced.
- **Scalability:** The multithreaded nature of ZGC can efficiently handle multiple pages, making it scalable for applications with large heaps.
- **Page metadata:** Each *ZPage* will have associated metadata that keeps track of information such as its size, occupancy, state (whether it's used for object allocation or relocation), and more.

In essence, *ZPages* and regions are fundamental concepts in ZGC that work hand in hand. While *ZPages* provide the actual memory areas for object storage, regions offer a mechanism for ZGC to efficiently track, allocate, and reclaim memory.

ZGC Adaptive Optimization: Triggering a ZGC Cycle

In modern applications, memory management has become more dynamic, which in turn makes it crucial to have a GC that can adapt in real time. ZGC stands as a testament to this need, offering a range of adaptive triggers to initiate its garbage collection cycles. These triggers, which may include a timer, warm-up, high allocation rates, allocation stall, proactive, and high utilization, are designed to cater to the dynamic memory demands of contemporary applications. For application developers and system administrators, understanding these triggers is not just beneficial—it's essential. Grasping their nuances can lead to better management and tuning of ZGC, ensuring that applications consistently meet their performance and latency benchmarks.

Timer-Based Garbage Collection

ZGC can be configured to trigger a GC cycle based on a timer. This timer-based approach ensures that garbage collection occurs at predictable intervals, offering a level of consistency in memory management.

- **Mechanism:** When the specified timer duration elapses and if no other garbage collection has been initiated in the meantime, a ZGC cycle is automatically triggered.

- **Usage:**
  - **Setting the timer:** The interval for this timer can be specified using the JVM option `-XX:ZCollectionInterval=<value-in-seconds>`.
    - **`-XX:ZCollectionInterval=0.01:`** Configures ZGC to check for garbage collection every 10 milliseconds.
    - **`-XX:ZCollectionInterval=3:`** Sets the interval to 3 seconds.
  - **Default behavior:** By default, the value of `ZCollectionInterval` is set to 0. This means that the timer-based triggering mechanism is turned off, and ZGC will not initiate garbage collection based on time alone.
- **Example logs:** When ZGC operates based on this timer, the logs would resemble the following output:

```
[21.374s][info][gc,start    ] GC(7) Garbage Collection (Timer)
[21.374s][info][gc,task     ] GC(7) Using 4 workers
…
[22.031s][info][gc          ] GC(7) Garbage Collection (Timer) 1480M(14%)->1434M(14%)
```

  These logs indicate that a GC cycle was triggered due to the timer, and they provide details about the memory reclaimed during this cycle.
- **Considerations:** The timer-based approach, while offering predictability, should be used judiciously. It's essential to understand ZGC's adaptive triggers. Using a timer-based collection should ideally be a supplementary strategy—a periodic "cleanse" to bring ZGC to a known state. This approach can be especially beneficial for tasks like refreshing classes or processing references, as ZGC cycles can handle concurrent unloading and reference processing.

Warm-up–Based GC

A ZGC cycle can be triggered during the warm-up phase of an application. This is especially beneficial during the ramp-up phase of the application when it hasn't reached its full operational load. The decision to trigger a ZGC cycle during this phase is influenced by the heap occupancy.

- **Mechanism:** During the application's ramp-up phase, ZGC monitors heap occupancy. When it reaches the thresholds of 10%, 20%, or 30%, ZGC triggers a *warm-up* cycle. This can be seen as priming the GC, preparing it for the application's typical memory patterns. If heap usage surpasses one of the thresholds and no other garbage collection has been initiated, a ZGC cycle is triggered. This mechanism allows the system to gather early samples of the GC duration, which can be used by other rules.
- **Usage:** ZGC's threshold, `soft_max_capacity`, determines the maximum heap occupancy it aims to maintain.[10] For the warm-up–based GC cycle, ZGC checks heap occupancy at 10%, 20%, and 30% of the `soft_max_capacity`, provided no other garbage

[10] https://malloc.se/blog/zgc-softmaxheapsize

collections have been initiated. To influence these warm-up cycles, end users can adjust the `-XX:SoftMaxHeapSize=<size>` command-line option.

- **Example logs:** When ZGC operates under this rule, the logs would resemble the following output:

```
[7.171s][info][gc,start    ] GC(2) Garbage Collection (Warmup)
[7.171s][info][gc,task     ] GC(2) Using 4 workers
…
[8.842s][info][gc          ] GC(2) Garbage Collection (Warmup) 2078M(20%)->1448M(14%)
```

  From the logs, it's evident that a warm-up cycle was triggered when the heap occupancy reached 20%.

- **Considerations:** The warm-up/ramp-up phases are crucial for applications, especially those that build extensive caches during start-up. These caches are often used heavily in subsequent transactions. By initiating a ZGC cycle during warm-up, the application can ensure that the caches are efficiently built and optimized for future use. This ensures a smoother transition to consistent performance as the application transitions from the warm-up phase to handling real-world transactions.

GC Based on High Allocation Rates

When an application allocates memory at a high rate, ZGC can initiate a GC cycle to free up memory. This prevents the application from exhausting the heap space.

- **Mechanism:** ZGC continuously monitors the rate of memory allocation. Based on the observed allocation rate and the available free memory, ZGC estimates the time until the application will run out of memory (OOM). If this estimated time is less than the expected duration of a GC cycle, ZGC triggers a cycle.

  ZGC can operate in two modes based on the `UseDynamicNumberOfGCThreads` setting:

  - **Dynamic GC workers:** In this mode, ZGC dynamically calculates the number of GC workers needed to avoid OOM. It considers factors such as the current allocation rate, variance in the allocation rate, and the average time taken for serial and parallel phases of the GC.
  - **Static GC workers:** In this mode, ZGC uses a fixed number of GC workers (`ConcGCThreads`). It estimates the time until OOM based on the moving average of the sampled allocation rate, adjusted with a safety margin for unforeseen allocation spikes. The safety margin ("headroom") is calculated as the space required for all worker threads to allocate a small ZPage and for all workers to share a single medium ZPage.

- **Usage:** Adjusting the `UseDynamicNumberOfGCThreads` setting and the `-XX:SoftMaxHeapSize=<size>` command-line option, and possibly adjusting the `-XX:ConGCThreads=<number>`, allows fine-tuning of the garbage collection behavior to suit specific application needs.

- **Example logs:** The following logs provide insights into the triggering of the GC cycle due to high allocation rates:

```
[221.876s][info][gc,start    ] GC(12) Garbage Collection (Allocation Rate)
[221.876s][info][gc,task     ] GC(12) Using 3 workers
```

```
…
[224.073s][info][gc          ] GC(12) Garbage Collection (Allocation Rate)
5778M(56%)->2814M(27%)
```

From the logs, it's evident that a GC was initiated due to a high allocation rate, and it reduced the heap occupancy from 56% to 27%.

GC Based on Allocation Stall

When the application is unable to allocate memory due to insufficient space, ZGC will trigger a cycle to free up memory. This mechanism ensures that the application doesn't stall or crash due to memory exhaustion.

- **Mechanism:** ZGC continuously monitors memory allocation attempts. If an allocation request cannot be satisfied because the heap doesn't have enough free memory and an allocation stall has been observed since the last GC cycle, ZGC initiates a GC cycle. This is a reactive approach, ensuring that memory is made available promptly to prevent application stalls.
- **Usage:** This trigger is inherent to ZGC's design and doesn't require explicit configuration. However, monitoring and understanding the frequency of such triggers can provide insights into the application's memory usage patterns and potential optimizations.
- **Example logs:**

```
…
[265.354s][info][gc          ] Allocation Stall (<thread-info>) 1554.773ms
[265.354s][info][gc          ] Allocation Stall (<thread-info>) 1550.993ms
…
…
[270.476s][info][gc,start    ] GC(19) Garbage Collection (Allocation Stall)
[270.476s][info][gc,ref      ] GC(19) Clearing All SoftReferences
[270.476s][info][gc,task     ] GC(19) Using 4 workers
…
[275.112s][info][gc          ] GC(19) Garbage Collection (Allocation Stall)
7814M(76%)->2580M(25%)
```

From the logs, it's evident that a GC was initiated due to an allocation stall, and it reduced the heap occupancy from 76% to 25%.

- **Considerations:** Frequent allocation stalls might indicate that the application is nearing its memory limits or has sporadic memory usage spikes. It's imperative to monitor such occurrences and consider increasing the heap size or optimizing the application's memory usage.

GC Based on High Usage

If the heap utilization rate is high, a ZGC cycle can be triggered to free up memory. This is the first step before triggering ZGC cycles due to high allocation rates. The GC is triggered as a preventive measure if the heap is at 95% occupancy and based on your application's allocation rates.

- **Mechanism:** ZGC continuously monitors the amount of free memory available. If the free memory drops to 5% or less of the heap's total capacity, a ZGC cycle is triggered.

This is especially useful in scenarios where the application has a very low allocation rate, such that the allocation rate rule doesn't trigger a GC cycle, but the free memory is still decreasing steadily.

- **Usage:** This trigger is inherent to ZGC's design and doesn't require explicit configuration. However, understanding this mechanism can help in tuning the heap size and optimizing memory usage.
- **Example logs:**

```
[388.683s][info][gc,start    ] GC(36) Garbage Collection (High Usage)
[388.683s][info][gc,task     ] GC(36) Using 4 workers
…
[396.071s][info][gc          ] GC(36) Garbage Collection (High Usage) 9698M(95%)->2726M(27%)
```

- **Considerations:** If the high usage–based GC is triggered frequently, it might indicate that the heap size is not adequate for the application's needs. Monitoring the frequency of such triggers and the heap's occupancy can provide insights into potential optimizations or the need to adjust the heap size.

Proactive GC

ZGC can proactively initiate a GC cycle based on heuristics or predictive models. This approach is designed to anticipate the need for a GC cycle, allowing the JVM to maintain smaller heap sizes and ensure smoother application performance.

- **Mechanism:** ZGC uses a set of rules to determine whether a proactive GC should be initiated. These rules consider factors such as the growth of heap usage since the last GC, the time elapsed since the last GC, and the potential impact on application throughput. The goal is to perform a GC cycle when it's deemed beneficial, even if the heap still has a significant amount of free space.
- **Usage:**
  - **Enabling/disabling proactive GC:** Proactive garbage collection can be enabled or disabled using the JVM option `ZProactive`.
    - **`-XX:+ZProactive`:** Enables proactive garbage collection.
    - **`-XX:-ZProactive`:** Disables proactive garbage collection.
  - **Default behavior:** By default, proactive garbage collection in ZGC is *enabled* (`-XX:+ZProactive`). When it is enabled, ZGC will use its heuristics to decide when to initiate a proactive GC cycle, even if there's still a significant amount of free space in the heap.
- **Example logs:**

```
[753.984s][info][gc,start    ] GC(41) Garbage Collection (Proactive)
[753.984s][info][gc,task     ] GC(41) Using 4 workers
…
[755.897s][info][gc          ] GC(41) Garbage Collection (Proactive) 5458M(53%)->1724M(17%)
```

From the logs, it's evident that a proactive GC was initiated, which reduced the heap occupancy from 53% to 17%.

- **Considerations:** The proactive GC mechanism is designed to optimize application performance by anticipating memory needs. However, it's vital to monitor its behavior to ensure it aligns with the application's requirements. Frequent proactive GCs might indicate that the JVM's heuristics are too aggressive, or the application has unpredictable memory usage patterns.

The depth and breadth of ZGC's adaptive triggers underscore its versatility in managing memory for diverse applications. These triggers, while generally applicable to most newer garbage collectors, find a unique implementation in ZGC. The specific details and sensitivity of these triggers can vary based on the GC's design and configuration. Although ZGC's default behavior is quite robust, a wealth of optimization potential lies beneath the surface. By delving into the specifics of each trigger and interpreting the associated logs, practitioners and system specialists can fine-tune ZGC's operations to better align them with their application's unique memory patterns. In the fast-paced world of software development, such insights can be the difference between good performance and exceptional performance.

Advancements in ZGC

Since its experimental debut in JDK 11 LTS, ZGC has undergone numerous enhancements and refinements in subsequent JDK releases. Here is an overview of some key improvements to ZGC from JDK 11 to JDK 17:

- **JDK 11: Experimental introduction of ZGC**. ZGC brought innovative concepts such as load barriers and colored pointers to the table, aiming to provide near-real-time, low-latency garbage collection and scalability for large heap sizes.
- **JDK 12: Concurrent class unloading.** Class unloading is the process of removing classes that are no longer needed from memory, thereby freeing up space and making the memory available for other tasks. In traditional GCs, class unloading requires a STW pause, which can lead to longer GC pause times. However, with the introduction of concurrent class unloading in JDK 12, ZGC became able to unload classes while the application is still running, thereby reducing the need for STW pauses and leading to shorter GC pause times.
- **JDK 13: Uncommitting unused memory.** In JDK 13, ZGC was enhanced so that it uncommitted unused memory more rapidly. This improvement led to more efficient memory usage and potentially shorter GC pause times.
- **JDK 14: Windows and macOS support added.** ZGC extended its platform support in JDK 14, becoming available on Windows and macOS. This broadened the applicability of ZGC to a wider range of systems.

- **JDK 15: Production readiness, compressed class pointers, and NUMA-awareness.** By JDK 15, ZGC had matured to a production-ready state, offering a robust low-latency garbage collection solution for applications with large heap sizes. Additionally, ZGC became NUMA-aware, considering memory placement on multi-socket systems with the NUMA architecture. This led to improved GC pause times and overall performance.
- **JDK 16: Concurrent thread stack processing.** JDK 16 introduced concurrent thread stack processing to ZGC, further reducing the work done during the remark pause and leading to even shorter GC pause times.
- **JDK 17: Dynamic GC threads, faster terminations, and memory efficiency.** ZGC now dynamically adjusts the number of GC threads for better CPU usage and performance. Additionally, ZGC's ability to abort ongoing GC cycles enables faster JVM termination. There's a reduction in mark stack memory usage, making the algorithm more memory efficient.

Future Trends in Garbage Collection

For runtimes, the memory management unit is the largest "disrupter" in efforts to harmonize operations with the underlying hardware. Specifically:

- The GC not only traverses the live object graph, but also relocates the live objects to achieve (partial) compaction. This relocation can disrupt the state of CPU caches because it moves live objects from different regions into a new region.
- While TLAB bump-the-pointer allocations are efficient, the disruption mainly occurs when the GC starts its work, given the contrasting nature of these two activities.
- Concurrent work, which is becoming increasingly popular with modern GC algorithms, introduces overhead. This is due to maintenance barriers and the challenge of keeping pace with the application. The goal is to achieve pause times in milliseconds or less and ensure graceful degradation in scenarios where the GC might be overwhelmed.

In today's cloud computing era, where data centers are under pressure to be both cost-effective and efficient, power and footprint efficiency are the real needs to be addressed. This is especially true when considering the GC's need to work concurrently and meet the increasing demands of modern applications. Incremental marking and the use of regions are strategies employed to make this concurrent work more efficient.

One of the emerging trends in modern GC is the concept of "tunable work units"—that is, the ability to adjust and fine-tune specific tasks within the GC process to optimize performance and efficiency. By breaking the GC process down into smaller, tunable units, it becomes possible to better align the GC's operations with the application's needs and the underlying hardware capabilities. This granularity allows for more precise control over aspects such as partial compaction, maintenance barriers, and concurrent work, leading to more predictable and efficient garbage collection cycles (Figure 6.17).

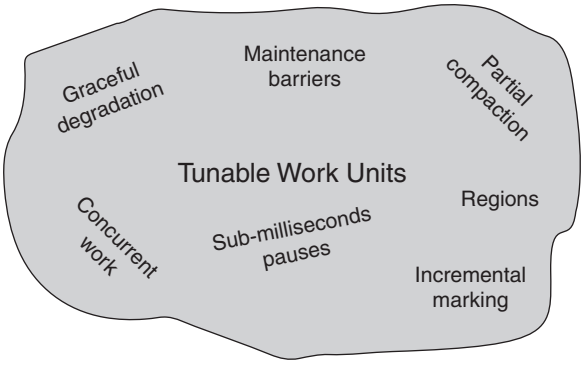

Figure 6.17 A Concurrent Garbage Collector's Tunable Work Units

As systems become more powerful and multi-core processors become the norm, GCs that utilize these cores concurrently will be at an advantage. However, power efficiency, especially in data centers and cloud environments, is crucial. It's not just about scaling with CPU usage, but also about how efficiently that power is used.

GCs need to be in sync with both the hardware on which they operate and the software patterns they manage. Doing so will require addressing the following needs:

- Efficient CPU and system cache utilization
- Minimizing latency for accessing such caches and bringing them into the right states
- Understanding allocation size, patterns, and GC pressures

By being more aware of the hardware and software patterns, and adapting its strategies based on the current state of the system, a GC can aid its intrinsic adaptiveness and can significantly improve both performance and efficiency.

Extrinsic tools such as machine learning can further boost GC performance by analyzing application behavior and memory usage patterns, with machine learning algorithms potentially automating much of the complex and time-consuming process of GC tuning. But it's not just about tuning: It's also about prompting the GC to behave in a particular way by recognizing and matching patterns. This can help avoid potential GC behavior issues before they occur. Such machine learning algorithms could also help in optimizing power usage by learning the most efficient ways to perform garbage collection tasks, thereby reducing the overall CPU usage and power consumption.

The future of garbage collection in the JVM will undoubtedly be dynamic and adaptive, driven by both intrinsic improvements and extrinsic optimizations. As we continue to push the boundaries of what's possible with modern applications, it's clear that our garbage collectors will need to keep pace, evolving and adapting to meet these new challenges head-on.

The field of garbage collection is not static; it evolves alongside the landscape of software development. New GCs are being developed, and the existing ones are being improved to handle the changing needs of applications. For instance,

- Low-pause collectors like ZGC and Shenandoah aim to minimize GC pause times so as to improve application performance.
- Recent JDK releases have featured continuous improvements in existing GCs (for example, G1, ZGC, and Shenandoah) aimed at improving their performance, scalability, and usability. As these GCs continue to evolve, they may become suitable for a wider range of workloads. As of the writing of this book, the enhancements made so far have sown the seeds that will likely turn ZGC and Shenandoah into generational GCs.
- Changes in application development practices, such as the rise of microservices and cloud-native applications, have led to an increased focus on containerization and resource efficiency. This trend will influence improvements to GCs because such applications often have different memory management requirements than the traditional monolithic applications.
- The rise of big data and machine learning applications, which often require handling large data sets in memory, is pushing the boundaries of garbage collection. These applications require GCs that can efficiently manage large heaps and provide good overall throughput.

As these trends continue to unfold, it will be important for architects and performance engineers to stay informed and be ready to adapt their garbage collection strategies as needed. This will lead to further improvements and enhancements to OpenJDK HotSpot VM garbage collectors.

Practical Tips for Evaluating GC Performance

When it comes to GC tuning, my advice is to build upon the methodologies and performance engineering approach detailed in Chapter 5. Start by understanding your application's memory usage patterns. GC logs are your best friends. There are various log parsers and plotters that can help you visualize your application patterns like a lit Christmas tree. Enriching your understanding with GC logs will lay the foundation for effective GC tuning.

And don't underestimate the importance of testing. Testing can help you understand the synergies between your chosen collector and your application's memory demands. Once we establish the synergies and their gaps, we can move to targeted, incremental tuning. Such tuning efforts can have a significant impact on application performance, and their incremental nature will allow you to thoroughly test any changes to ensure they have the desired effect.

Evaluating the performance of different GCs with your workloads is a crucial step in optimizing your application's performance. Here are some practical tips that resonate with the benchmarking strategies and structured approach championed in Chapter 5:

- **Measure GC performance:** Utilize tools like Java's built-in JMX (Java Management Extensions) or VisualVM[11] for intermittent, real-time monitoring of GC performance. These tools can provide valuable metrics such as total GC time, maximum GC pause time, and heap usage. For continuous, low-overhead monitoring, especially in

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

production, Java Flight Recorder (JFR) is the tool of choice. It provides detailed runtime information with minimal impact on performance. GC logs, by comparison, are ideal for post-analysis, capturing a comprehensive history of GC events. Analyzing these logs can help you identify trends and patterns that might not be apparent from a single snapshot.

- **Understand GC metrics:** It's important to understand what the different GC metrics mean. For instance, a high GC time might indicate that your application is spending a lot of time in garbage collection, which could impact its performance. Similarly, a high heap usage might indicate that your application has a high "GC churn" or it isn't sized appropriately for your live data set. "GC churn" refers to how frequently your application allocates and deallocates memory. High churn rates and under-provisioned heaps can lead to frequent GC cycles. Monitoring these metrics will help you make informed decisions about GC tuning.
- **Interpret metrics in the context of your application:** The impact of GC on your application's performance can vary depending on its specific characteristics. For instance, an application with a high allocation rate might be more affected by GC pause times than an application with a low allocation rate. Therefore, it's important to interpret GC metrics in the context of your application. Consider factors such as your application's workload, transaction patterns, and memory usage patterns when interpreting these metrics. Additionally, correlate these GC metrics with overall application performance indicators such as response time and throughput. This holistic approach ensures that GC tuning efforts are aligned not just with memory management efficiency but also with the application's overall performance objectives.
- **Understand GC triggers and adaptation:** Garbage collection is typically triggered when the eden space is nearing capacity or when a threshold related to the old generation is reached. Modern GC algorithms are adaptive and can adjust their behavior based on the application's memory usage patterns in these different areas of the heap. For instance, if an application has a high allocation rate, the young GC might be triggered more frequently to keep up with the rate of object creation. Similarly, if the old generation is highly utilized, the GC might spend more time compacting the heap to free up space. Understanding these adaptive behaviors can help you tune your GC strategy more effectively.
- **Experiment with different GC algorithms:** Different GC algorithms have different strengths and weaknesses. For instance, the Parallel GC is designed to maximize throughput, while ZGC is designed to minimize pause times. Additionally, the G1 GC offers a balance between throughput and pause times and can be a good choice for applications with a large heap size. Experiment with different GC algorithms to see which one works best with your workload. Remember, the best GC algorithm for your application depends on your specific performance requirements and constraints.
- **Tune GC parameters:** Most GC algorithms offer a range of parameters that you can fine-tune to optimize performance. For instance, you can adjust the heap size or the ratio of young-generation to old-generation space. Be sure to understand what these parameters do before adjusting them, as improper tuning can lead to worse performance. Start with the default settings or follow recommended best practices as a baseline. From there, make incremental adjustments and closely monitor their effects. Consistent with the

experimental design principles outlined in Chapter 5, testing each change in a controlled environment is crucial. Utilizing a staging area that mirrors your production environment allows for a safe assessment of each tuning effort, ensuring no disruptions to your live applications. Keep in mind that GC tuning is inherently an iterative performance engineering process. Finding the optimal configuration often involves a series of careful experiments and observations.

When selecting a GC for your application, it is crucial to assess its performance in the context of your workloads. Remember, the objective is not solely to enhance GC performance, but rather to ensure your application provides the best possible user experience.

Evaluating GC Performance in Various Workloads

Effective memory management is pivotal to the efficient execution of Java applications. As a central aspect of this process, garbage collection must be optimized to deliver both efficiency and adaptability. To achieve this, we will examine the needs and shared characteristics across a variety of open-source frameworks such as Apache Hadoop, Apache Hive, Apache HBase, Apache Cassandra, Apache Spark, and other such projects. In this context, we will delve into different transaction patterns, their relationship with in-memory storage mechanisms, and their effects on heap management, all of which inform our approach to optimizing GC to best suit our application and its transactions, while carefully considering their unique characteristics.

Types of Transactional Workloads

The projects and frameworks can be broadly categorized into three groups based on their transactional interactions with in-memory databases and their influence on the Java heap:

- **Analytics:** Online analytical processing (OLAP)
- **Operational stores:** Online transactional processing (OLTP)
- **Hybrid:** Hybrid transactional and analytical processing (HTAP)

Analytics (OLAP)

This transaction type typically involves complex, long-running queries against large data sets for business intelligence and data mining purposes. The term “stateless” describes these interactions, signifying that each request and response pair is independent, with no information retained between transactions. These workloads are often characterized by high throughput and large heap demands. In the context of garbage collection, the primary concern is the management of transient objects and the high allocation rates, which could lead to a large heap size. Examples of OLAP applications in the open-source world include Apache Hadoop and Spark, which are used for distributed processing of large data sets across clusters of computers.[12] The

[12] www.infoq.com/articles/apache-spark-introduction/

memory management requirements for these kinds of applications often benefit from GCs (for example, G1) that can efficiently handle large heaps and provide good overall throughput.

Operational Stores (OLTP)

This transaction type is usually associated with serving real-time business transactions. OLTP applications are characterized by a large number of short, atomic transactions such as updates to a database. They require fast, low-latency access to small amounts of data in large databases. In these interactive transactions, the state of data is frequently read from or written to; thus, these interactions are "stateful." In the context of Java garbage collection, the focus is on managing the high allocation rates and pressure from medium-lived data or transactions. Examples include NoSQL databases like Apache Cassandra and HBase. For these types of workloads, low-pause collectors such as ZGC and Shenandoah are often a good fit, as they are designed to minimize GC pause times, which can directly impact the latency of transactions.

Hybrid (HTAP)

HTAP applications involve a relatively new type of workload that is a combination of OLAP and OLTP, requiring both analytical processing and transactional processing. Such systems aim to perform real-time analytics on operational data. For instance, this category includes in-memory computing systems such as Apache Ignite, which provide high-performance, integrated, and distributed in-memory computing capabilities for processing OLAP and OLTP workloads. Another example from the open-source world is Apache Flink, which is designed for stream processing but also supports batch processing tasks. Although not a stand-alone HTAP database, Flink complements HTAP architectures with its real-time processing power, especially when integrated with systems that manage transactional data storage. The desired GC traits for such workloads include high throughput for analytical tasks, combined with low latency for real-time transactions, suggesting an enhancement of GCs might be optimal. Ideally, these workloads would use a throughput-optimized GC that minimizes pauses and maintains low latency—optimized generational ZGC, for example.

Synthesis

By understanding the different types of workloads (OLAP, OLTP, HTAP) and their commonalities, we can pinpoint the most efficient GC traits for each. This knowledge aids in optimizing GC performance and adaptiveness, which contributes to the overall performance of Java applications. As the nature of these workloads continues to evolve and new ones emerge, the ongoing study and enhancement of GC strategies will remain crucial to ensure they stay effective and adaptable. In this context, automatic memory management becomes an integral part of optimizing and adapting to diverse workload scenarios in the Java landscape.

The key to optimizing GC performance lies in understanding the nature of the workload and choosing the right GC algorithm or tuning parameters accordingly. For instance, OLAP workloads, which are stateless and characterized by high throughput and large heap demands, may benefit from GCs like optimized G1. OLTP workloads, which are stateful and require fast, low-latency access to data, may be better served by low-pause GCs like generational ZGC or

Shenandoah. HTAP workloads, which combine the characteristics of both OLAP and OLTP, will drive improvements to GC strategies to achieve optimal performance.

Adding to this, the increasing presence of machine learning and other advanced analytical tools in performance tuning suggests that future GC optimization might leverage these technologies to further refine and automate tuning processes. Considering the CPU utilization associated with each GC strategy is also essential because it directly influences not just garbage collection efficiency but the overall application throughput.

By tailoring the garbage collection strategy to the specific needs of the workload, we can ensure that Java applications run efficiently and effectively, delivering the best possible performance for the end user. This approach, combined with the ongoing research and development in the field of GC, will ensure that Java continues to be a robust and reliable platform for a wide range of applications.

Live Data Set Pressure

The "live data set" (LDS) comprises the volume of live data in the heap, which can impact the overall performance of the garbage collection process. The LDS encompasses objects that are still in use by the application and have not yet become eligible for garbage collection.

In traditional GC algorithms, the greater the live data set pressure, the longer the GC pauses can be, as the algorithm has to process more live data. However, this doesn't hold true for all GC algorithms.

Understanding Data Lifespan Patterns

Different types of data contribute to LDS pressure in varying ways, highlighting the importance of understanding your application's data lifespan patterns. Data can be broadly categorized into four types based on their lifespan:

- **Transient data:** These short-lived objects are quickly discarded after use. They usually shouldn't survive past minor GC cycles and are collected in the young generation.
- **Short-lived data:** These objects have a slightly longer lifespan than transient data but are still relatively short-lived. They may survive a few minor GC cycles but are typically collected before being promoted to the old generation.
- **Medium-lived data:** These objects have a lifespan that extends beyond several minor GC cycles and often get promoted to the old generation. They can increase the heap's occupancy and apply pressure on the GC.
- **Long-lived data:** These objects remain in use for a significant portion of the application's lifetime. They reside in the old generation and are collected only during old-generation cycles.

Impact on Different GC Algorithms

G1 is designed to provide more consistent pause times and better performance by incrementally processing the heap in smaller regions and predicting which regions will yield the most garbage. Yet, the volume of live data still impacts its performance. Medium-lived data and short-lived data surviving minor GC cycles can increase the heap's occupancy and apply unnecessary pressure, thereby affecting G1's performance.

ZGC, by comparison, is designed to have pause times that are largely independent of the size of the heap or the LDS. ZGC maintains low latency by performing most of its work concurrently without stopping application threads. However, under high memory pressure, it employs load-shedding strategies, such as throttling the allocation rate, to maintain its low-pause guarantees. Thus, while ZGC is designed to handle larger LDS more gracefully, extreme memory pressures can still affect its operations.

Optimizing Memory Management

Understanding the lifespan patterns of your application's data is a key factor in effectively tuning your garbage collection strategy. This knowledge allows you to strike a balance between throughput and pause times, which are the primary trade-offs offered by different GC algorithms. Ultimately, this leads to more efficient memory management.

For instance, consider an application with a high rate of transient or short-lived data creation. Such an application might benefit from an algorithm that optimizes minor GC cycles. This optimization could involve allowing objects to age within the young generation, thereby reducing the frequency of old-generation collections and improving overall throughput.

Now consider an application with a significant amount of long-lived data. This data might be spread throughout the lifespan of the application, serving as an active cache for transactions. In this scenario, a GC algorithm that incrementally handles old-generation collections could be beneficial. Such an approach can help manage the larger LDS more efficiently, reducing pause times and maintaining application responsiveness.

Obviously, the behavior and performance of GCs can vary based on the specific characteristics of your workloads and the configuration of your JVM. As such, regular monitoring and tuning are essential. By adapting your GC strategy to the changing needs of your application, you can maintain optimal GC performance and ensure the efficient operation of your application.

This page intentionally left blank

Chapter 7

Runtime Performance Optimizations: A Focus on Strings, Locks, and Beyond

Introduction

Efficiency in runtime performance is crucial for scaling applications and meeting user demands. At the heart of this is the JVM, managing runtime facets such as just-in-time (JIT) compilation, garbage collection, and thread synchronization—each profoundly influencing Java applications' performance.

Java's evolution from Java 8 to Java 17 showcases a commitment to performance optimization, evident in more than 350 JDK Enhancement Proposals (JEPs). Noteworthy are optimizations like the reduced footprint of `java.lang.String`, advancements in contended locking mechanisms, and the *indy-fication* of string concatenation,[1] all significantly contributing to runtime efficiency. Additionally, optimizations such as spin-wait hints, though less apparent, are crucial in enhancing system performance. They are particularly effective in cloud platforms and multithreaded environments where the wait time is expected to be very short. These intricate optimizations, often concealed beneath the complexities of large-scale environments, are central to the performance gains realized by service providers.

Throughout my career, I have seen Java's runtime evolve dynamically. Major updates and preview features, like virtual threads, are increasingly supported by various frameworks[2] and libraries and adeptly integrated by large organizations. This integration, as discussed by industry leaders at a QCon San Francisco conference,[3] underscores Java's strategic role in these organizations and points toward a shift to a deeper, architectural approach to JVM optimizations.

[1] https://openjdk.org/jeps/280

[2] https://spring.io/blog/2023/11/23/spring-boot-3-2-0-available-now

[3] https://qconsf.com/track/oct2023/jvm-trends-charting-future-productivity-performance

This trend is particularly evident in high-performance databases and storage systems where efficient threading models are critical. The chapter aims to demystify the "how" and "why" behind these performance benefits, covering a broad spectrum of runtime aspects from bytecode engineering to string pools, locks, and the nuances of threading.

Further emphasizing the runtime, this chapter explores G1's string deduplication and transitions into the realm of concurrency and parallelism, unveiling Java's Executor Service, thread pools, and the `ForkJoinPool` framework. Additionally, it introduces Java's `CompletableFuture` and the groundbreaking concepts of virtual threads and *continuations,* providing a glimpse into the future of concurrency with Project Loom. While various frameworks may abstract these complexities, an overview and reasoning behind their evaluation and development is imperative for leveraging the JVM's threading advancements.

Understanding the holistic optimization of the internal frameworks within the JVM, as well as their coordination and differentiation, empowers us to better utilize the tools that support them, thereby bridging the benefits of JVM's runtime improvements to practical applications.

A key objective of this exploration is to discuss how the JVM's internal engineering augments code execution efficiency. By dissecting these enhancements through the lens of performance engineering tools, profiling instruments, and benchmarking frameworks, we aim to reveal their full impact and offer a thorough understanding of JVM performance engineering. Although these improvements to the JVM, may manifest differently in distributed environments, this chapter distills the core engineering principles of the JVM alongside the performance engineering tools available to us.

As we journey through runtime's performance landscape, novices and seasoned developers alike will appreciate how the JVM's continual enhancements streamline the development process. These refinements contribute to a more stable and performant foundation, which leads to fewer performance-related issues and a smoother deployment process, refining the comprehensive system experience in even the most demanding of environments.

String Optimizations

As the Java platform has evolved, so, too, has its approach to one of the most ubiquitous constructs in programming: the string. Given that strings are a fundamental aspect of nearly every Java application, any improvements in this area can significantly impact the speed, memory footprint, and scalability of an application. Recognizing this relationship, the Java community has dedicated considerable effort to string optimization. This section focuses on four significant optimizations that have been introduced across various Java versions:

- **Literal and interned string optimization:** This optimization involves the management of a "pool of strings" to reduce heap occupancy, thereby enhancing memory utilization.
- **String deduplication in G1:** The deduplication feature offered by the G1 garbage collector aims to reduce heap occupancy by deduplicating identical `String` objects.
- **Indy-fication of string concatenation:** We'll look at the change in how string concatenation is managed at the bytecode level, which enhances performance and adds flexibility.

- **Compact strings:** We will examine how the shift from `char[]` to `byte[]` backing arrays for strings can reduce applications' memory footprint, halving the space requirements for ASCII text.

Each optimization not only brings unique advantages but also presents its own set of considerations and caveats. As we explore each of these areas, we aim to understand both their technical aspects and their practical implications for application performance.

Literal and Interned String Optimization in HotSpot VM

Reducing the allocation footprint in Java has always been a significant opportunity for enhancing performance. Long before Java 9 introduced the shift from `char[]` to `byte[]` for backing arrays in strings, the OpenJDK HotSpot VM employed techniques to minimize memory usage, particularly for strings.

A principal optimization involves a "pool of strings." Given strings are immutable and often duplicated across various parts of an application, the `String` class in Java utilizes a pool to conserve heap space. When the `String.intern()` method is invoked, it checks if an identical string is already in the pool. If so, the method returns a reference to the pre-existing string instead of allocating a new object.

Consider an instance, *String1*, which would have had a `String1` object and a backing `char[]` in the days before Java 9 (Figure 7.1).

Invoking `String1.intern()` checks the pool for an existing string with the same value. If none is found, `String1` is added to the pool (Figure 7.2).

Figure 7.3 depicts the situation before the strings are interned. When a new string, `String2`, is created and interned, and it equals `String1`, it will reference the same `char[]` in the pool (Figure 7.4).

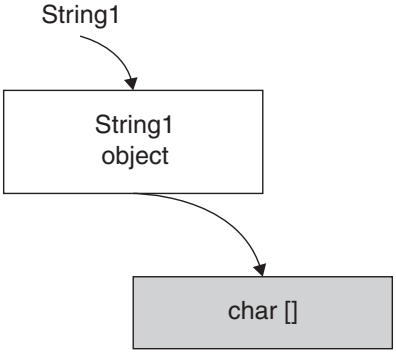

Figure 7.1 A String Object

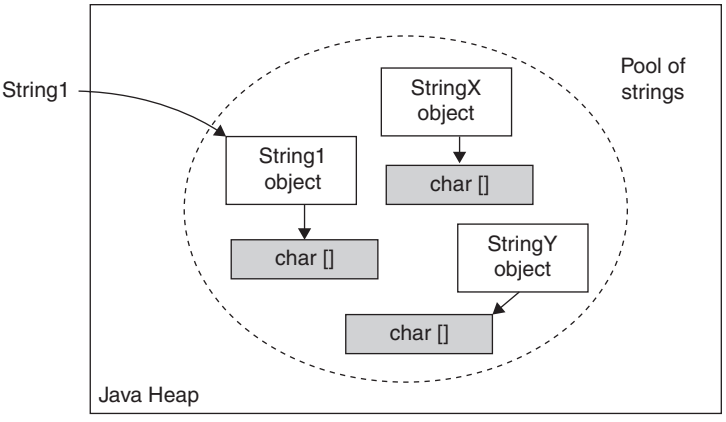

Figure 7.2 A Pool of Strings Within the Java Heap

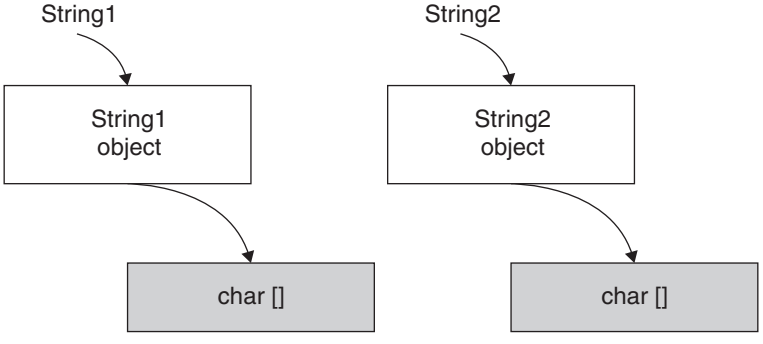

Figure 7.3 Two String Objects Before Interning

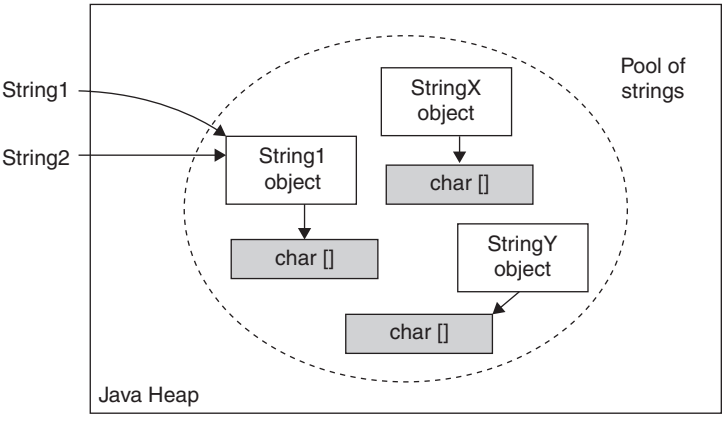

Figure 7.4 Interned String Objects Sharing the Same Reference

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

String Deduplication Optimization and G1 GC

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

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

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

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

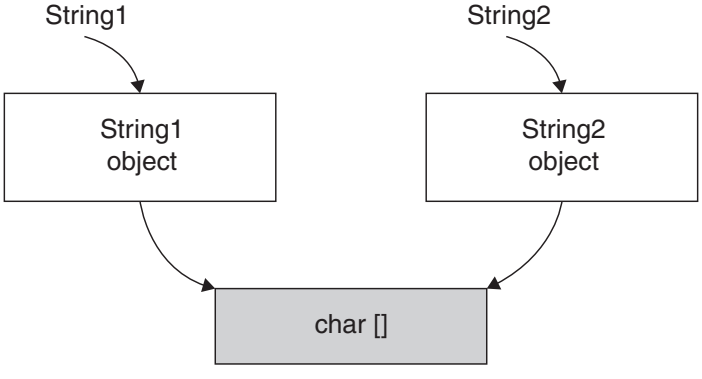

Figure 7.5 String Deduplication Optimization in Java

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

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

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

Reducing Strings' Footprint

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

Indy-fication of String Concatenation

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

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

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

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

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

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

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

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

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

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

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

Here's the bytecode disassembly:

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

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

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

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

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

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

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

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

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

Here's the BSM excerpt from the output:

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

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

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

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

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

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

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

Compact Strings

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

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

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

Impact of Compact Strings

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

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

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

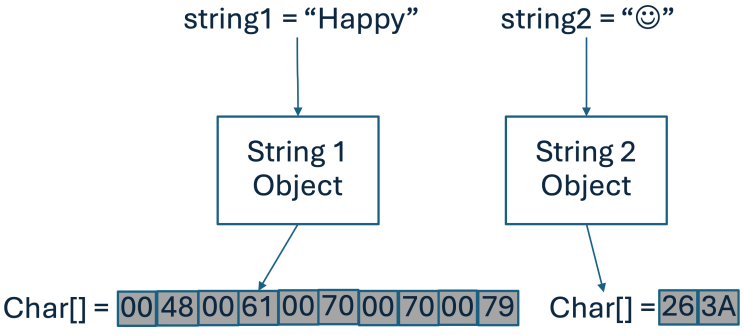

Figure 7.6 Traditional String Representation in JDK 8

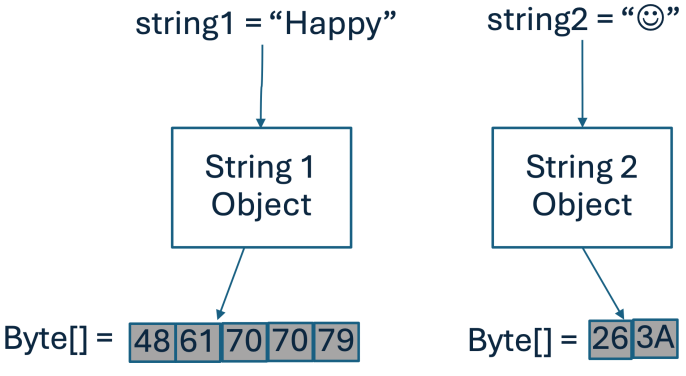

Figure 7.7 Compact Strings Representation in JDK 17

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

Visualizing String Representations

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

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

Profiling Compact Strings: Insights into JVM Internals Using NetBeans IDE

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

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

```
package mergefiles;

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

public class MergeFiles {

    public static void main(String[] args) {

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

```
            String appline, gcline;

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

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

    }

}
```

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

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

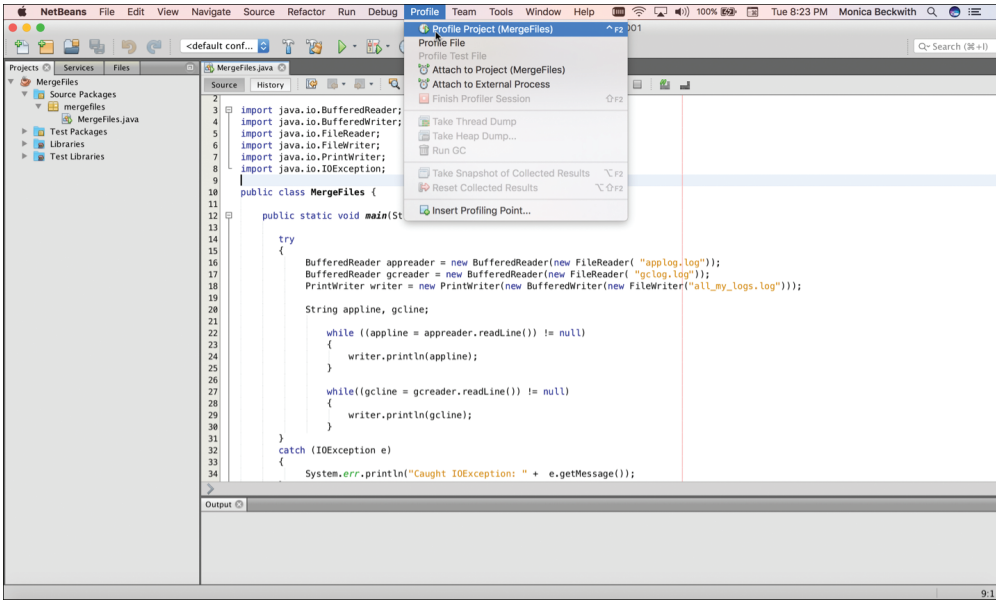

Figure 7.8 Apache NetBeans Profile Menu

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

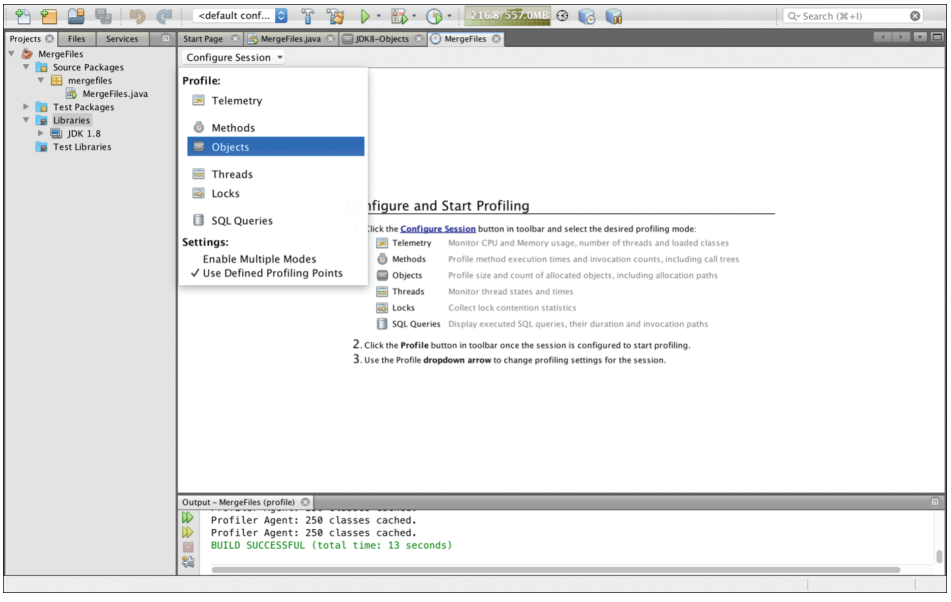

Figure 7.9 Apache NetBeans Configure Session

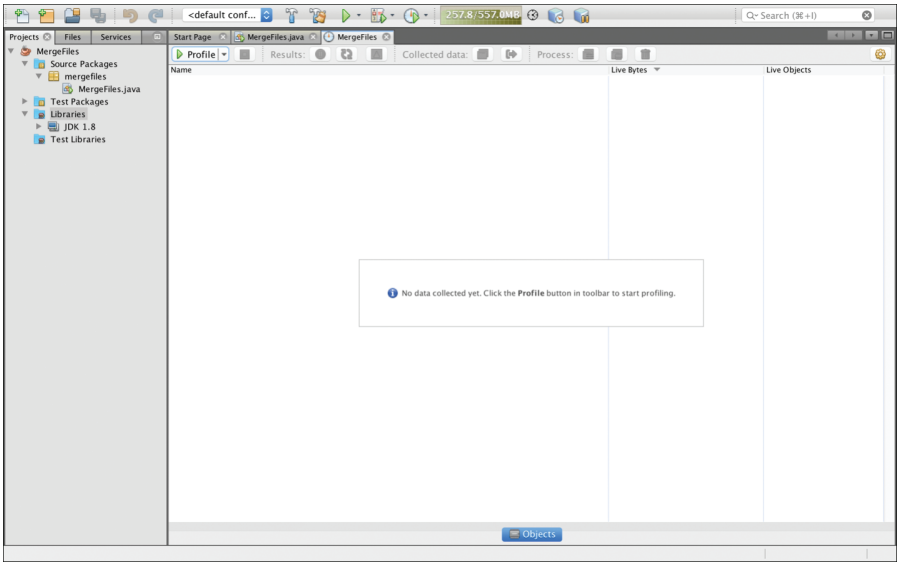

Figure 7.10 Apache NetBeans Profiling Setup

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

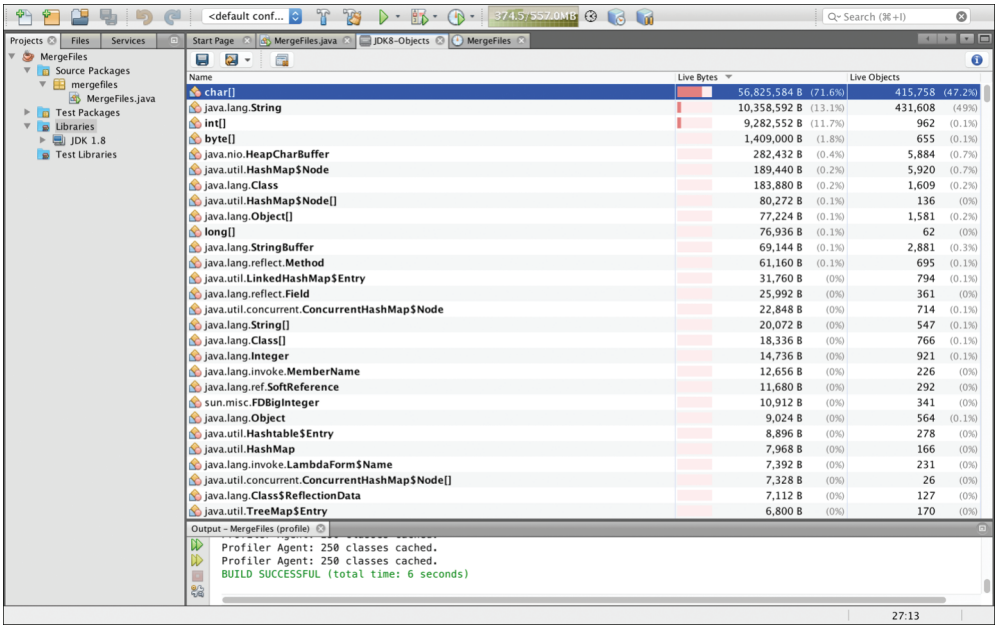

Figure 7.11 Apache NetBeans Profiling Complete for All Classes

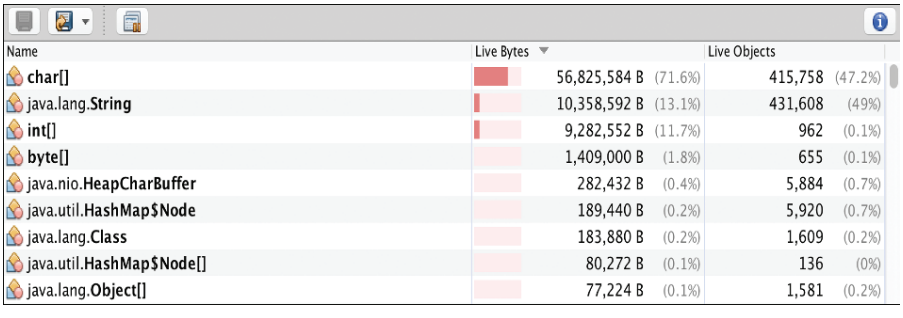

Figure 7.12 Zoomed-in Profiling Window

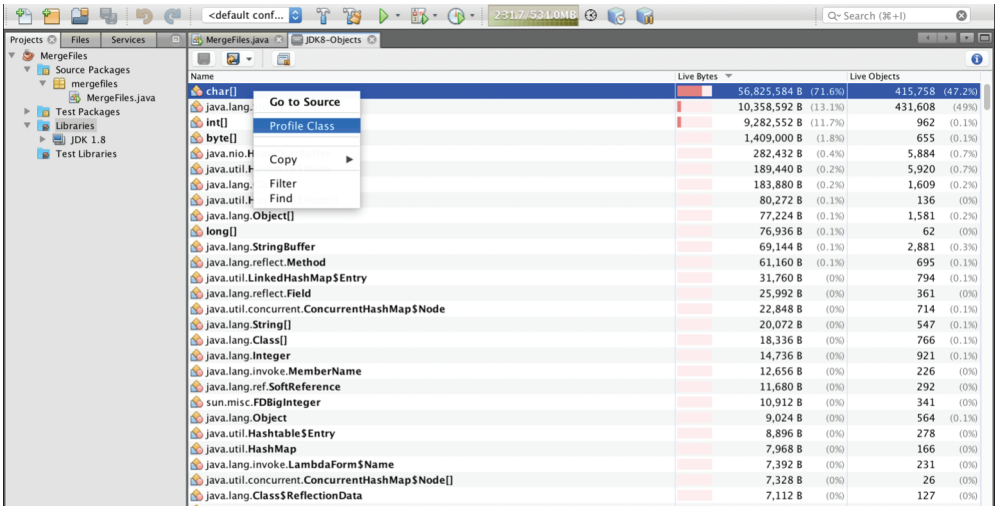

Figure 7.13 Apache NetBeans Profile Class Menu

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

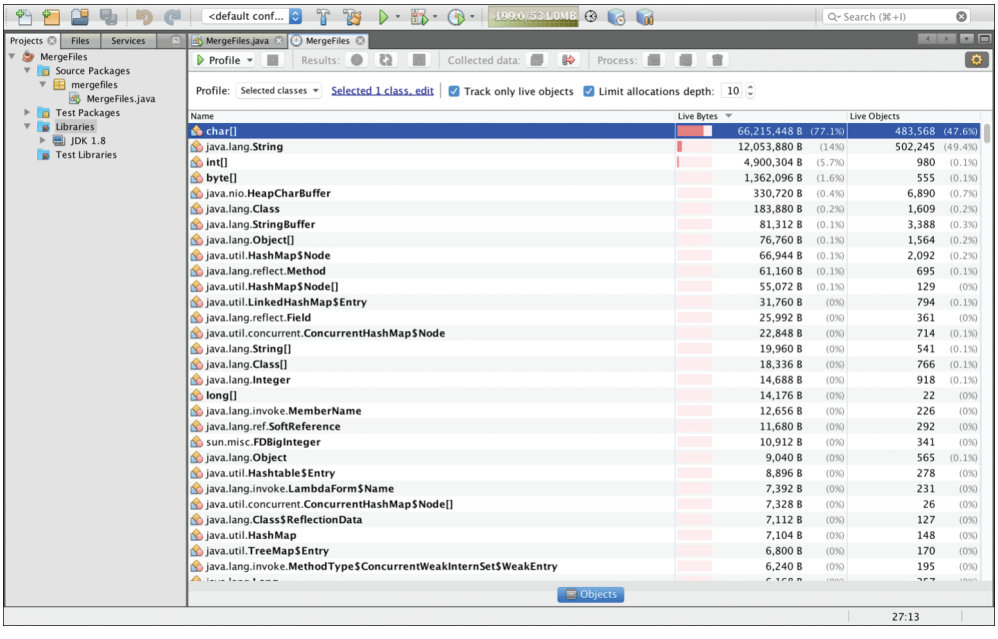

Figure 7.14 Apache NetBeans Profiling Single Class

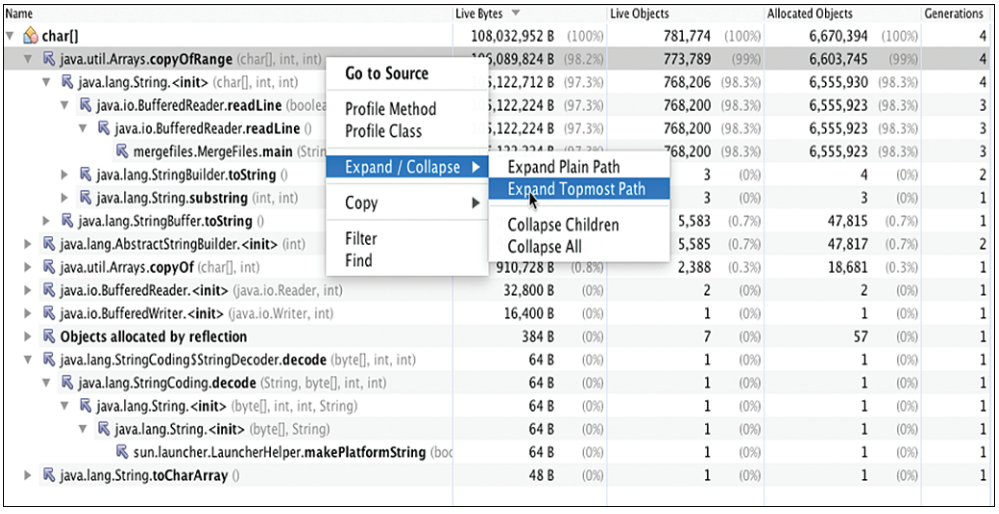

Figure 7.15 Apache NetBeans Expanded Class Menu

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

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

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

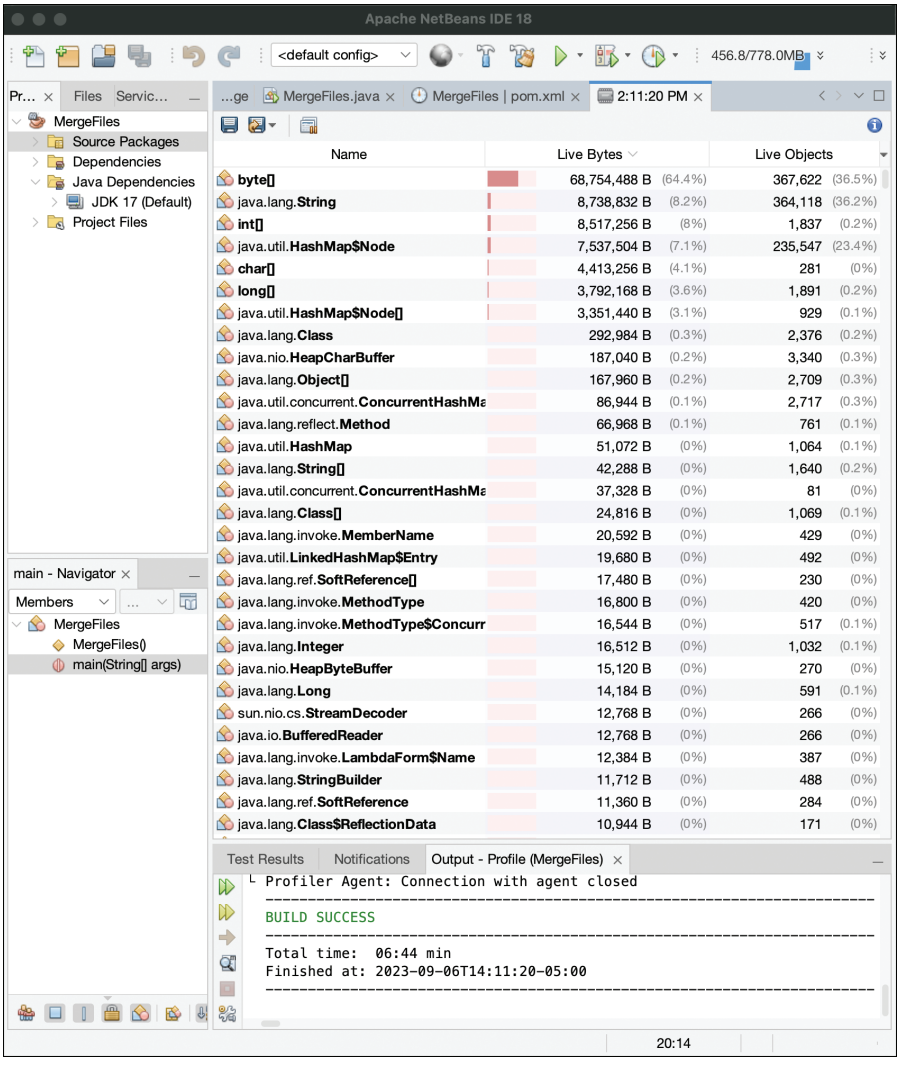

Figure 7.16 Apache NetBeans 18 Captured Profile

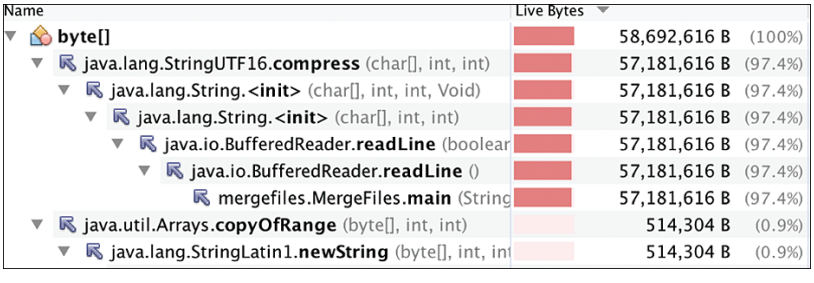

Figure 7.17 Zoomed-in Profile Showing byte[] at the Top

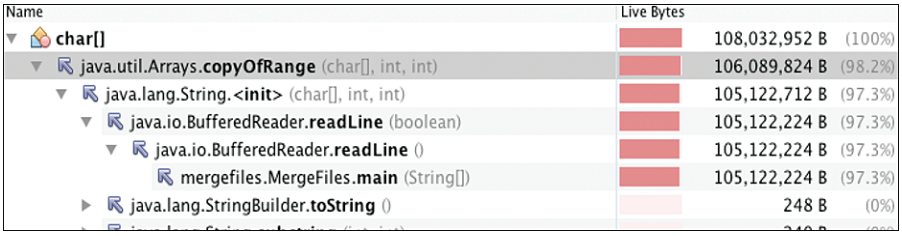

Figure 7.18 Zoomed-in Profile Showing Char[] at the Top

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

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

Enhanced Multithreading Performance: Java Thread Synchronization

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

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

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

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

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

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

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

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

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

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

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

Synchronized methods offer a similar mechanism:

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

NOTE

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

The Role of Monitor Locks

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

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

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

Lock Types in OpenJDK HotSpot VM

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

Lock Contention Dynamics

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

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

Uncontended and Deflated Locks

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

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

Contended and Inflated Locks

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

Code Example and Analysis

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

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

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

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

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

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

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

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

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

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

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

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

The code takes the following actions:

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

Advancements in Java's Locking Mechanisms

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

API-Level Enhancements

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

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

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

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

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

HotSpot VM Optimizations

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

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

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

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

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

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

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

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

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

Optimizing Contention: Enhancements since Java 9

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

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

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

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

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

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

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

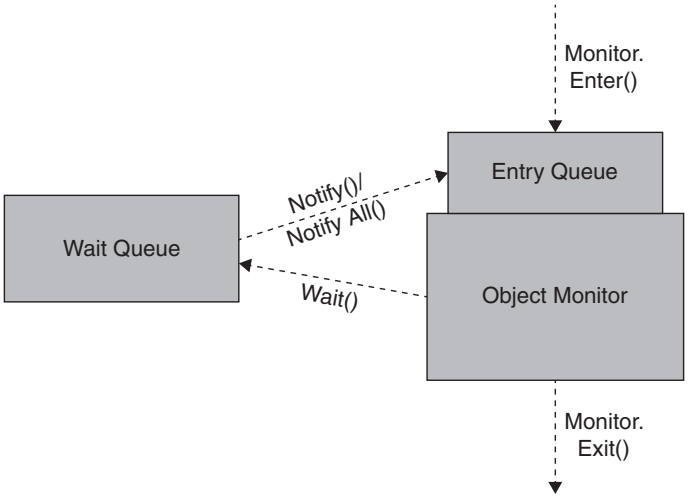

Figure 7.19 Object's Monitor Wait and Entry Queue

Improved Java Monitor Enter and Fast Exit Operations

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

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

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

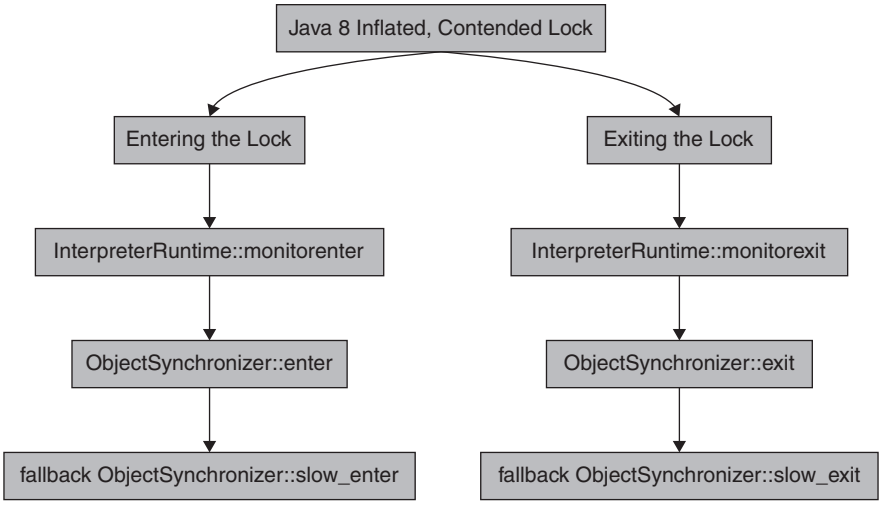

Figure 7.20 High-Level Overview of Contended Lock Enter and Exit Call Paths for Java 8

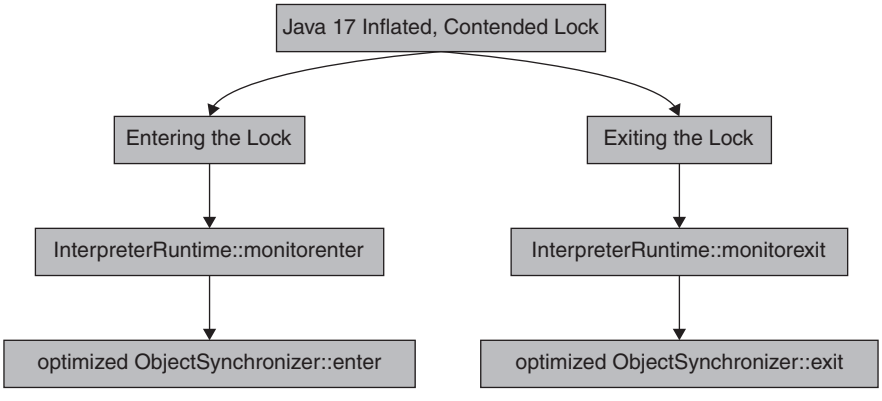

Figure 7.21 High-Level Overview of Contended Lock Enter and Exit Call Paths for Java 17

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

Visualizing Contended Lock Optimization: A Performance Engineering Exercise

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

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

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

Our exploration centers on two pivotal aspects:

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

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

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

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

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

[12] See Chapter 5 for more information.

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

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

Setup and Data Collection

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

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

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

Testing and Performance Analysis

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

---

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

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

Benchmarking Configuration

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

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

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

Benchmarking Execution

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

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

In the preceding line, note the following options:

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

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

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

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

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

Comparing Performance Between Java 8 and Java 17

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

Flamegraph Analysis

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

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

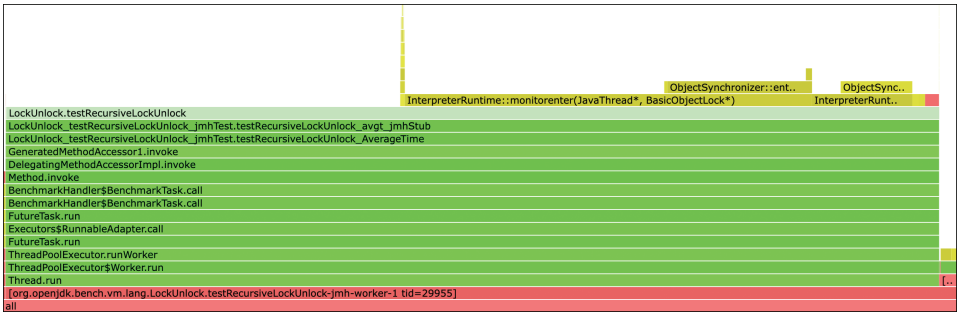

Figure 7.22 Zoomed-in Call Stack of the RecursiveLockUnLock Test on Java 17

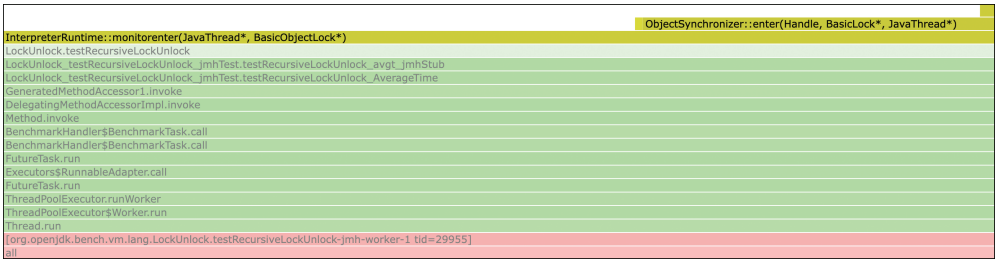

Figure 7.23 Zooming Further into Java 17's Monitor Enter Stack

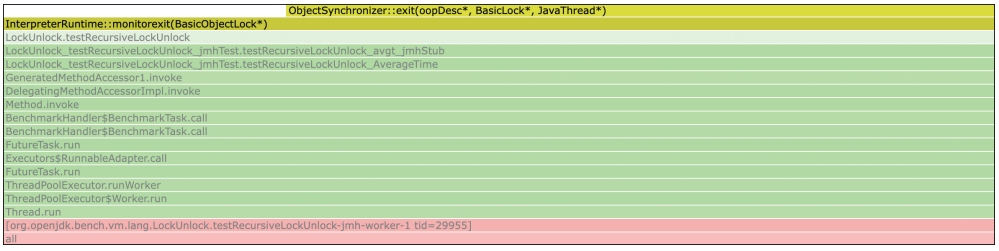

Figure 7.24 Zoomed-in View of Java 17's Monitor Exit Stack

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

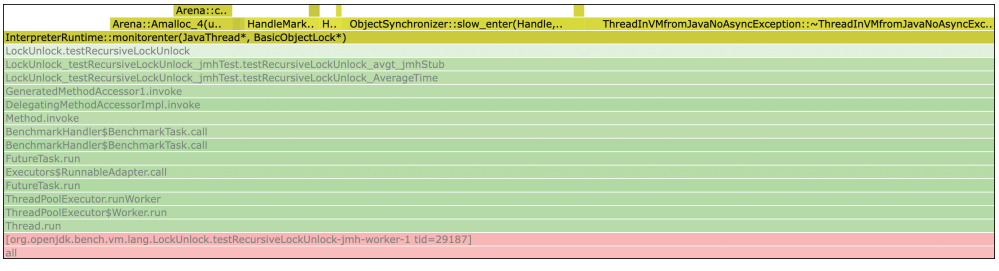

Figure 7.25 Zoomed-in View of Java 8's Monitor Enter Stack

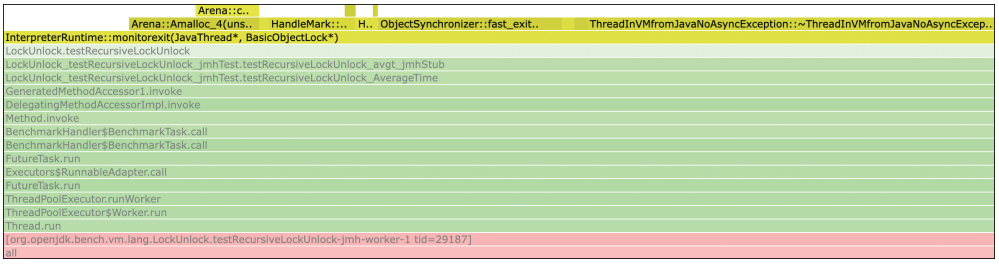

Figure 7.26 Zoomed-in View of Java 8's Monitor Exit Stack

Benchmarking Results

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

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

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

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

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

Call Tree Analysis

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

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

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

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

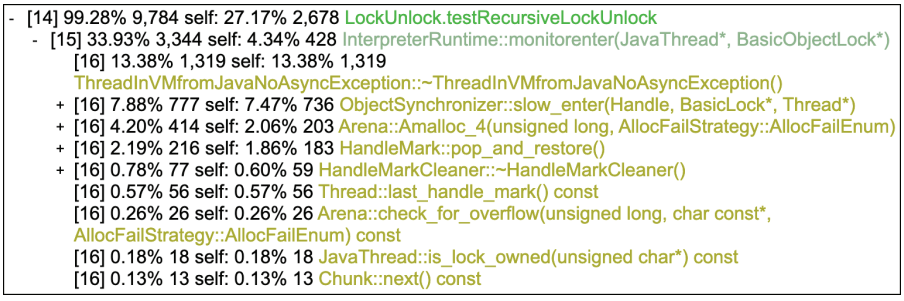

Figure 7.28 Java 8's Call Tree View for the testRecursiveLockUnlock's monitorenter Routine

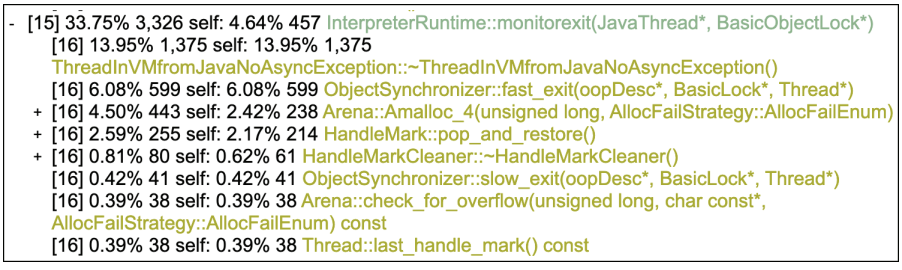

Figure 7.29 Java 8's Call Tree View for the testRecursiveLockUnlock's monitorexit Routine

Detailed Comparison

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

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

Excerpts from Java 17's call tree show:

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

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

Inclusive time for `monitorenter` and `monitorexit` =

9,648 (inclusive time for `testRecursiveLockUnLock`)

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

= 5,458 samples

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

Excerpts from Java 8's call tree show:

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

Inclusive time for `monitorenter` and `monitorexit` =

9,784 (inclusive time for `testRecursiveLockUnLock`)

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

= 6670 samples

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

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

Java 17 Versus Java 8: Interpreting the Data

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

Performance improvement in percent speed increase:

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

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

Consistent profiling conditions:

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

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

Inclusive time for monitor entry and exit:

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

Exclusive time in `testRecursiveLockUnlock` method:

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

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

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

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

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

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

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

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

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

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

Java 17 optimizations:

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

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

Table 7.4 **Java 17 monitorenter Stack**

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

Table 7.5 **Java 17 monitorexit Stack**

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

Synthesizing Contended Lock Optimization: A Reflection

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

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

In essence, this exploration was more than a peek under the JVM's hood—it was a deep dive into the essence of performance engineering within Java's ecosystem. Through rigorous measurement, profiling, and analysis, we've illuminated the path of Java's evolution in handling contended locks, showcasing the ongoing commitment to enhancing the efficiency and speed of Java applications worldwide.

Spin-Wait Hints: An Indirect Locking Improvement

Before diving into the intricacies of spin-wait hints, it's essential to understand their broader context. Although most Java developers might never directly use spin-wait hints, they play a crucial role in improving the efficiency of concurrent algorithms and data structures in high-load servers and compute-intensive cloud applications, where performance issues due to locking and context switching are prevalent. By optimizing how threads wait for locks, these hints are subtly woven into the JVM's thread scheduling and processor interactions, allowing for a more intelligent approach to busy-wait loops. This strategic use of spin-wait hints not only results in reduced CPU consumption but also minimizes unnecessary context switches—improving thread responsiveness on systems that are designed to leverage such hints.

NOTE For those unfamiliar with the term "PAUSE," it's an Intel instruction used to optimize how threads wait in spin-loops, reducing CPU power consumption during active polling. While it might seem like a niche topic, understanding this instruction can provide valuable insights into how Java and the underlying hardware interact to optimize performance.

Leveraging Intrinsics for Enhanced Performance

Building upon the concept of adaptive spinning introduced in Java 7, Java has made further strides in optimizing spin-wait mechanisms. These enhancements to the generated code effectively leverage the unique capabilities of various hardware platforms. For instance, architectures supporting the Single Input Multiple Data (SIMD) instruction set benefit from vectorization enhancements in the OpenJDK HotSpot VM. This is achieved through intrinsic assembly stubs that accelerate operations, and through auto-vectorization, where the compiler automatically parallelizes code execution for increased efficiency.

Intrinsics are precompiled, hand-optimized assembly stubs designed to enhance JIT-compiled code by leveraging the specific capabilities of the underlying architecture. These can range from complex mathematical computations to super-efficient memory management—providing a pathway for JIT-compiled code to achieve performance that's closer to what can be done in lower-level languages.

The diagram in Figure 7.30 depicts JIT code execution when no intrinsics are involved. Contrast that with the same process but with intrinsics shown in Figure 7.31. These stubs deliver optimized native code sequences and facilitate direct access to advanced CPU instructions or provide access to functionalities that aren't available in the Java programming language.

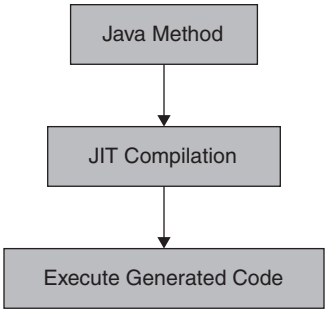

Figure 7.30 JIT Code Execution Without Intrinsics

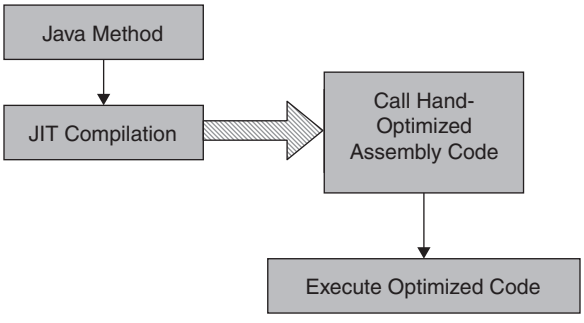

Figure 7.31 JIT Code Execution with Intrinsics

Understanding Spin-Loop Hints and PAUSE Instructions

In a multiprocessor (MP) system, such as x86-64 architectures, when a field or variable is locked and a thread secures the lock, other threads vying for the same lock enter a wait loop (also called a busy-wait loop, spin-wait loop, or spin-loop). These threads are said to be actively polling. Active polling has its merits: A thread can instantly access a lock once it's available. However, the flip side is that such looped polling consumes a significant amount of power while the thread is merely waiting to perform its task. Moreover, MP systems have a processing pipeline that speculates about loop execution based on always-taken or never-taken branches and ends up simplifying the loop based on speculation. When the looping thread gets access to the lock, the pipeline needs to be cleared (a process known as pipeline flushing) because the speculations made by the processor are no longer valid. This can impose a performance penalty on MP systems.

To address this issue, Streaming SIMD Extensions 2 (SSE2)[15] https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html on the x86-64 architecture introduced the PAUSE instruction. This instruction hints at a spin-loop, allowing the addition of a predetermined delay to the loop. Java 9 embraced this development by introducing the `onSpinWait()` method in `java.lang.Thread`, allowing the Java runtime to intrinsically invoke the PAUSE instruction on supporting CPUs.

[15] https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html

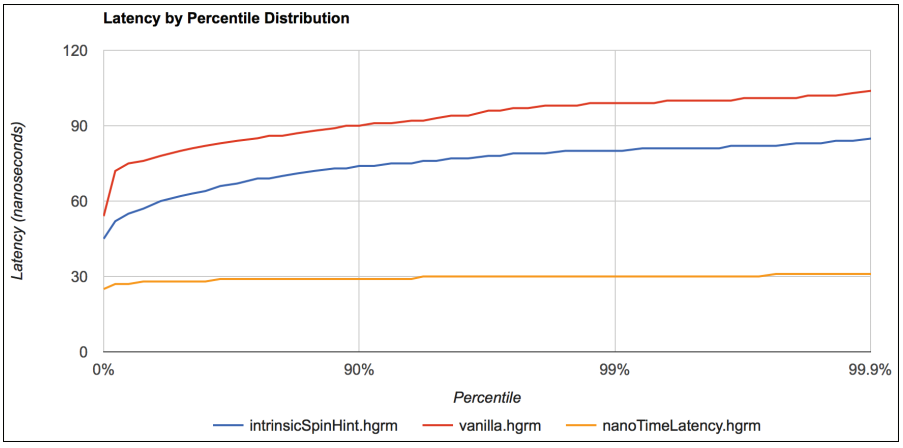

Figure 7.32 Improvements Due to Spin-Wait Hints

NOTE On non-Intel architectures or Intel architectures predating the introduction of the PAUSE instruction, this instruction is treated as a NOP (no operation), ensuring compatibility without performance benefits.

JEP 285: *Spin-Wait Hints*[16] has proven invaluable for Java applications, particularly by enabling the effective use of the PAUSE instruction in scenarios with short spin-loops. Gil Tene, a Java champion, chief technology officer and co-founder of Azul Systems, and a co-author of the spin-wait JEP, has demonstrated its practical advantages, as evidenced by his graph reproduced in Figure 7.32.

Transitioning from the Thread-per-Task Model to More Scalable Models

After exploring locks and spin-wait hints, it's essential to understand the evolution of concurrency models in Java. In the realm of Java concurrency, the traditional thread-per-task model, in which each new task spawns a new thread, has its limitations. While straightforward, this approach can become resource-intensive, especially for applications juggling numerous short-lived tasks. As we transition from this model, two primary approaches to scalability emerge:

- **Asynchronous/reactive style:** This approach breaks up a large task into smaller tasks that run in a pipeline. While it offers scalability, it may not always align seamlessly with the Java platform's features.

[16] https://openjdk.org/jeps/285

- **Thread-per-request with virtual threads:** This approach encourages the use of blocking, synchronous, and maintainable code. It leverages the full spectrum of Java language features and platform capabilities, making it more intuitive for many developers.

Our discussion will further explore the latter's practical applications, focusing on Java's concurrency framework enhancements.

Traditional One-to-One Thread Mapping

To better understand the traditional thread-per-task model, let's look at a typical thread creation example. When a new task needs execution, we create a Java process thread (jpt):

```
// Traditional thread creation
Thread thread = new Thread(() -> {
    // Task to be executed in the thread
});
thread.start();
```

Figure 7.33 illustrates the runtime structure and the interactions between the JVM process and OS when a thread is introduced by the JVM process. The diagram showcases the convention threading model in Java. Each Java thread (jpt0, jpt1, jptn) is typically mapped one-to-one to an OS thread. Every thread possesses its own Java stack, program counter (PC) register, native stack, and runtime data. Whereas the OS is responsible for scheduling and determining thread priorities, Java manages synchronization, locks, wait queues, and thread pools. The standard practice of mapping Java threads to OS threads in a one-to-one ratio can pose challenges. For instance, when a Java process thread (jpt) is blocked, it can also block the corresponding OS thread, thereby diminishing scalability.

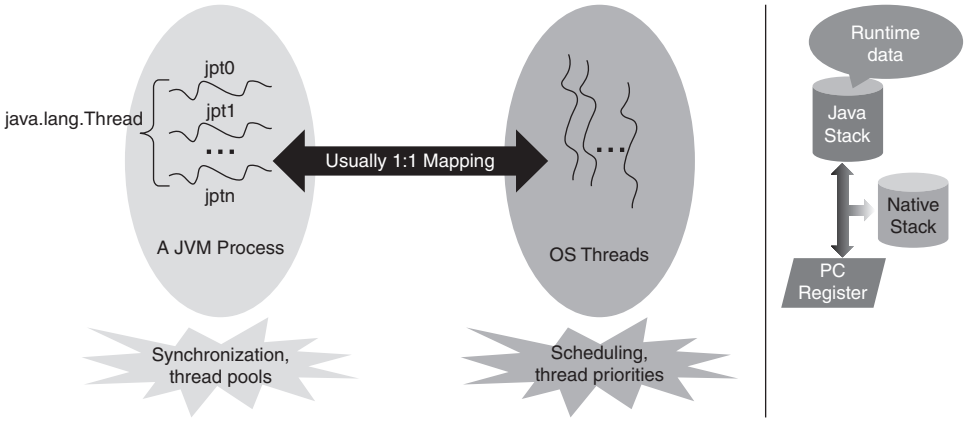

Figure 7.33 A JVM Process, Operating System Threads, and Runtime Structures

Increasing Scalability with the Thread-per-Request Model

To address the scalability limitations of one-to-one thread mapping, Java introduced higher-level concurrency utilities. The thread-per-request model, for instance, allows each request or transaction to be handled by a separate thread, improving scalability. The model is often used in conjunction with tools like the Executor Service and frameworks like `ForkJoinPool` and `CompletableFuture` that provide ways to manage concurrency.

Java Executor Service and Thread Pool

Java's Executor Service framework manages a pool of worker threads, allowing tasks to be executed on an existing thread from the pool, rather than creating a new thread for each task. This approach, often referred to as the "thread pool" or "worker-thread" model, is more efficient and scalable. When used to implement the thread-per-request model, it aids scalability by ensuring there's a thread for every transaction; these threads have low overhead because they are served from a thread pool.

Here's an example of employing a thread pool to implement an Executor Service for incoming requests:

```
ExecutorService executorService = Executors.newFixedThreadPool(10);

for (int i = 0; i < 1000; i++) {
  executorService.execute(() -> {
    // Task to be executed in one of the threads of the pool
  });
}

executorService.shutdown();
```

Figure 7.34 illustrates the Java Executor Service, which manages a thread pool. The diagram depicts a blocking queue (Blocking Q) holding tasks (Task0, Task1, Task2) that are waiting to be executed by the threads in the pool. This model facilitates efficient thread management and helps prevent resource waste, since threads in the pool wait on the blocking queue when not processing tasks, instead of consuming CPU cycles. However, this can lead to high overhead if not managed properly, especially when you're dealing with blocking operations and a large number of threads.

Java ForkJoinPool Framework

While Executor Services and thread pools offer a significant improvement over the traditional thread-per-task model, they have limitations. They can struggle with efficiently handling a large number of small tasks, especially in applications that would benefit from parallel processing. This is where the `ForkJoinPool` framework comes into play.

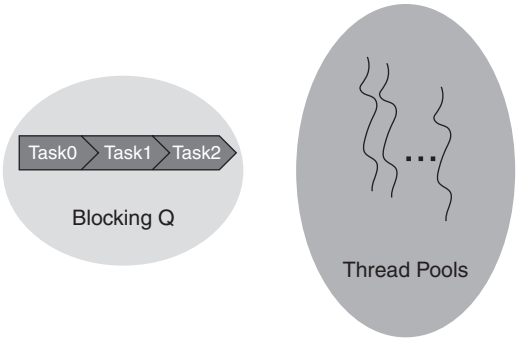

Figure 7.34 A Blocking Queue with Simplified Java Executor Service

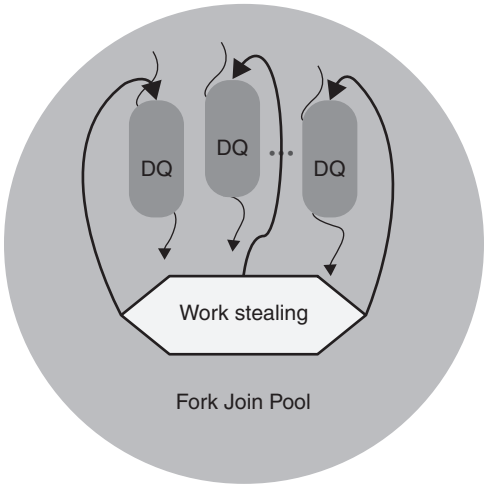

Figure 7.35 ForkJoinPool with Work-Stealing Algorithm

Introduced in Java 7, the `ForkJoinPool` framework is designed to maximize the efficiency of programs that can be divided into smaller tasks for parallel execution. Each worker thread manages a double-ended queue (DQ), allowing for the dynamic execution of tasks. The framework uses a work-stealing algorithm, in which worker threads that run out of tasks can steal tasks from other busy threads' deques. This allows the `ForkJoinPool` to fully utilize multiple processors, significantly improving performance for certain tasks.

Figure 7.35 illustrates the Java `ForkJoinPool` in action. The ovals labeled "DQ" represent the individual deques of the worker threads. The curved arrows between the deques illustrate the work-stealing mechanism, a distinctive feature that allows idle threads to take on tasks from their peers' deques, thereby maintaining a balance of workload across the pool.

Let's enhance our learning by looking at a simple example:

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

class IncrementAction extends RecursiveAction {
    private final ReservationCounter counter;
    private final int incrementCount;

    IncrementAction(ReservationCounter counter, int incrementCount) {
        this.counter = counter;
        this.incrementCount = incrementCount;
    }

    @Override
    protected void compute() {
        // If the increment count is small, increment the counter directly.
        // This is the base case for the recursion.
        if (incrementCount <= 100) {
            for (int i = 0; i < incrementCount; i++) {
                counter.increment();
            }
        } else {
            // If the increment count is larger, split the task into two smaller tasks.
            // This is the recursive case.
            // The ForkJoinPool will execute these two tasks in parallel, if possible.
            int half = incrementCount / 2;
            IncrementAction firstHalf = new IncrementAction(counter, half);
            IncrementAction secondHalf = new IncrementAction(counter, incrementCount - half);
            invokeAll(firstHalf, secondHalf);
        }
    }
}

class ForkJoinLockLoop {
    public static void main(String[] args) throws InterruptedException {
        ReservationCounter counter = new ReservationCounter();
        ForkJoinPool pool = new ForkJoinPool();

        // Submit the IncrementAction to the ForkJoinPool.
        // The pool handles dividing the task into smaller tasks and their parallel execution.
        pool.invoke(new IncrementAction(counter, 1000));
```

```
        System.out.println("Final count: " + counter.getCount());
    }
}
```

This code example demonstrates how the `ForkJoinPool` can be used to execute an `IncrementAction`, a `RecursiveAction` that increments the counter. Depending on the increment count, it either increments the counter directly or splits the task into smaller tasks for recursive invocation.

Java CompletableFuture

Before diving into virtual threads, it's essential to understand another powerful tool in Java's concurrency toolkit: `CompletableFuture`. Introduced in Java 8, the `CompletableFuture` framework represents a future computation that can be concluded in a controlled manner, thus offering a rich API for composing and combining tasks in a functional style.

One of `CompletableFuture`'s standout features is its ability to handle asynchronous computation. Unlike `Future`, which represents the result of a computation that may not have completed yet, `CompletableFuture` can be manually completed at any time. This makes it a powerful tool for handling tasks that may be computed asynchronously.

`CompletableFuture` also integrates seamlessly with the Java Streams API, allowing for complex data transformations and computations, contributing to more readable and maintainable concurrent code patterns. Moreover, it provides robust error-handling capabilities. Developers can specify actions to be taken in case a computation completes exceptionally, simplifying error handling in a concurrent context.

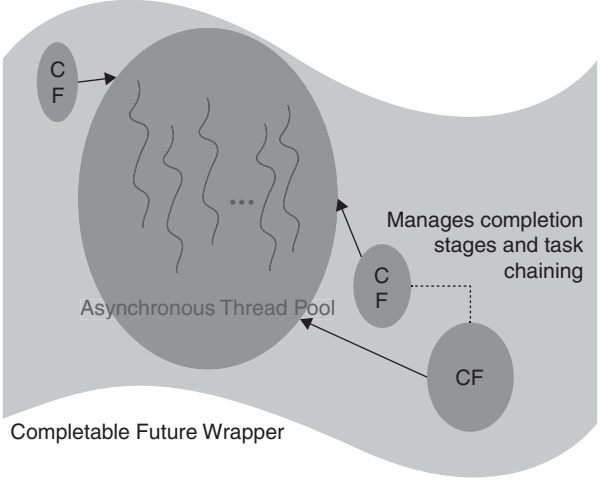

Figure 7.36 CompletableFuture Task Management and Execution

Under the hood, `CompletableFuture` uses a wrapper on top of standard thread pools, offering a higher abstraction level for managing and controlling tasks. Figure 7.36 depicts this

abstraction by demonstrating how individual `CompletableFuture` instances, denoted as "CF," are managed. These instances are executed within an asynchronous thread pool, embodying the concurrency principle where tasks are run in parallel, independent of the main application flow. The diagram also shows how `CompletableFuture` instances can be chained, leveraging their API to manage completion stages and task sequences, thus illustrating the framework's compositional prowess.

Here's a simple example of how you might use `CompletableFuture` in the context of the `ReservationService`:

```
class ReservationService {
    private final ReservationCounter hotelCounter = new ReservationCounter();
    private final ReservationCounter flightCounter = new ReservationCounter();
    private final ReservationCounter carRentalCounter = new ReservationCounter();

    void incrementCounters() {
        CompletableFuture<Void> hotelFuture =
                CompletableFuture.runAsync(new DoActivity(hotelCounter));
        CompletableFuture<Void> flightFuture =
                CompletableFuture.runAsync(new DoActivity(flightCounter));
        CompletableFuture<Void> carRentalFuture =
                CompletableFuture.runAsync(new DoActivity(carRentalCounter));

        // Combine the futures
        CompletableFuture<Void> combinedFuture =
                CompletableFuture.allOf(hotelFuture, flightFuture, carRentalFuture);

        // Block until all futures are complete
        combinedFuture.join();
    }
}
```

In this example, we create a `CompletableFuture` for each `DoActivity` task. We then use `CompletableFuture.allOf` to combine these futures into a single `CompletableFuture`. This combined future completes when all of the original futures have completed. By calling `join` on the combined future, we block until all of the increment operations are complete. Thus, this example demonstrates how `CompletableFuture` can be used to manage and coordinate multiple asynchronous tasks.

In the next section, we'll see how virtual threads build on these concepts to reimagine concurrency in Java. This approach promotes the use of maintainable blocking (i.e., synchronous) code while fully leveraging the features of the Java language and platform.

Reimagining Concurrency with Virtual Threads

Traditionally, threads, which are Java's concurrency unit, have run concurrently and sequentially. However, the typical model of mapping Java threads to OS threads has limitations. This model becomes particularly problematic when dealing with blocking operations and thread pools.

Project Loom,[17] introduced in JDK 21,[18] aims to reimagine concurrency in Java. It introduces virtual threads, which are lightweight, efficient threads managed by the JVM that allow Java to achieve better scalability without resorting to the asynchronous or reactive approach.

Virtual Threads in Project Loom

Virtual threads, also known as lightweight threads, are a new kind of Java thread. Unlike traditional threads, which are managed by the OS, virtual threads are managed by the JVM. This allows the JVM to create and switch between virtual threads much more efficiently than the OS can with traditional threads. Virtual threads are designed to have a low memory footprint and fast start-up time, making them ideal for tasks that demand intense concurrency.

Let's build on the previous diagrams to further our understanding and establish a visual relationship. Figure 7.37 is a high-level overview diagram of the new model introduced by Project Loom. In this model, a large number of Java virtual threads (jvt0, jvt1, jvtn) can be created within a single JVM process. These virtual threads are not directly mapped to OS threads. Instead, they are managed by the JVM, which can schedule them onto a smaller number of OS threads as needed.

Figure 7.37 also introduces the concept of carrier threads. Each virtual thread is scheduled onto a carrier thread, which is a traditional Java thread mapped one-to-one to an OS thread. The JVM can switch a carrier thread between multiple virtual threads, allowing it to manage multiple virtual threads concurrently. This model also highlights the work-stealing algorithm, in which idle carrier threads can steal tasks from the queues of busy carrier threads, leading to better utilization of system resources.

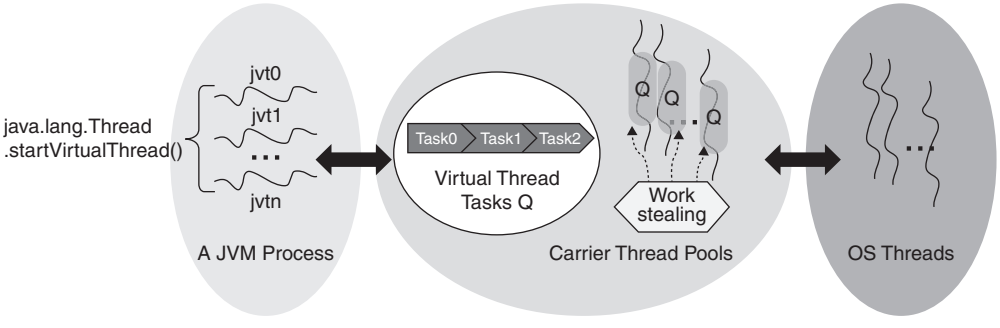

Figure 7.37 Java Virtual Threads

Virtual Threads and Their Carriers

A virtual thread's life cycle may involve being scheduled onto different carrier threads. Despite this, the virtual thread remains oblivious to the specific carrier thread it's scheduled on at any given moment.

[17] https://openjdk.org/jeps/444

[18] www.infoq.com/news/2023/04/virtual-threads-arrives-jdk21/

A crucial aspect to understand is the separation of stack traces between the carrier thread and the virtual thread. If an exception is thrown within a virtual thread, the resulting stack trace will not include any frames from the carrier thread. This separation is also evident when generating thread dumps: The stack frames of a virtual thread and its carrier thread are kept distinct and do not intermingle.

Similarly, thread-local variables maintain this separation. Any thread-local variables associated with the carrier thread are not accessible to the virtual thread, and vice versa. This ensures that the state of the carrier thread and the virtual thread remain independent, further emphasizing the encapsulation of virtual threads.

NOTE JEP 444: *Virtual Threads*[19] explicitly states that virtual threads should never be pooled. The JVM's scheduler takes care of that step for you.

Integration with Virtual Threads and Existing APIs

One of the key goals of Project Loom is to ensure that virtual threads can be integrated seamlessly with existing APIs. In consequence, developers can utilize virtual threads with existing libraries and frameworks without the need for significant code modifications.

In traditional Java concurrency, the `ExecutorService` API has been a cornerstone for managing threads. With Project Loom, this API has been enhanced to work harmoniously with virtual threads. For instance, in our `ReservationService` example, tasks were submitted to the `ExecutorService`, which then took care of the intricacies of creating and scheduling virtual threads. This seamless integration extends to other APIs like `CompletableFuture`, which now supports the `runAsync` method with virtual threads and is backward compatible, allowing developers to use virtual threads without altering legacy code. Such integration underscores the ease with which developers can adopt virtual threads in their applications, reaping the benefits of improved scalability and performance without the need to familiarize themselves with new APIs or overhaul existing code.

Practical Example Using Virtual Threads

To further illustrate the seamless integration of virtual threads, let's consider a simple `ReservationCounter`:

```
class ReservationService {
    private final ReservationCounter hotelCounter = new ReservationCounter();
    private final ReservationCounter flightCounter = new ReservationCounter();
    private final ReservationCounter carRentalCounter = new ReservationCounter();

    void incrementCounters() {
        try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
```

[19] https://openjdk.org/jeps/444

```
            executor.submit(new DoActivity(hotelCounter));
            executor.submit(new DoActivity(flightCounter));
            executor.submit(new DoActivity(carRentalCounter));
        }
    }
}

class ReservationCounter {
    private int count = 0;

    synchronized void increment() {
        count++;
    }

    synchronized int getCount() {
        return count;
    }
}

class DoActivity implements Runnable {
    private final ReservationCounter counter;

    DoActivity(ReservationCounter counter) {
        this.counter = counter;
    }

    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            counter.increment();
        }
    }
}

class LoomLockLoop {
    public static void main(String[] args) throws InterruptedException {
        ReservationService service = new ReservationService();
        service.incrementCounters();

        System.out.println("Final hotel count: " + service.hotelCounter.getCount());
        System.out.println("Final flight count: " + service.flightCounter.getCount());
        System.out.println("Final car rental count: " + service.carRentalCounter.getCount());
    }
}
```

In this example, the `DoActivity` class is a `Runnable` that increments the `ReservationCounter` 1000 times. We have introduced a `ReservationService` that manages three different counters—one for hotel reservations, one for flight reservations, and one for car rental reservations. In the `incrementCounters` method, we create a new `DoActivity` for each counter and submit it to the `ExecutorService`. Each submission creates a new virtual thread that executes the `DoActivity`'s `run` method. This allows the increment operations for the three counters to be performed concurrently, demonstrating the efficiency and scalability of virtual threads in handling multiple concurrent tasks.

Parallelism with Virtual Threads

Virtual threads offer a significant advantage when the goal is to efficiently manage parallel tasks. As demonstrated in the previous example, we can create a large number of virtual threads to perform tasks concurrently. However, the true power of virtual threads lies in the JVM's ability to manage these threads and schedule them onto carrier threads for execution.

The parallelism of the virtual thread scheduler is determined by the number of carrier threads available for scheduling virtual threads. Out of the box, the value matches the number of available processors, but it can be tuned with the system property `jdk.virtualThreadScheduler.parallelism`. Thus, even though we might have a large number of virtual threads, the actual number of OS threads used is limited, allowing for efficient resource utilization.

To illustrate this approach, let's enhance our `ReservationCounter` example to use virtual threads for each activity and perform the increment operations in parallel:

```
class ReservationService {
    private final ReservationCounter hotelCounter = new ReservationCounter();
    private final ReservationCounter flightCounter = new ReservationCounter();
    private final ReservationCounter carRentalCounter = new ReservationCounter();

    void incrementCounters() {
        try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
            CompletableFuture<Void> hotelFuture = CompletableFuture.runAsync
(new DoActivity(hotelCounter), executor);
            CompletableFuture<Void> flightFuture = CompletableFuture.runAsync
(new DoActivity(flightCounter), executor);
            CompletableFuture<Void> carRentalFuture = CompletableFuture.runAsync
(new DoActivity(carRentalCounter), executor);

            // Wait for all operations to complete
            CompletableFuture.allOf(hotelFuture, flightFuture, carRentalFuture).join();
        }
    }
}

// ... rest of the classes remain the same ...
```

```
class LoomLockLoop {
    public static void main(String[] args) throws InterruptedException {
        ReservationService service = new ReservationService();
        service.incrementCounters();

        System.out.println("Final hotel count: " + service.hotelCounter.getCount());
        System.out.println("Final flight count: " + service.flightCounter.getCount());
        System.out.println("Final car rental count: " + service.carRentalCounter.getCount());
    }
}
```

In this version, we use `CompletableFuture.runAsync` (showcased in the section on the Java `CompletableFuture` framework) to submit the `DoActivity` tasks to the executor. This returns a `CompletableFuture` that completes when the task is done. We then use `CompletableFuture.allOf` to create a new `CompletableFuture` that completes when all of the given futures complete. This allows us to wait for all of the increment operations to finish before we print the final counts, demonstrating how virtual threads can be used to efficiently handle parallel tasks.

Continuations: The Underpinning of Virtual Threads

Under the hood, the implementation of virtual threads in Project Loom is based on the concept of *continuations*. A continuation represents a computation that can be suspended and resumed. It's essentially a saved stack frame that can be reloaded when the computation is resumed.

This ability to suspend and resume computations is what allows virtual threads to be so lightweight. Unlike traditional threads, which consume resources even when they're blocked, virtual threads consume resources only when they're actually doing work.

Continuations are a powerful concept, but they're mostly hidden from developers. When you're working with virtual threads, you don't need to worry about managing continuations yourself. The JVM takes care of all the details for you.

In our `ReservationService` example, each `DoActivity` task could potentially be suspended and resumed multiple times as it's executed by the `ExecutorService`. This is because each `DoActivity` task is run on a virtual thread, and when a virtual thread is blocked (for example, when waiting for a lock on the `ReservationCounter`), the JVM saves its current state as a continuation and switches to executing another virtual thread. When the blocking operation is complete, the JVM can resume the suspended virtual thread by reloading its continuation.

Conclusion

In this chapter, we journeyed through the intricate landscape of JVM runtime performance optimizations, shedding light on various facets from strings to locks. We began by demystifying string optimizations, including the intricacies of the string pool, string interning, and the transformative impact of string deduplication. We then moved on to the topic of locks, their role in synchronization.

Next, we delved into the realm of concurrency, discussing the traditional model of concurrency and its inherent challenges, and the innovative solutions provided by the Executor Service and the `ForkJoinPool` framework. These tools offer a fresh perspective on thread management and concurrent task execution. Furthermore, the `CompletableFuture` framework emerged as a beacon for handling asynchronous computations, embracing the elegance of functional programming. Finally, we explored the concepts of virtual threads and continuations with Project Loom. This initiative signifies a turning point in Java concurrency, promising greater scalability and efficiency in managing lightweight threads within the JVM.

As we look to the future, we can expect to see further advancements in JVM performance optimizations. For performance engineers, it's crucial to stay abreast of these developments and understand how to leverage them to build better software.

In the end, performance engineering is about making informed decisions. It's about understanding the trade-offs, knowing the tools and techniques at your disposal, and applying them judiciously to meet your performance goals. This chapter aimed to equip you with a deeper understanding of these elements, and I trust it serves as a valuable resource in your ongoing journey as a performance engineer.

This page intentionally left blank

Chapter 8

Accelerating Time to Steady State with OpenJDK HotSpot VM

Introduction

Start-up and warm-up/ramp-up performance optimization is a key factor in Java application performance, particularly for transient applications and microservices. The objective is to minimize the JVM's start-up and warm-up time, thereby accelerating the execution of the application's main method. Inefficient start-up or warm-up performance can lead to prolonged response times and increased resource consumption, which can negatively impact the user's interaction with the application. Therefore, it's crucial to focus on this aspect of Java application performance.

While the terms *warm-up* and *ramp-up* are often used interchangeably, by understanding their nuances, we can better appreciate the intricacies of JVM start-up and warm-up performance. So, it's essential to distinguish between them:

- **Warm-up:** In the context of Java applications and the JVM, warm-up typically refers to the time it takes for the JVM system to gather enough profiling data to optimize the code effectively. This is particularly crucial for JIT compilation because it relies on runtime behavior to make its optimizations.
- **Ramp-up:** Ramp-up generally denotes the time it takes for the application system to reach its full operational capacity or performance. This could encompass not just code optimization, but other factors such as resource allocation, system stabilization, and more.

In this chapter, we first explore the intricate details of JVM start-up and warm-up performance, beginning with a thorough understanding of the processes and their various phases. We then explore the steady-state of a Java application, focusing on how the JVM manages this state. As we navigate through these topics, we will discuss a variety of techniques and advancements that can help us achieve faster start-up and warm-up times. Whether you're developing a

microservices architecture, deploying your application in a containerized environment, or working with a traditional server-based environment, the principles and techniques of JVM start-up and warm-up optimization remain consistent.

JVM Start-up and Warm-up Optimization Techniques

The JIT compiler has seen continuous improvements, significantly benefiting JVM start-up and ramp-up (which includes warm-up) optimizations. Enhancements in bytecode interpretation have led to improved ramp-up performance. Additionally, Class Data Sharing optimizes start-up by preprocessing class files, facilitating faster loading and sharing of data among JVM processes. This reduces both the time and the memory required for class loading, thereby improving start-up performance.

The transition from the Permanent Generation (PermGen) to Metaspace in JDK 8 has positively impacted ramp-up times. Unlike PermGen, which had a fixed maximum size, Metaspace dynamically adjusts its size based on the application's runtime requirements. This flexibility can lead to more efficient memory usage and faster ramp-up times.

Likewise, the introduction of code cache segmentation in JDK 9 has played a significant role in enhancing ramp-up performance. It enables the JVM to maintain separate code caches for non-method, profiled, and non-profiled code, improving the organization and retrieval of compiled code and further boosting performance.

Going forward, Project Leyden aims to reduce Java's start-up time on HotSpot VM, focusing on static images for efficiency. Broadening the horizon beyond HotSpot, we have a major player in GraalVM, which emphasizes native images for faster start-up and reduced memory footprint.

Decoding Time to Steady State in Java Applications

Ready, Set, Start up!

In the context of Java/JVM applications, start-up refers to the critical process that commences with the invocation of the JVM and is thought to culminate when the application's main method starts executing. This intricate process encompasses several phases—namely, JVM bootstrapping, bytecode loading, and bytecode verification and preparation. The efficiency and speed of these operations significantly influences the application's start-up performance. However, as rightly pointed out by my colleague Ludovic Henry, this definition can be too restrictive.

In a broader sense, the start-up process extends beyond the execution of the `Application.main` method—it encompasses the entire initialization phase until the application starts serving its primary purpose. For instance, for a web service, the start-up process is not just limited to the JVM's initialization but continues until the service starts handling requests.

This extended start-up phase involves the application's own initialization processes, such as parsing command-line arguments or configurations, including setting up sockets, files, and other resources. These tasks can trigger additional class loading and might even lead to JIT

compilation and flow into the ramp-up phase. The efficiency and speed of these operations play a pivotal role in determining the application's overall start-up performance.

Phases of JVM Start-up

Let's explore the specific phases within the JVM's start-up process in more detail.

JVM Bootstrapping

This foundational phase focuses on kick-starting the JVM and setting up the runtime environment. The tasks performed during this phase include parsing command-line arguments, loading the JVM code into memory, initializing internal JVM data structures, setting up memory management, and establishing the initial Java thread. This phase is indispensable for the successful execution of the application.

For instance, consider the command `java OnlineLearningPlatform`. This command initiates the JVM and invokes the `OnlineLearningPlatform` class, marking the beginning of the JVM bootstrapping phase. The `OnlineLearningPlatform`, in this context, represents a web service designed to provide online learning resources.

Bytecode Loading

During this phase, the JVM loads the requisite Java classes needed to start the application. This includes loading the class containing the application's `main` method, as well as any other classes referenced during the execution of the `main` method, including system classes from the standard libraries. The class loader performs class loading by reading the `.class` files from the classpath and converting the bytecode in these files into a format executable by the JVM.

So, for example, for the `OnlineLearningPlatform` class, we might have a `Student` class that is loaded during this phase.

Bytecode Verification and Preparation

Following class loading, the JVM verifies that the loaded bytecode has the proper formatting and adheres to Java's type system. It also prepares the classes for execution by allocating memory for class variables, initializing them to default values, and executing any static initialization blocks. The Java Language Specification (JLS) strictly defines this process to ensure consistency across different JVM implementations.

For instance, the `OnlineLearningPlatform` class might have a `Course` class that has a static initializer block that initializes the `courseDetails` map. This initialization happens during the bytecode verification and preparation phase.

```
public class Course {
    private static Map<String, String> courseDetails;

    static {
   // Static initialization of course details
        courseDetails = new HashMap<>();
        courseDetails.put("Math101", "Basic Mathematics");
```

```
        courseDetails.put("Eng101", "English Literature");
        // Additional courses
    }

    // Additional methods
}
```

The start-up phase is critical because it directly impacts both user experience and system resource utilization. Slow start-ups can lead to subpar user experiences, particularly in interactive applications where users are waiting for the application to start. Additionally, they can result in inefficient use of system resources because the JVM and the underlying system need to do a lot of work during start-up.

Reaching the Application's Steady State

Beyond the start up phase, a Java application's life cycle includes two other significant phases: the ramp-up phase and the steady-state phase. Once the application starts its primary function, it enters the ramp-up phase, which is characterized by a gradual improvement in the application's performance until it reaches peak performance. The steady-state phase occurs when the application executes its main workload at peak performance.

Let's take a closer look at the ramp-up phase, during which the application performs several tasks:

- **Interpreter and JIT compilation:** Initially, the JVM interprets the bytecode of the methods into native code. These methods are derived from the loaded and initialized classes. The JVM then executes this native code. As the application continues to run, the JVM identifies frequently executed code paths (hot paths). These paths are then JIT compiled for optimized performance. The JIT compiler employs various optimization techniques to generate efficient native code, which can significantly improve the performance of the application.
- **Class loading for specific tasks:** As the application begins its primary tasks, it may need to load additional classes specific to these operations.
- **Cache warming:** Both the application (if utilized) and the JVM have caches that store frequently accessed data. During ramp-up, these caches are populated or "warmed up" with data that's frequently accessed, ensuring faster data retrieval in subsequent operations.

Let's try to put this in perspective: Suppose that along with our `Course` class, we have an `Instructor` class that is loaded after our web service initializes and obtains its up-to-date instructors list. For that `Instructor` class, the `teach` method could be one of the many methods that the JIT compiler optimizes.

Now take a look at the class diagram in Figure 8.1, which illustrates the `OnlineLearningPlatform` web service with `Student`, `Course`, and `Instructor` classes and their relationships. The `OnlineLearningPlatform` class is the main class that is invoked during the JVM bootstrapping phase. The `Student` class might be loaded during the bytecode-loading phase. The `Course` class, with its static initializer block, illustrates the

bytecode verification and preparation phase. The `Instructor` class is loaded during the ramp-up phase, and the `teach` and `getCourseDetails` methods may reach different levels of optimizations during ramp-up. Once they reach peak performance, the web service is ready for steady-state performance.

The phase chart in Figure 8.2 shows the JIT compilation times, the class loader times and the garbage collection (GC) times as the application goes through the various phases of its life cycle. Let's study these phases further and build an application timeline.

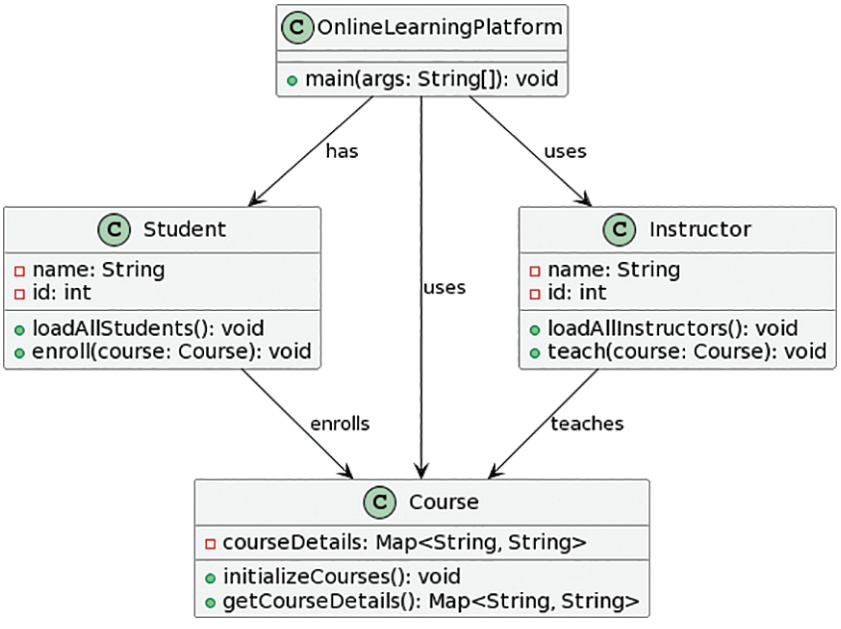

Figure 8.1 Class Diagram for the OnlineLearningPlatform Web Service

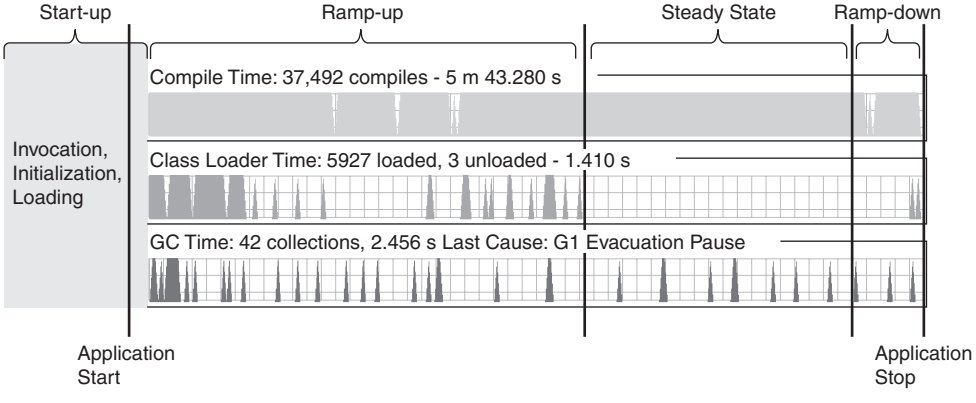

Figure 8.2 Application-Phase Chart Showing Garbage Collection, Compilation, and Class-Loading Timelines

An Application's Life Cycle

For most well-designed applications, we will see a life cycle that goes though the following phases (as shown in Figure 8.3):

- **Start-up:** The JVM is invoked and starts transitioning to the next phase after the application initializes its state. The start-up phase includes JVM initialization, class loading, class initialization, and application initialization work.
- **Ramp-up:** This phase is characterized by a gradual improvement in code generation, leading to increases in the application's performance until it reaches peak performance.
- **Steady-state:** In this phase, the application executes its main workload at peak performance.
- **Ramp-down:** In this phase, the application begins to shut down and release resources.
- **Application stop:** The JVM terminates, and the application stops.

The timeline in Figure 8.3 represents the progression of time from left to right. The application starts after the application completes initialization and then begins to ramp up. The application reaches its steady-state after the ramp-up phase. When the application is ready to stop, it enters the ramp-down phase, and finally the application stops.

Managing State at Start-up and Ramp-up

State management is a pivotal aspect of Java application performance, especially during the start-up and ramp-up phases. The state of an application encompasses various elements, including data, sockets, files, and more. Properly managing this state ensures that the application will run efficiently and lays the foundation for its subsequent operations.

State During Start-up

The state at start-up is a comprehensive snapshot of all the variables, objects, and resources with which the application is interacting at that moment:

- **Static variables:** Variables that retain their values even after the method that created them has finished executing.
- **Heap objects:** Objects that are dynamically allocated during runtime and that reside in a heap region of memory.

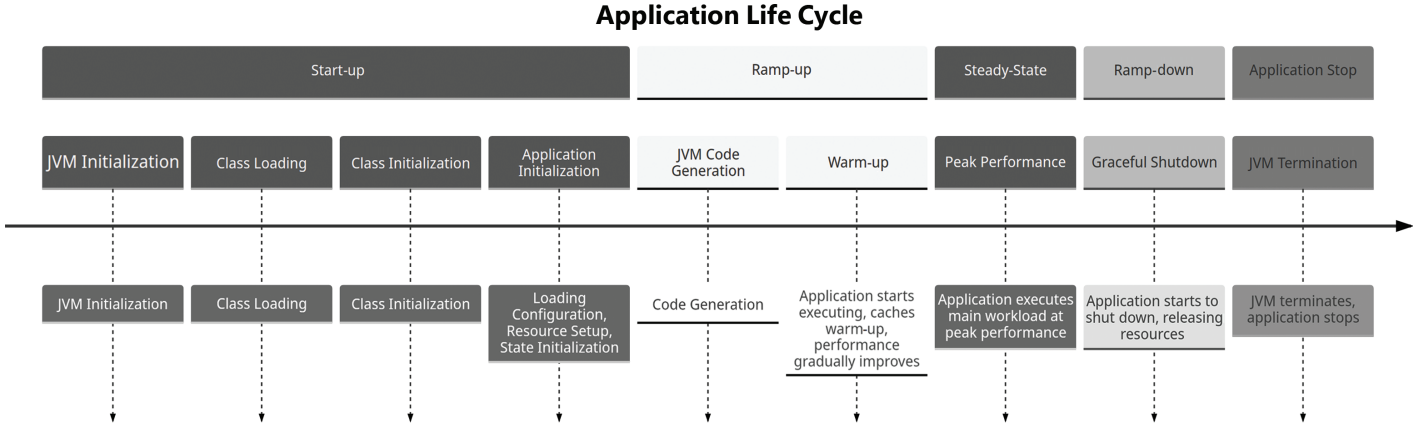

Figure 8.3 An Application Life Cycle

- **Running threads:** The sequences of programmed instructions that are executed by the application.
- **Resources:** File descriptors, sockets, and other resources that the application uses.

The flowchart in Figure 8.4 summarizes the initialization of the application's state, from loading of the static variables to the allocation of resources, and the transition through ramp-up to the steady-state.

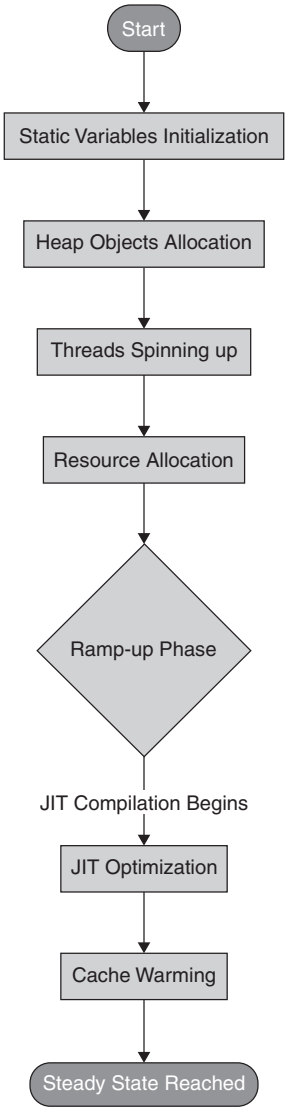

Figure 8.4 Application State Management Flowchart

Transition to Ramp-up and Steady State

Following the start-up phase, the application enters the ramp-up phase, where it starts processing real-world tasks. During this phase, the JIT compiler refines the code based on actual usage patterns. The culmination of the ramp-up phase is marked by the application achieving a steady-state. At this point, performance stabilizes, and the JIT compiler has optimized the majority of the application's code.

As shown in Figure 8.5, the duration from start-up to the steady-state is termed the "time to steady-state." This metric is crucial for gauging the performance of applications, especially those with short lifespans or those operating in serverless environments. In such scenarios, swift start-ups and efficient workload ramp-ups are paramount.

Together, Figure 8.4 and Figure 8.5 provide a visual representation of the application's life cycle, from the start-up phase through to achieving a steady-state, highlighting the importance of efficient state management.

Benefits of Efficient State Management

Proper state management during the start-up and ramp-up phase can reduce the time to steady-state, resulting in the following benefits:

- **Improved performance:** Spending less time on initialization and loading data structures expedites the application's transition to the steady state, thereby enhancing performance.
- **Optimized memory usage:** Efficient state management reduces the amount of memory required to store the application's state, leading to a smaller memory footprint.
- **Better resource utilization:** At the hardware and operating system (OS) levels, streamlined state management can lead to better utilization of the CPU and memory resources and reduce the load on the OS's file and network systems.
- **Containerized environments:** In environments where applications run in containers, state management is even more critical. In these scenarios, resources are often more constrained, and the overhead of the container runtime needs to be accomodated.

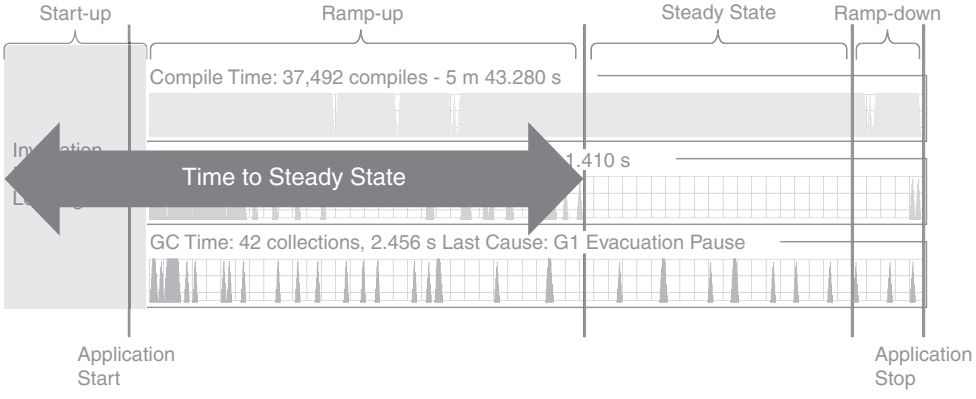

Figure 8.5 An Application Timeline Depicting Time to Steady State

In the current HotSpot VM landscape, one technique stands out for its ability to manage state and improve time-to-steady-state performance: Class Data Sharing.

Class Data Sharing

Class Data Sharing (CDS), a powerful JVM feature, is designed to significantly enhance the start-up performance of Java applications. It accomplishes this feat by preprocessing class files and converting them into an internal format that can be shared among multiple JVM instances. This feature is particularly beneficial for extensive applications that utilize a multitude of classes, as it reduces the time taken to load these classes, thereby accelerating start-up times.

The Anatomy of the Shared Archive File

The shared archive file created by CDS is an efficient and organized data structure that contains several *shared spaces*. Each of these spaces is dedicated to storing a different type of data. For instance,

- ***MiscData* space:** This space is reserved for various metadata types.
- ***ReadWrite* and *ReadOnly* spaces:** These spaces are allocated for the actual class data.

Such a structured approach optimizes data retrieval and streamlines the loading process, contributing to faster start-up times.

Memory Mapping and Direct Usage

Upon JVM initiation, the shared archive file is memory-mapped into its process space. Given that the class data within the shared archive is already in the JVM's internal format, it's directly usable, eliminating the need for any conversions. This direct access further trims the start-up time.

Multi-instance Benefits

The benefits of CDS extend beyond a single JVM instance. The shared archive file can be memory-mapped into multiple JVM processes running on the same machine. This mechanism of memory mapping and sharing is particularly beneficial in cloud environments, where resource constraints and cost-effectiveness are primary concerns. Allowing multiple JVM instances to share the same physical memory pages reduces the overall memory footprint, leading to enhanced resource utilization and cost savings. Furthermore, this sharing mechanism improves the start-up time of subsequent JVM instances, as the shared archive file has already been loaded into memory by the first JVM instance. This proves invaluable in serverless environments where rapid function start-ups in response to events are essential.

Dynamic Dumping with -XX:ArchiveClassesAtExit

The `-XX:ArchiveClassesAtExit` option allows the JVM to dynamically dump the classes loaded by the application to a shared archive file. By preloading the classes that are actually used by the application during the next JVM start, this feature fine-tunes start-up performance.

Using CDS involves the following steps:

1. Generate a shared archive file that contains the preprocessed class data. This is done using the `-Xshare:dump` JVM option.
2. Once the archive file is generated, it can be used by JVM instances using the `-Xshare:on` and `-XX:SharedArchiveFile` options.

For example:

```
java -Xshare:dump -XX:SharedArchiveFile=my_app_cds.jsa -cp my_app.jar
java -Xshare:on -XX:SharedArchiveFile=my_app_cds.jsa -cp my_app.jar MyMainClass
```

In this example, the first command generates a shared archive file for the classes used by `my_app.jar`, and the second command runs the application using the shared archive file.

Ahead-of-Time Compilation

Ahead-of-time (AOT) compilation is a technique that plays a significant role in managing state and improving the time-to-peak performance in the managed runtimes. Introduced in JDK 9, AOT aimed to enhance start-up performance by compiling methods to native code before the JVM was invoked. This precompilation meant that the native code was ready beforehand and eliminated the need to wait for the interpretation of the bytecode. This approach is particularly beneficial for short-lived applications where the JIT compiler might not have enough time to effectively optimize the code.

NOTE Before we get into the intricacies of AOT, it's crucial to understand its counterpart, JIT compilation. In the HotSpot VM, the interpreter is the first to execute, but it doesn't optimize the code. The JIT-ing process brings in the optimizations, but it requires time to gather profiling data for frequently executed methods and loops. As a result, there's a delay, especially for code that needs an optimized state for its efficient execution. Several components contribute to transitioning the code to its optimized state. For instance, the (segmented) code cache maintains the JIT code in its various states (for example, profiled and non-profiled). Additionally, compilation threads manage optimization and deoptimization states via adaptive and speculative optimizations. Notably, the HotSpot VM speculatively optimizes dynamic states, effectively converting them into static states.[1] These elements collectively support the transition of code from its initial interpreted state to its optimized form, which can be viewed as the overhead of JIT compilation.

[1] John Rose pointed out this behavior in his presentation at JVMLS 2023: https://cr.openjdk.org/~jrose/pres/202308-Leyden-JVMLS.pdf.

AOT introduced static compilation to dynamic languages. Prior to HotSpot VM, platforms like IBM's J9 and Excelsior JET had successfully employed AOT. Execution engines that rely on profile-guided techniques to provide adaptive optimizations can potentially suffer from slow start-up times due to the need for a warm-up phase. In HotSpot VM's case, the compilation always begins in interpreted mode. AOT compilation, with its precompiled methods and shared libraries, is a great solution to these warm-up challenges. With AOT at the forefront and the adaptive optimizer in the background, the application can achieve its peak performance in an expedited fashion.

In JDK 9, the AOT feature had experimental status and was built on top of the Graal compiler. Graal is a dynamic compiler that is written in Java, whereas HotSpot VM is written in C++. To support Graal, HotSpot VM needed to support a new JVM compiler interface, called JVMCI.[2] A first pass at this effort was completed for a few known libraries such as `java.base`, `jdk.compiler`, `jdk.vm.compiler`, and a few others.

The AOT code added to JDK 9[3] worked at two levels—tiered and non-tiered, where non-tiered was the default mode of execution and was geared toward increasing application responsiveness. The tiered mode was akin to the T2 level of HotSpot VM's tiered compilation, which is the client compiler with profiling information. After that, the methods that crossed the AOT invocation thresholds were recompiled by the client compiler at the T3 level and full profiling was carried out on those methods, which eventually led to recompilation by the server compiler at the T4 level.

AOT compilation could be used in conjunction with CDS to further improve start-up performance. The CDS feature allows the JVM to share common class metadata across different Java processes, while AOT compilation allowed the JVM to use precompiled code instead of interpreting bytecode. When used together, these features could significantly reduce start-up time and memory footprint.

However, AOT compilers (within the context of the OpenJDK HotSpot VM) faced certain limitations. For instance, AOT-compiled code had to be stored in shared libraries, necessitating the use of position-independent code (PIC).[4] This requirement was due to the fact that the location where the code would be loaded into memory could not be predetermined, preventing the AOT compiler from making location-based optimizations. In contrast, the JIT compiler is allowed to make more assumptions about the execution environment and optimizes the code accordingly.

The JIT compiler can directly reference symbols, eliminating the need for indirections and resulting in more-efficient code. However, these optimizations by the JIT compiler increase start-up time, as the compilation process happens during application execution. By comparison, an interpreter, which executes bytecode directly, doesn't need to spend time compiling the code, but the execution speed of interpreted code is generally much slower than that of compiled code.

---

[2] https://openjdk.org/jeps/243

[3] https://openjdk.org/jeps/295

[4] https://docs.oracle.com/cd/E26505_01/html/E26506/glmqp.html

NOTE It is important to note that different virtual machines, such as GraalVM, implement AOT compilation in ways that can overcome some of these limitations. In particular, GraalVM's AOT compilation enhanced with profile-guided optimizations (PGO) can achieve peak performance comparable to JIT[5] and even allows for certain optimizations that take advantage of a closed-world assumption, which are not possible in a JIT setting. Additionally, GraalVM's AOT compiler can compile 100% of the code—which could be an advantage over JIT, where some "cold" code may remain interpreted.

Project Leyden: A New Dawn for Java Performance

While AOT promised performance gains, the complexity it added to the HotSpot VM overshadowed its benefits. The challenges of maintaining a separate AOT compiler, the additional maintenance burden due to its reliance on the Graal compiler, and the inefficiencies of generating PIC made it clear that a different approach was needed.[6] This led to the inception of Project Leyden.

The essence of Java application performance lies in understanding and managing the various states particularly through capturing snapshots of variables, objects, and resources, and ensuring they're efficiently utilized. This is where Project Leyden steps in. Introduced in JDK 17, Leyden aims to address the challenges of Java's slow start-up time, slow time-to-peak performance, and large footprint. It's not just about faster start-ups; it's about ensuring that the entire life cycle of a Java application, from start-up to steady-state, is optimized.

Training Run: Capturing Application Behavior for Optimal Performance

Project Leyden introduces the concept of a "training run," which is similar to the step in which we generate the archive of preprocessed classes for CDS. In a training run, we aim to capture all of the application's behavior, especially during the start-up and warm-up phases. This recorded behavior, encompassing method calls, resource allocations, and other runtime intricacies, is then harnessed to generate a custom runtime image. Tailored to the application's specific needs, this image ensures swifter start-ups and a quicker ascent to peak performance. It's Leyden's way of ensuring that Java applications are not just performing adequately, but optimally.

Leyden's precompiled runtime images mark a shift towards enhanced predictability in performance, which is particularly beneficial for applications where traditional JIT compilation might not fully optimize the code due to shorter runtimes. This advancement represents a leap in Java's evolution, streamlining the start-up process and enabling consistent performance.

---

[5] Alina Yurenko. "GraalVM for JDK 21 Is Here!" https://medium.com/graalvm/graalvm-for-jdk-21-is-here-ee01177dd12d#0df7.

[6] https://devblogs.microsoft.com/java/aot-compilation-in-hotspot-introduction/

Condensers: Bridging the Gap Between Code and Performance

Whereas CDS serves as the underlying backbone when managing states, Leyden introduces *condensers*—tools akin to the guardians of state management. Condensers ensure that computation, whether directly expressed by a program or done on its behalf, is shifted to the most optimal point in time. By doing so, they preserve the essence of the program while offering developers the flexibility to choose performance over certain functionalities.

Shifting Computation with Project Leyden: A Symphony of States

Java performance optimization relies on the meticulous management of states and their seamless transition from one phase to another. Leyden's approach to shifting computation is a testament to this philosophy. It's not just about moving work around; it's about ensuring that every piece of computation, every variable, and every resource is optimally placed and utilized. This is the art of driving both start-up and warm-up times into the background, making them almost negligible.

Consider our `OnlineLearningPlatform` web service. Prior to Leyden optimization, we could have something like this code snippet:

```
public class OnlineLearningPlatform {
    public static void main(String[] args) {
        // Initialization
        System.out.println("Initializing OnlineLearningPlatform...");

        // Load Student and Course classes
        Student.loadAllStudents();
        Course.initializeCourses();

        // Ramp-up phase: Load Instructor class after its up-to-date list
        Instructor.loadAllInstructors();
        Instructor.teach(new Course()); // This method is a candidate for JIT optimization
    }
}

public class Student {
    private String name;
    private int id;

    public static void loadAllStudents() {
        // Load all students
    }

    public void enroll(Course course) {
        // Enroll student in a course
    }
}
```

```
public class Course {
    private static Map<String, String> courseDetails;

    static {
        courseDetails = new HashMap<>();
        courseDetails.put("Math101", "Basic Mathematics");
        courseDetails.put("Eng101", "English Literature");
        // Additional courses
    }

    public static void initializeCourses() {
        // Initialize courses
    }

    public Map<String, String> getCourseDetails() {
        return courseDetails;
    }
}

public class Instructor {
    private String name;
    private int id;

    public static void loadAllInstructors() {
        // Load all instructors
    }

    public void teach(Course course) {
        // Teach a course
    }
}
```

In this setup, the initialization of `OnlineLearningPlatform`, the loading of the `Student` and `Course` classes, and the ramp-up phase with the `Instructor` class all happen sequentially, potentially leading to longer start-up times. However, with Leyden, we can have something like the following:

```
public class OnlineLearningPlatform {
    public static void main(String[] args) {
        // Initialization with Leyden's shifted computation
        System.out.println("Initializing OnlineLearningPlatform with Leyden optimizations...");

        // Load Student and Course classes from preprocessed images
        Student.loadAllStudentsFromImage();
        Course.initializeCoursesFromImage();
```

```
        // Ramp-up phase: Load Instructor class from preprocessed images and JIT optimizations
        Instructor.loadAllInstructorsFromImage();
        Instructor.teachOptimized();
    }
}

public class Student {
    private String name;
    private int id;

    public static void loadAllStudentsFromImage() {
        // Load all students from preprocessed image
    }

    public void enroll(Course course) {
        // Enroll student in a course
    }
}

public class Course {
    private static Map<String, String> courseDetails;

    static {
        // Load course details from preprocessed image
        courseDetails = ImageLoader.loadCourseDetails();
    }

    public static Map<String, String> initializeCoursesFromImage() {
        // Load course details from preprocessed image and return
        return courseDetails;
    }

    public Map<String, String> getCourseDetails() {
        return courseDetails;
    }
}

public class Instructor {
    private String name;
    private int id;

    public static void loadAllInstructorsFromImage() {
        // Load all instructors from preprocessed image
    }
```

```
    public void teachOptimized(Course course) {
        // Teach a course with JIT optimizations
    }

}
```

The notable shifts are as follows:

- Classes are now sourced from preprocessed images, evident from methods like `loadAllStudentsFromImage()`, `initializeCoursesFromImage()`, and `loadAllInstructorsFromImage()`.
- The static initializer block in the `Course` class now loads course details from a preprocessed image using the `ImageLoader.loadCourseDetails()` method.
- The `Instructor` class introduces `teachOptimized()`, which represents the JIT-optimized teaching method.

By harnessing Leyden's enhancements, the `OnlineLearningPlatform` can leverage preprocessed images for loading classes and JIT-optimized methods. The revised class diagram for our example post-Project Leyden is shown in Figure 8.6.

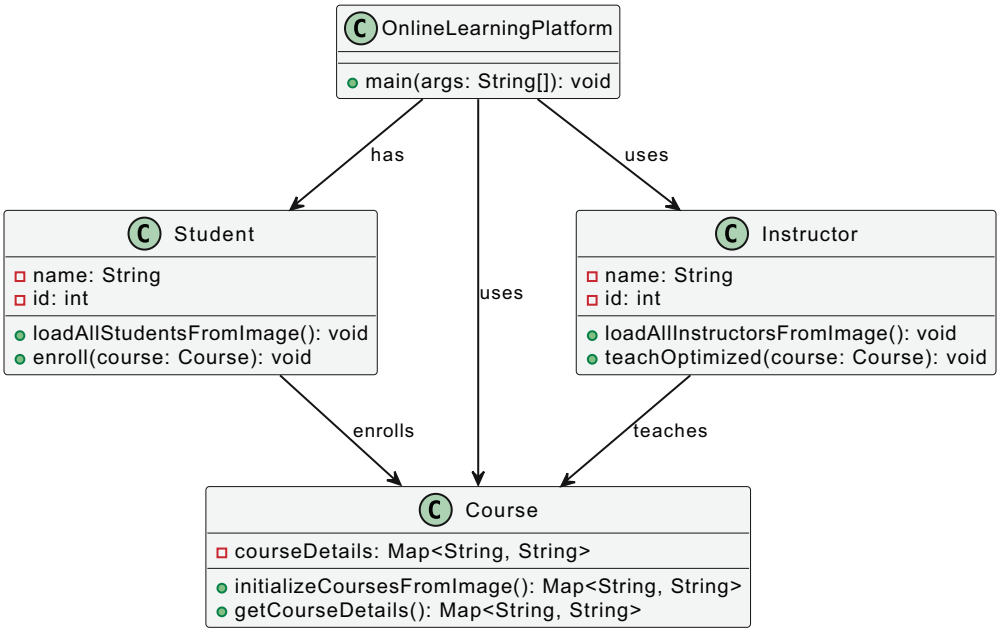

Figure 8.6 New Class Diagram: Post–Project Leyden Optimizations

Looking ahead, Leyden promises a harmonious blend of performance and adaptability. It's about giving developers the tools and the freedom to choose how they want to manage states, how they want to optimize performance, and how they want their applications to interact with the underlying JVM. From introducing condensers to launching the concept of a training run, Leyden is setting the stage for a future in which Java performance optimization is not just a technical endeavor but an art.

As the Java ecosystem continues to evolve, GraalVM emerges as a pivotal innovation, complementing the extension of performance optimization beyond Java's traditional boundaries.

GraalVM: Revolutionizing Java's Time to Steady State

Standing at the forefront of dynamic evolution, GraalVM focuses on optimizing start-up and warm-up performance by leveraging its capabilities and its proficiency in dynamic compilation. It harnesses the power of static images to enhance performance across a wide array of applications.

GraalVM is a polyglot virtual machine that seamlessly supports JVM languages such as Java, Scala, and Kotlin, as well as non-JVM languages such as Ruby, Python, and WebAssembly. Its true prowess lies in its ability to drastically enhance both start-up and warm-up times, ensuring applications not only initiate swiftly but also reach their optimal performance in a reduced time frame.

The heart of GraalVM is the Graal compiler (Figure 8.7). This dynamic compiler, tailored for dynamic or "open-world" execution, similar to the OpenJDK HotSpot VM, is adept at producing C2-grade machine code for a variety of processors. It employs "self-optimizing" techniques, leveraging dynamic and speculative optimizations to make informed predictions about program behavior. If a speculation misses the mark, the compiler can de-optimize and recompile, ensuring applications warm up rapidly and achieve peak performance sooner.

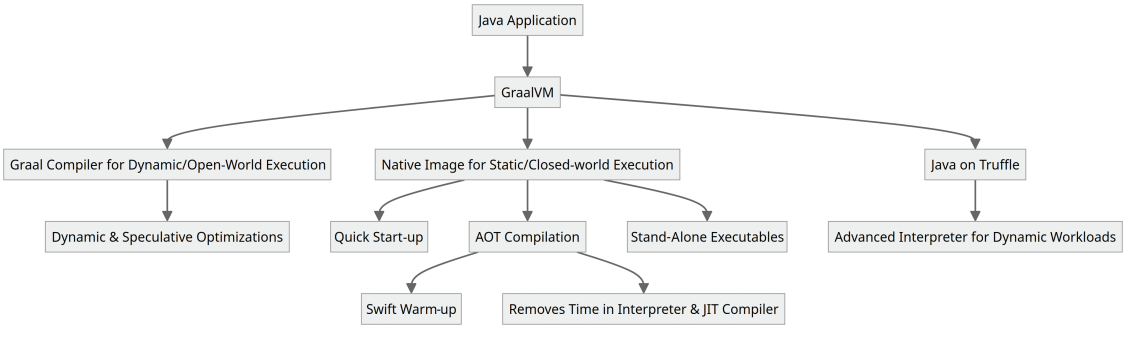

Figure 8.7 GraalVM Optimization Strategies

Beyond dynamic execution, GraalVM offers the power of native image generation for "closed world" optimization. In this scenario, the entire application landscape is known at compile

time. Thus, GraalVM can produce stand-alone executables that encapsulate the application, essential libraries, and a minimal runtime. This approach virtually eliminates time spent in the JVM's interpreter and JIT compiler, ushering in instantaneous start-up and swift warm-up times—a paradigm shift for short-lived applications and microservices.

Simplifying Native Image Generation

GraalVM's AOT compilation amplifies its performance credentials. By precompiling Java bytecode to native code, it minimizes the overhead of the JVM's interpreter during start-up. Furthermore, by sidestepping the traditional JIT compilation phase, applications experience a swifter warm-up, reaching peak performance in a shorter span of time. This is invaluable for architectures like microservices and serverless environments, for which both rapid start-up and quick warm-up are of the essence.

In recent versions of GraalVM, the process of creating native images has been simplified as the native image capability is now included within the GraalVM distribution itself.[7] This enhancement streamlines the process for developers:

```
# Install GraalVM (replace VERSION and OS with your GraalVM version and operating system)
$ curl -LJ https://github.com/graalvm/graalvm-ce-builds/releases/download/vm-VERSION/graalvm-ce-java11-VERSION-OS.tar.gz -o graalvm.tar.gz
$ tar -xzf graalvm.tar.gz

# Set the PATH to include the GraalVM bin directory
$ export PATH=$PWD/graalvm-ce-java11-VERSION-OS/bin:$PATH

# Compile a Java application to a native image
$ native-image -jar your-app.jar
```

Executing this command creates a precompiled, stand-alone executable of the Java application, which includes a minimal JVM and all necessary dependencies. The application can start executing immediately, without needing to wait for the JVM to start up or classes to be loaded and initialized.

Enhancing Java Performance with OpenJDK Support

The OpenJDK community has been working on improving the support for *native-image* building, making it easier for developers to create native images of their Java applications. This includes improvements in areas such as class initialization, heap serialization, and service binding, which contribute to faster start-up times and smaller memory footprints. By providing the ability to compile Java applications to native executables, GraalVM makes Java a more attractive option for high-demand computing environments.

[7] https://github.com/oracle/graal/pull/5995

Introducing Java on Truffle

To provide a complete picture of the execution modes supported by GraalVM, it's important to mention Java on Truffle.[8] This is an advanced Java interpreter that runs atop the Truffle framework and can be enabled using the `-truffle` flag when executing Java programs:

```
# Execute Java with the Truffle interpreter
$ java -truffle [options] class
```

This feature complements the existing JIT and AOT compilation modes, offering developers an additional method for executing Java programs that can be particularly beneficial for dynamic workloads.

Emerging Technologies: CRIU and Project CRaC for Checkpoint/Restore Functionality

OpenJDK continues to innovate via emerging projects such as CRIU and CraC. Together, they contribute to reducing the time to steady-state, expanding the realm of high-performance Java applications.

CRIU (Checkpoint/Restore in Userspace) is a pioneering Linux tool; it was initially crafted by Virtuozzo[9] and subsequently made available as an open-source program. CRIU's designed to support the live migration feature of OpenVZ, a server virtualization solution.[10] The brilliance of CRIU lies in its ability to momentarily freeze an active application, creating a checkpoint stored as files on a hard drive. This checkpoint can later be utilized to resurrect and execute the application from its frozen state, eliminating the need for any alterations or specific configurations.

The following command sequence creates a checkpoint of the process with the specified PID and stores it in the specified directory. The process can then be restored from this checkpoint.

```
# Install CRIU
$ sudo apt-get install criu

# Checkpoint a running process
$ sudo criu dump -t [PID] -D /checkpoint/directory -v4 -o dump.log

# Restore a checkpointed process
$ sudo criu restore -D /checkpoint/directory -v4 -o restore.log
```

Recognizing the potential of CRIU, Red Hat introduced it as a stand-alone project in OpenJDK. The aim was to provide a way to freeze the state of a running Java application, store it, and then restore it later or on a different system. This capability could be used to migrate running applications between systems, save and restore complex applications' state for debugging purposes, and create snapshots of applications for later analysis.

[8] www.graalvm.org/latest/reference-manual/java-on-truffle/

[9] https://criu.org/Main_Page

[10] https://wiki.openvz.org/Main_Page

Building on the capabilities of CRIU, Project CRaC (Coordinated Restore at Checkpoint)—a new project in the Java ecosystem—aims to integrate CRIU's checkpoint/restore functionality into the JVM. This could potentially allow Java applications to be checkpointed and restored, providing another avenue for improving time to steady-state. CRaC is currently in its early stages but represents a promising direction for the future of Java start-up performance optimization.

Let's simulate the process of checkpointing and restoring the state of the `OnlineLearningPlatform` web service:

```
import org.crac.Context;
import org.crac.Core;
import org.crac.Resource;

public class OnlineLearningPlatform implements Resource {
    public static void main(String[] args) throws Exception {
        // Register the platform in a global context for CRaC
        Core.getGlobalContext().register(new OnlineLearningPlatform());

        // Initialization
        System.out.println("Initializing OnlineLearningPlatform...");

        // Load Student and Course classes
        Student.loadAllStudents();
        Course.initializeCourses();

        // Ramp-up phase: Load Instructor class after its up-to-date list
        Instructor.loadAllInstructors();
        Instructor.teach(new Course()); // This method is a candidate for JIT optimization
    }

    @Override
    public void beforeCheckpoint(Context<? extends Resource> context) throws Exception {
        System.out.println("Preparing to checkpoint OnlineLearningPlatform...");
    }

    @Override
    public void afterRestore(Context<? extends Resource> context) throws Exception {
        System.out.println("Restoring OnlineLearningPlatform state...");
    }
}
```

In this enhanced version, the `OnlineLearningPlatform` class implements the `Resource` interface from CRaC. This allows us to define the behavior before a checkpoint (`beforeCheckpoint` method) and after a restore (`afterRestore` method). The platform is registered with CRaC's global context, enabling it to be checkpointed and restored.

The integration of CRIU into the JVM is a complex task because it requires changes to the JVM's internal structures and algorithms to support the freezing and restoring of application state. It also requires coordination with the operating system to ensure that the state of the JVM and the application it is running is correctly captured and restored. Despite the challenges posed, the integration of CRIU into the JVM offers remarkable potential benefits.

Figure 8.8 shows a high-level representation of the relationship between a Java application, Project CRaC, and CRIU. The figure includes the following elements:

1. **Java application:** This is the starting point, where the application runs and operates normally.
2. **Project CRaC:**
   - It provides an API for Java applications to initiate checkpoint and restore operations.
   - When a checkpoint is requested, Project CRaC communicates with CRIU to perform the checkpointing process.
   - Similarly, when a restore is requested, Project CRaC communicates with CRIU to restore the application from a previously saved state.
3. **CRIU:**
   - Upon receiving a checkpoint request, CRIU freezes the Java application process and saves its state as image files on the hard drive.
   - When a restore request is made, CRIU uses the saved image files to restore the Java application process to its checkpointed state.
4. **Checkpoint state and restored state:**
   - The "Checkpoint State" represents the state of the Java application at the time of checkpointing.
   - The "Restored State" represents the state of the Java application after it has been restored. It should be identical to the checkpoint state.
5. **Image files:** These are the files created by CRIU during the checkpointing process. They contain the saved state of the Java application.
6. **Restored process:** This represents the Java application process after it has been restored by CRIU.

Although these projects are still in the early stages of development, they represent promising directions for the future of Java start-up performance optimization. The integration of CRIU into the JVM could have a significant impact on the performance of Java applications. By allowing applications to be checkpointed and restored, it could potentially reduce start-up times, improve performance, and enable new use cases for Java applications.

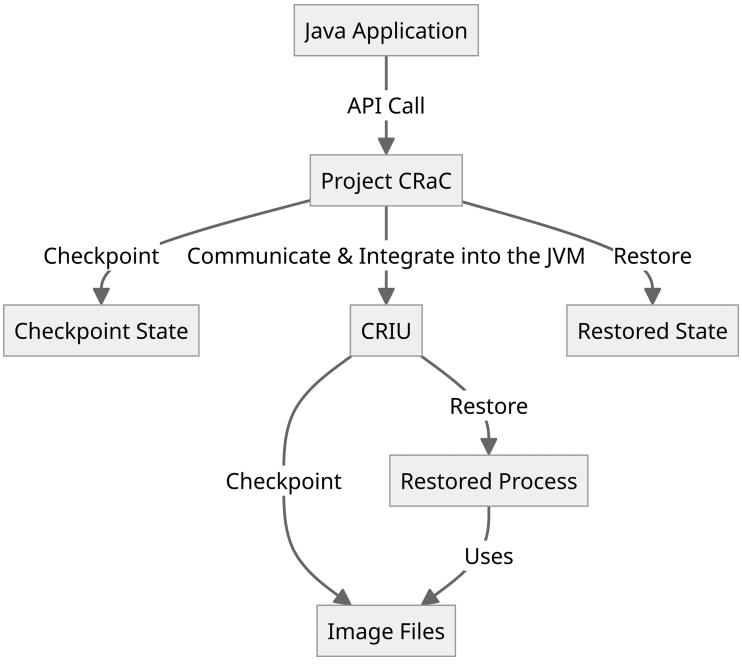

Figure 8.8 Applying Project CRaC and CRIU Strategies to a Java Application

Start-up and Ramp-up Optimization in Serverless and Other Environments

As cloud technologies evolve, serverless and containerized environments have become pivotal in application deployment. Java, with its ongoing advancements, is well suited to addressing the unique challenges of these modern computing paradigms, as shown in Figure 8.9.

- **Serverless cold starts:** GraalVM addresses JVM's cold start delays by pre-compiling applications, providing a quick function invocation response, while CDS complements by reusing class metadata to accelerate start-up times.
- **CDS enhancements:** These techniques are pivotal in both serverless and containerized setups, reusing class metadata for faster start-up and lower memory usage.
- **Prospective optimizations:** Anticipated projects like Leyden and CRaC aim to further refine start-up efficiency and application checkpointing, creating a future in which Java's "cold start" issues become a relic of the past.
- **Containerized dynamics:** Rapid scaling and efficient resource utilization are at the forefront, with Java poised to harness container-specific optimizations for agile performance.

Figure 8.9 encapsulates Java's ongoing journey toward a seamless cloud-native experience, underpinned by robust start-up and ramp-up strategies.

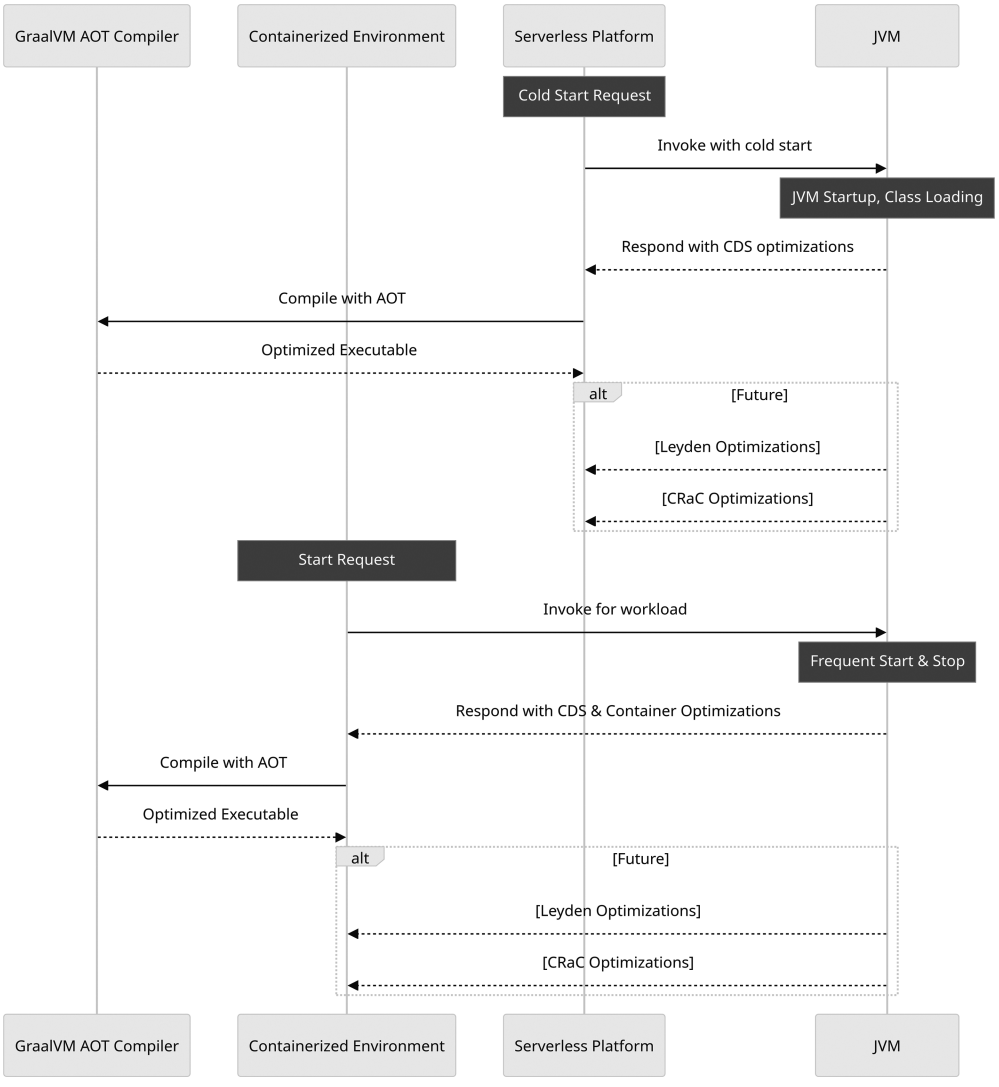

Figure 8.9 JVM Landscape with Containerized and Serverless Platforms

Serverless Computing and JVM Optimization

Serverless computing, with its on-demand execution of application functions, offers a paradigm shift in application deployment and scaling. This approach, which liberates developers from managing servers, enhances operational efficiency and allows for automatic scaling to match workloads. However, serverless environments introduce the "cold start" problem, which is especially pronounced when a dormant function is invoked. This necessitates resource allocation, runtime initiation, and application launch, which can be time-intensive for Java applications due to JVM start-up, class loading, and JIT compilation processes.

The "cold start" problem comes into play in scenarios where a Java application, after being inactive, faces a sudden influx of requests in a serverless environment. This necessitates rapid start-up of multiple application instances, where delays can impact user experience.

To address this issue, JVM optimizations like CDS and JIT compiler enhancements can significantly mitigate cold-start delays, making Java more suitable for serverless computing. Platforms like AWS Lambda and Azure Functions support Java and provide the flexibility to fine-tune JVM configurations. By leveraging CDS and calibrating other JVM options, developers can optimize serverless Java functions, ensuring quicker start-up and ramp-up performance of those functions.

Anticipated Benefits of Emerging Technologies

Emerging technologies like Project Leyden are shaping Java's future performance, particularly for serverless computing. Leyden promises to enhance Java's start-up phase and steady-state efficiency through innovations like "training runs" and "condensers." These developments could substantially cut cold-start times. Even so, it's critical to remember that Leyden's features, as of JDK 21, are still evolving and not ready for production. When Leyden matures, it could transform serverless Java applications, enabling them to quickly react to traffic spikes without the usual start-up delays, thanks to precomputed state storage during "training runs."

Alongside Leyden, Project CRaC offers a hands-on approach. Developers can pinpoint and prepare specific application segments for checkpointing, leading to stored states that can be rapidly reactivated. This approach is particularly valuable in serverless contexts, where reducing cold starts is critical.

These initiatives—Leyden and CRaC—signify a forward leap for Java in serverless environments, driving it toward a future where applications can scale instantaneously, free from the constraints of cold starts.

Containerized Environments: Ensuring Swift Start-ups and Efficient Scaling

In the realm of modern application deployment, containerization has emerged as a game-changer. It encapsulates applications in a lightweight, consistent environment, making them a perfect fit for microservices architectures and cloud-native deployments. Similar to serverless computing, time-to-steady-state performance is crucial. In containerized environments, applications are packaged along with their dependencies into a container, which is then run on a container orchestration platform (for example, Kubernetes). Because containers can be started and stopped frequently based on the workload, having fast start-up and ramp-up times is essential to ensure that the application can quickly scale up to handle the increased load.

Current JVM optimization techniques, such as CDS and JIT compiler enhancements, already contribute significantly to enhancing performance in these contexts. These established strategies include

- **Opt for a minimal base Docker image** to minimize the image size, which consequently improves the start-up and ramp-up durations.
- **Tailor JVM configurations** to better suit the container environment, such as setting the JVM heap size to be empathetic to the memory limits of the container.

- **Strategically layer Docker images** to harness Docker's caching mechanism and speed up the build process.
- **Implement health checks** to ensure the application is running correctly in the container.

By harnessing these techniques, you can ensure that your Java applications in containerized environments start up and ramp up as quickly as possible, thereby providing a better user experience and making more efficient use of system resources.

Figure 8.10 illustrates both the current strategies and the future integration of Leyden's (and CRaC's) methodologies. It's a roadmap that developers can follow today while looking ahead to the benefits that upcoming projects promise to deliver.

GraalVM's Present-Day Contributions

GraalVM is currently addressing some of the challenges that Java faces in modern computing environments. Its notable feature, the native image capability, provides AOT compilation, significantly reducing start-up times—a crucial advantage for serverless computing and microservices architectures. GraalVM's AOT compiler complements the traditional HotSpot VM by targeting specific performance challenges, which is beneficial for scenarios where low latency and efficient resource usage are paramount.

Although GraalVM's AOT compiler is not a one-size-fits-all solution, it offers a specialized approach to Java application performance, catering to the needs of specific use cases in the cloud-native landscape. This versatility positions Java as a strong contender in diverse deployment environments, ensuring that applications remain agile and performant.

Key Takeaways

Java's evolution continues to align with cloud-native requirements. Projects like Leyden and CRaC, alongside the capabilities of GraalVM, showcase Java's adaptability and commitment to performance enhancement and scalability:

- **Adaptable Java evolution:** With initiatives like Project Leyden and the existing strengths of GraalVM, Java demonstrates its adaptability to modern cloud-native deployment paradigms.
- **Optimization techniques:** Understanding and applying JVM optimization techniques is crucial. This includes leveraging CDS and JIT compiler enhancements for better performance.
- **GraalVM AOT compilation in specific scenarios:** For use cases where rapid start-up and a small memory footprint are critical, such as in serverless environments or microservices, GraalVM's AOT compiler offers an effective solution. It complements the traditional HotSpot VM by addressing specific performance challenges of these environments.
- **Anticipating the impact of future projects:** Projects Leyden and CRaC signify Java's forward leap in serverless environments, promising instantaneous scalability and reduced cold-start constraints.

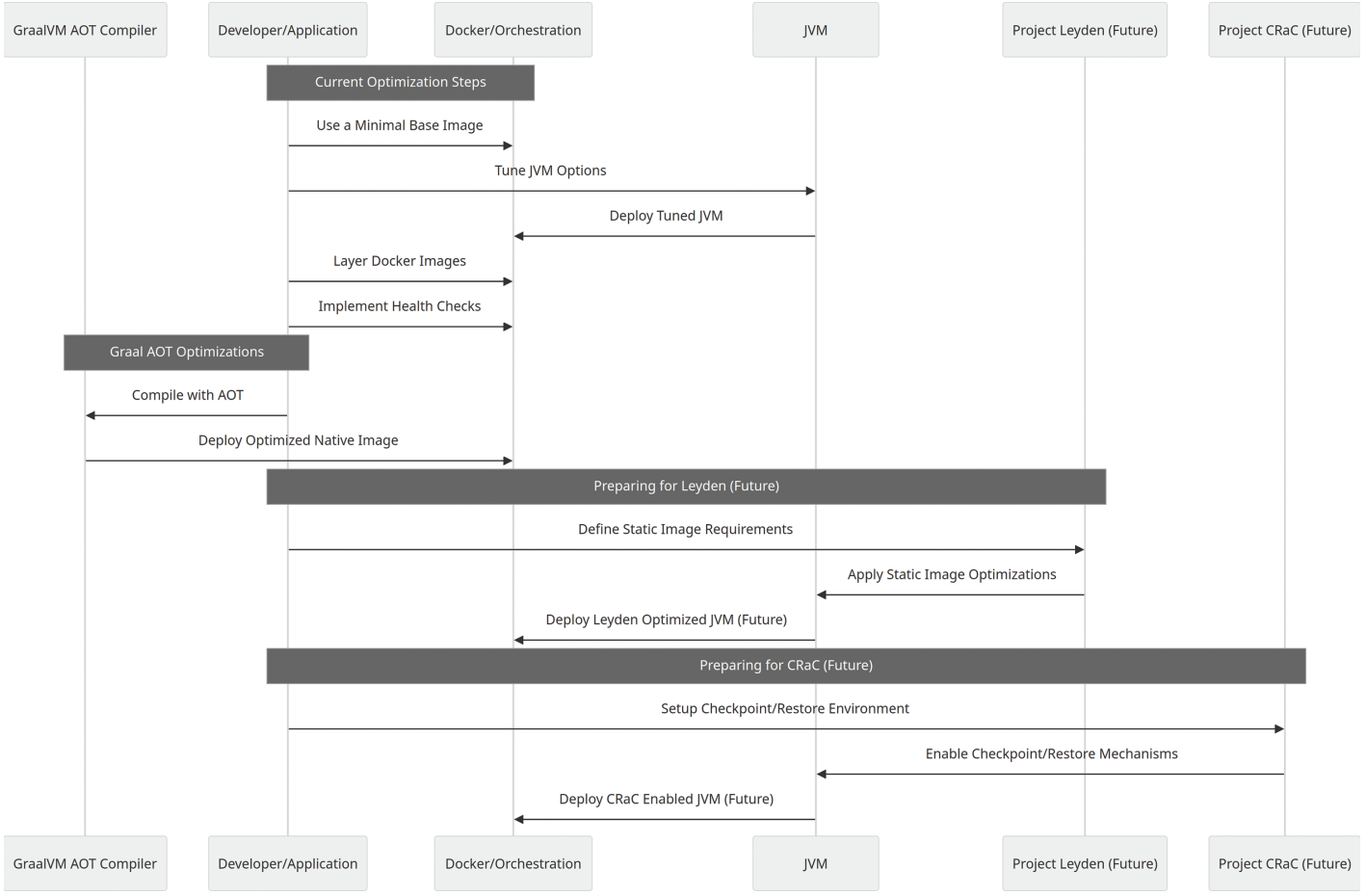

Figure 8.10 Containers' Strategies When Running on the JVM

- **Serverless and containerized performance:** For both serverless and containerized deployments, Java is evolving to ensure swift start-ups and efficient scaling, aligning with the demands of modern cloud applications.
- **Balanced approach:** As the Java ecosystem evolves, a balanced approach in technology adoption is key. Developers should weigh their application needs and the specific benefits offered by each technology, be it current JVM optimizations, GraalVM's capabilities, or upcoming innovations like Leyden and CRaC.

Boosting Warm-up Performance with OpenJDK HotSpot VM

The OpenJDK HotSpot VM is a sophisticated piece of software that employs a multitude of techniques to optimize the performance of Java applications. These techniques are designed to work in harmony, each addressing different aspects of the application life cycle, from start-up to steady-state. This section delves into the intricate workings of the HotSpot VM, shedding light on the mechanisms it uses to boost performance and the roles they play in the broader context of Java application performance.

In the dynamic world of scalable architectures, including microservices and serverless models, Java applications are often subjected to the rigors of frequent restarts and inherently transient lifespans. Such environments amplify the significance of an application's initialization phase, its warm-up period, and its journey to a steady operational state. The efficiency with which an application navigates these phases can make or break its responsiveness, especially when faced with fluctuating traffic demands or the need for rapid scalability. That's where the various optimizations employed by the HotSpot VM—including JIT compilation, adaptive optimization and tiered compilation, and the strategic choice between client and server compilations—come into play.

Compiler Enhancements

In Chapter 1, "The Performance Evolution of Java: The Language and the Virtual Machine", we looked into the intricacies of HotSpot VM's JIT compiler and its array of compilation techniques (Figure 8.11). As we circle back to this topic, let's summarize the process while emphasizing the optimizations, tailored for start-up, warm-up, and steady-state phases, within the OpenJDK HotSpot VM.

Start-up Optimizations

Start-up in the JVM is primarily handled by the (s)lower tiers. The JVM also takes charge of the *invokedynamic* bootstrap methods (BSM; discussed in detail in Chapter 7, "Runtime Performance Optimizations: A Focus on Strings, Locks, and Beyond") and class initializers.

- **Bytecode generation:** The Java program is first converted into bytecode. This is the initial phase of the Java compilation process, in which the high-level code written by developers is transformed into a platform-agnostic format that can be executed by the JVM (irrespective of the underlying hardware).

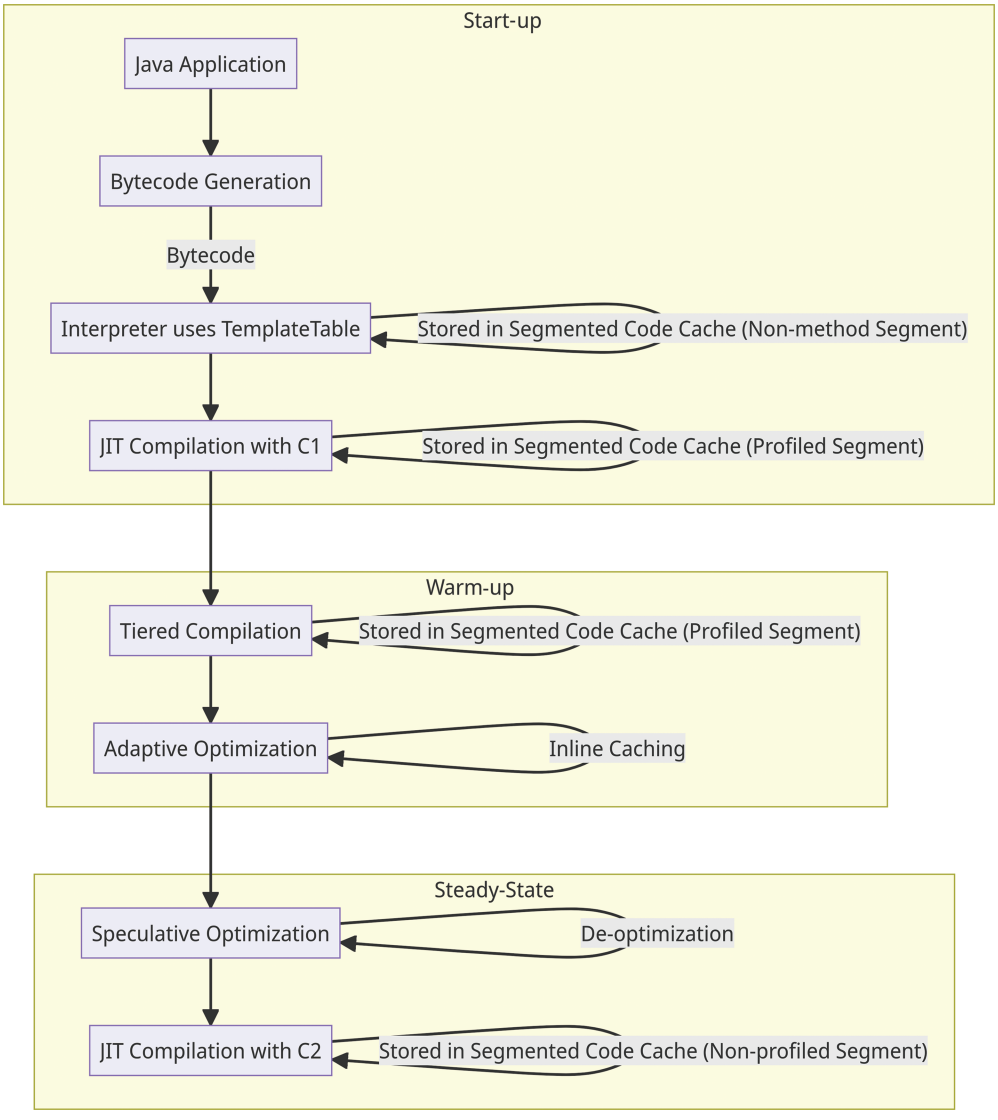

Figure 8.11 OpenJDK Compilation Strategies

- **Interpretation:** The bytecode is then interpreted based on a description table, the *TemplateTable*. This table provides a mapping between bytecode instructions and their corresponding machine code sequences, allowing the JVM to execute the bytecode.
- **JIT compilation with client compiler (C1):** The C1 JIT compiler is designed to provide a balance between quick compilation and a basic level of optimization. It translates the bytecode into native machine code, ensuring that the application quickly moves from the start-up phase and begins its journey toward peak performance. During this phase,

the C1 compiler helps with gathering profiling information about the application behavior. This profiling data is crucial because it informs the subsequent more aggressive optimizations carried out by the C2 compiler. The C1's JIT compilation process is particularly beneficial for methods that are invoked multiple times, because it avoids the overhead of repeatedly interpreting the same bytecode.

- **Segmented *CodeCache*:** The optimized code generated from different tiers of JIT compilation, along with the profiling data, is stored in the *Segmented CodeCache*. This cache is divided into segments, with the *Profiled* segment containing lightly optimized, profiled methods with a short lifetime. It also has a *Non-method* segment that contains non-method code like the bytecode interpreter itself. As we get closer to steady state, the non-profiled, fully optimized methods are also stored in the *CodeCache* in the *Non-profiled* segment. The cache allows for faster execution of frequently used code sequences.

Warm-up Optimizations

As Java applications move beyond the start-up phase, the warm-up phase becomes crucial in setting the stage for peak performance. The JIT compilation, particularly with the client compiler (C1), plays a pivotal role during this transition.

- **Tiered compilation:** A cornerstone of the JVM's performance strategy is tiered compilation, in which code transitions from T0 (interpreted code) to T4 (the highest level of optimization). Tiered compilation allows the JVM to balance the trade-off between faster start-up and peak performance. In the early stages of execution, the JVM uses a faster but less optimized level of compilation to ensure quick start-up. As the application continues to run, the JVM applies more aggressive optimizations to achieve higher peak performance.
- **Adaptive optimization:** The JIT-compiled code undergoes adaptive optimization, which can result in further optimized code or on-stack replacement (OSR). Adaptive optimization involves profiling the running application and optimizing its performance based on its behavior. For example, methods that are called frequently may be further optimized, whereas those superseded may be de-optimized, with their resources reallocated.
- **Inlined caching:** A nuanced technique employed during warm-up is *inlined caching*. With this approach, small yet frequently invoked methods are inlined within the calling method. This strategy minimizes the overhead of method calls and is informed by the profiling data amassed during the warm-up.

Steady-State Optimizations

As Java applications mature in their life cycle, transitioning from warm-up to a steady operational state, the JVM employs a series of advanced optimization techniques. These are designed to maximize the performance of long-running applications, ensuring that they operate at peak efficiency.

- **Speculative optimization:** One of the standout techniques is speculative optimization. Leveraging the rich profiling data accumulated during the warm-up phase, the JVM makes informed predictions about probable execution paths. It then tailors the code optimization based on these anticipations. If any of these educated speculations

proves inaccurate, the JVM is equipped to gracefully revert to a prior, less optimized code version, safeguarding the application's integrity. This strategy shines in enduring applications, where the minor overhead of occasional optimization rollbacks is vastly overshadowed by the performance leaps from accurate speculations.

- **De-optimization:** Going hand in hand with speculative optimization is the concept of de-optimization. This mechanism empowers the JVM to dial back certain JIT compiler optimizations when foundational assumptions are invalidated. A classic scenario is when a newly loaded class overrides a method that had been previously optimized. The ability to revert ensures that the application remains accurate and responsive to dynamic changes.
- **JIT compilation with server compiler (C2):** For long-running applications, the meticulous server compiler (C2) backed with profiling information takes over, applying aggressive and speculative optimizations to performance-critical methods. This process significantly improves the performance of Java applications, particularly for long-running applications where the same methods are invoked multiple times.

Segmented Code Cache and Project Leyden Enhancements

Building upon our detailed discussion of the Segmented *CodeCache* in Chapter 1, "The Performance Evolution of Java: The Language and the Virtual Machine," let's now consider its impact on reducing time-to-steady-state performance of Java applications.

The Segmented *CodeCache* is designed to prioritize the storage of frequently executed code, ensuring that it is readily available for execution. Diving deeper into the mechanics of the *CodeCache*, it's essential to understand the role of the *Code ByteBuffer*. This Buffer acts as an intermediary storage, holding the compiled code before it's moved to the Segmented *CodeCache*. This two-step process ensures efficient memory management and optimized code retrieval. Furthermore, the segmentation of the cache allows for more efficient memory management, as each segment can be sized independently based on its usage. In the context of start-up and warm-up performance, the Segmented *CodeCache* contributes significantly by ensuring that the most critical parts of the application code are compiled and ready for execution as quickly as possible, thereby enhancing the overall performance of Java applications.

With the advent of Project Leyden, this mechanism will see further enhancements. One of the standout features of Project Leyden will be its ability to load the Segmented *CodeCache* directly from an archive. This will bypass the traditional process of tiered code compilation during start-up and warm-up, leading to a significant reduction in start-up times. Instead of compiling the code every time the application starts, Project Leyden will allow the precompiled code to be archived and then directly loaded into the Segmented *CodeCache* upon subsequent start-ups.

This approach will not only accelerate the start-up process but also ensure that the application benefits from optimized code from the get-go. By leveraging this feature, developers will achieve faster application responsiveness, especially in environments where accelerated time-to-steady-state is crucial.

The Evolution from PermGen to Metaspace: A Leap Forward Toward Peak Performance

In the history of Java, prior to the advent of Java 8, the JVM memory included a space known as the Permanent Generation (PermGen). This segment of memory was designated for storing class metadata and statics. However, the PermGen model was not without its flaws. It had a fixed size, which could lead to `java.lang.OutOfMemoryError: PermGen space` errors if the space allocated was inadequate for the demands of the application.

Start-up Implications

The fixed nature of PermGen meant that the JVM had to allocate and deallocate memory during start-up, leading to slower start-up times. In an effort to rectify these shortcomings, Java 8 introduced a significant change: the replacement of PermGen with Metaspace.[11] Metaspace, unlike its predecessor, is not a contiguous heap space, but rather is located in the native memory and used for class metadata storage. It grows automatically by default, and its maximum limit is the amount of available native memory, which is much larger than the typical maximum PermGen size. This crucial modification has contributed to more efficient memory management during start-up, potentially accelerating start-up times.

Warm-up Implications

The dynamic nature of Metaspace, which allows it to grow as needed, ensures that the JVM can quickly adapt to the demands of the application during the warm-up phase. This flexibility reduces the chances of memory-related bottlenecks during this critical phase.

When the Metaspace is filled up, a full garbage collection (GC) is triggered to clear unused class loaders and classes. If GC cannot reclaim enough space, the Metaspace is expanded. This dynamic nature helps avoid the out-of-memory errors that were common with PermGen.

Steady-State Implications

The shift from PermGen to Metaspace ensures that the JVM reaches a steady-state more efficiently. By eliminating the constraints of a fixed memory space for class metadata and statics, the JVM can manage its memory resources and mitigate the risk of out-of-memory errors, resulting in Java applications that are more robust and reliable.

However, although Metaspace can grow as needed, it's not immune to memory leaks. If class loaders are not handled properly, the native memory can fill up, resulting in an `OutOfMemoryError: Metaspace`. Here's how:

1. **Class loader life cycle and potential leaks:**
    a. Each class loader has its own segment of the Metaspace where it loads class metadata. When a class loader is garbage collected, its corresponding Metaspace segment is also freed. However, if live references to the class loader (or to any classes it loaded) persist, the class loader won't be garbage collected, and the Metaspace memory it's using

[11] www.infoq.com/articles/Java-PERMGEN-Removed/

won't be freed. This is how improperly managed class loaders can lead to memory leaks.

   b. Class loader leaks can occur in several other ways. For example, suppose a class loader loads a class that starts a thread, and the thread doesn't stop when the class loader is no longer needed. In that case, the active thread will maintain a reference to the class loader, preventing it from being garbage collected. Similarly, suppose a class loaded by a class loader registers a static hook (for example, a shutdown hook or a JDBC driver), and doesn't deregister the hook when it's done. This can also prevent the class loader from being garbage collected.

   c. In the context of garbage collection, the class loader is a GC root. Any object that is reachable from a GC root is considered alive and is not eligible for garbage collection. So, as long as the class loader remains alive, all the classes it has loaded (and any objects those classes reference) are also considered alive and are not eligible for garbage collection.

2. **`OutOfMemoryError: Metaspace`:** If enough memory leaks accumulate (due to improperly managed class loaders), the Metaspace could fill up. Under that condition, the JVM will trigger a full garbage collection to clear out unused class loaders and classes. If this doesn't free up enough space, the JVM will attempt to expand the Metaspace. If the Metaspace can't be expanded because not enough native memory is available, an `OutOfMemoryError: Metaspace` will be thrown.

It's crucial to monitor Metaspace usage and adjust the maximum size limit as needed. Several JVM options can be used to control the size of the Metaspace:

- **`-XX:MetaspaceSize=[size]`** sets the initial size of the Metaspace. If it is not specified, the Metaspace will be dynamically resized based on the application's demand.
- **`-XX:MaxMetaspaceSize=[size]`** sets the maximum size of the Metaspace. If it is not specified, the Metaspace can grow without limit, up to the amount of available native memory.
- **`-XX:MinMetaspaceFreeRatio=[percentage]` and `-XX:MaxMetaspaceFreeRatio=[percentage]`** control the amount of free space allowed in the Metaspace after GC before it will resize. If the percentage of free space is less than or greater than these thresholds, the Metaspace will shrink or grow accordingly.

Tools such as VisualVM, JConsole, and Java Mission Control can be used to monitor Metaspace usage and provide valuable insights into memory usage, GC activity, and potential memory leaks. These tools can also help identify class loader leaks by showing the number of classes loaded and the total space occupied by these classes.

Java 16 brought a significant upgrade to Metaspace with the introduction of JEP 387: *Elastic Metaspace*.[12] The primary objectives of JEP 387 were threefold:

- Efficiently relinquish unused class-metadata memory back to the OS
- Minimize the overall memory footprint of Metaspace
- Streamline the Metaspace codebase for enhanced maintainability

[12] https://openjdk.org/jeps/387

Here's a breakdown of the key changes that JEP 387 brought to the table:

- **Buddy-based allocation scheme:** This mechanism organizes memory blocks based on size, facilitating rapid allocation and deallocation in Metaspace. It's analogous to a buddy system, ensuring efficient memory management.
- **Lazy memory commitment:** A judicious approach in which the JVM commits memory only when it's imperative. This strategy curtails the JVM's memory overhead, especially when the allocated Metaspace significantly surpasses its actual utilization.
- **Granular memory management:** This enhancement empowers the JVM to manage Metaspace in finer segments, mitigating internal fragmentation and optimizing memory consumption.
- **Revamped reclamation policy:** A proactive strategy enabling the JVM to swiftly return unused Metaspace memory to the OS. This is invaluable for applications with fluctuating Metaspace usage patterns, ensuring a consistent and minimal memory footprint.

These innovations adeptly address Metaspace management challenges, including memory fragmentation. In summary, the dynamic nature of Metaspace, complemented by the innovations introduced in JEP 387, underscores Java's commitment to optimizing memory utilization, reducing fragmentation, and promptly releasing unused memory back to the OS.

Conclusion

In this chapter, we have comprehensively explored JVM start-up and warm-up performance—a critical aspect of Java application performance. Various optimization techniques introduced into the Java ecosystem, such as CDS, GraalVM, and JIT compiler enhancements, have significantly improved start-up and ramp-up times. For microservices architectures, serverless computing, and containerized environments, rapid initializations and small memory footprints are crucial.

As we step into the future, the continuous evolution of Java's performance capabilities remains at the forefront of technological innovation. The introduction of Project Leyden, with its training runs, and the emergence of CRIU and Project CRaC further amplify the potential to drive Java performance optimization.

As developers and performance engineers, it's important that we stay informed about these kinds of advancements and understand how to apply them in different environments. By doing so, we can ensure that our Java applications are as efficient and performant as possible, providing the best possible user experiences.

Chapter 9

Harnessing Exotic Hardware: The Future of JVM Performance Engineering

Introduction to Exotic Hardware and the JVM

In the ever-advancing realm of computational technology, specialized hardware components like graphics processing units (GPUs), field programmable gate arrays (FPGAs), and a plethora of other accelerators are making notable strides. Often termed "exotic hardware," these components are just the tip of the iceberg. Cutting-edge hardware accelerators, including tensor processing units (TPUs), application-specific integrated circuits (ASICs), and innovative AI chips such as Axelera,[1] are reshaping the performance benchmarks across diverse applications. While these powerhouses are primarily tailored to supercharge machine learning and intricate data processing tasks, their prowess isn't limited to just that arena. JVM-based applications, with the aid of specialized interfaces and libraries, can tap into these resources. For example, GPUs, celebrated for their unparalleled parallel computation capabilities, can be a boon for Java applications, turbocharging data-heavy tasks.

Historically, the JVM, as a platform-independent execution environment, has been associated with general-purpose computing in which the bytecode is compiled to machine code for the CPU. This design has allowed the JVM to provide a high degree of portability across different hardware platforms, as the same bytecode can run on any device with a JVM implementation.

However, the rise of exotic hardware has brought new opportunities and challenges for the JVM. Although general-purpose CPUs are versatile and capable, they are often complemented by specialized hardware components that provide capabilities tailored for specific tasks. For instance, while CPUs have specialized instruction sets like AVX512 (Advanced Vector Extensions)[2] for Intel processors and SVE (Scalable Vector Extension)[3] for Arm architectures, which enhance performance for specific tasks, other hardware accelerators offer the ability to

[1] www.axelera.ai/

[2] https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html

[3] https://developer.arm.com/Architectures/Scalable%20Vector%20Extensions

perform large-scale matrix operations or specialized computations essential for tasks like deep learning. To take full advantage of these capabilities, the JVM needs to be able to compile bytecode into machine code that can run efficiently on such hardware.

This has led to the development of new language features, APIs, and toolchains designed to bridge the gap between the JVM and specialized hardware. For example, the Vector API, which is part of Project Panama, allows developers to express computations that can be efficiently vectorized on hardware that supports vector operations. By effectively harnessing these hardware capabilities, we can achieve remarkable performance improvements.

Nevertheless, the task of effectively utilizing these specialized hardware components in JVM-based applications is far from straightforward. It necessitates adaptations in both language design and toolchains to tap into these hardware capabilities effectively, including the development of APIs capable of interfacing with such hardware and compilers that can generate code optimized for these components. However, challenges such as managing memory access patterns, understanding hardware-specific behaviors, and dealing with the heterogeneity of hardware platforms can make this a complex task.

Several key concepts and projects have been instrumental in this adaptation process:

- **OpenCL:** An open standard that enables portable parallel programming across heterogeneous systems, including CPUs, GPUs, and other processors. Although OpenCL code can be executed on a variety of hardware platforms, its performance is *not* universally portable. In other words, the efficiency of the same OpenCL code can vary significantly depending on the hardware on which it's executed. For instance, while a commodity GPU might offer substantial performance improvements over sequential code, the same OpenCL code might run more slowly on certain FPGAs.[4]
- **Aparapi:** A Java API designed for expressing data parallel workloads. Aparapi translates Java bytecode into OpenCL, enabling its execution on various hardware accelerators, including GPUs. Its runtime component manages data transfers and the execution of the generated OpenCL code, providing performance benefits for data-parallel tasks. Although Aparapi abstracts much of the complexity, achieving optimal performance on specific hardware might require tuning.
- **TornadoVM:** An extension of the OpenJDK GraalVM, TornadoVM offers the unique benefit of dynamically recompiling and optimizing Java bytecode for different hardware targets at runtime. This allows Java programs to automatically adapt to available hardware resources, such as GPUs and FPGAs, without any code changes. The dynamic nature of TornadoVM ensures that applications can achieve optimal performance portability based on the specific hardware and the nature of the application.
- **Project Panama:** An ongoing project in the OpenJDK community aimed at improving the connection between the JVM's interoperability with native code. It focuses on two main areas:
  - **Vector API:** Designed for vector computations, this API ensures runtime compilation to the most efficient vector hardware instructions on supported CPU architectures.

[4] Michail Papadimitrioua, Juan Fumero, Athanasios Stratikopoulosa, Foivos S. Zakkakb, and Christos Kotselidis. "Transparent Compiler and Runtime Specializations for Accelerating Managed Languages on FPGAs." *Art, Science, and Engineering of Programming* 5, no. 2 (2020). https://arxiv.org/ftp/arxiv/papers/2010/2010.16304.pdf.

- **Foreign Function and Memory (FFM) API:** This tool allows a program to call routines or utilize services written in a different language. Within the scope of Project Panama, the FFM enables Java code to interact seamlessly with native libraries, thereby enhancing the interoperability of Java with other programming languages and systems.

This chapter explores the challenges associated with JVM performance engineering and discusses some of the solutions that have been proposed. Although our primary focus will be on the OpenCL-based toolchain, its important to acknowledge that CUDA[5] is one of the most widely adopted parallel programming models for GPUs. The significance of language design and toolchains cannot be overstated when it comes to effectively utilizing the capabilities of exotic hardware. To provide a practical understanding, a series of illustrative case studies will be presented, showcasing examples of how these challenges have been addressed and the opportunities that exotic hardware presents for JVM performance engineering.

Exotic Hardware in the Cloud

The availability of cloud computing has made it convenient for end users to gain access to specialized or heterogeneous hardware. In the past, leveraging the power of exotic hardware often required a significant upfront investment in physical hardware. This requirement was a formidable barrier for many developers and organizations, particularly those with limited resources.

In recent years, the cloud computing revolution has dramatically changed this landscape. Now, developers can access and leverage the power of exotic hardware without making a substantial initial investment. This has been made possible by major cloud service providers, including Amazon Web Services (AWS),[6] Google Cloud,[7] Microsoft Azure,[8] and Oracle Cloud Infrastructure.[9] These providers have extended their offerings with virtual machines that come equipped with GPUs and other accelerators, enabling developers to harness the power of exotic hardware in a flexible and cost-effective manner. NVIDIA has been at the forefront of GPU virtualization, offering a flexible approach to video encoding/decoding and broad hypervisor support, but AMD and Intel also have their own distinct methods and capabilities.[10]

To address the virtualization complexities, many cloud "provisioners" ensure that only one host requiring the GPU is deployed on physical hardware, thus gaining full and exclusive access to it. Although this approach allows other (non-GPU) hosts to utilize the CPUs, it does speak to some of the underlying complexities of utilizing exotic hardware in the cloud. Indeed, harnessing specialized hardware components in the cloud presents its own set of challenges, particularly from a software and runtime perspective. Some of these challenges are summarized in the following subsections.

---

[5] https://developer.nvidia.com/cuda-zone

[6] https://aws.amazon.com/ec2/instance-types/f1/

[7] https://cloud.google.com/gpu

[8] www.nvidia.com/en-us/data-center/gpu-cloud-computing/microsoft-azure/

[9] www.oracle.com/cloud/compute/gpu/

[10] Gabe Knuth. "NVIDIA, AMD, and Intel: How They Do Their GPU Virtualization." *TechTarget* (September 26, 2016). www.techtarget.com/searchvirtualdesktop/opinion/NVIDIA-AMD-and-Intel-How-they-do-their-GPU-virtualization.

Hardware Heterogeneity

The landscape of cloud computing is marked by a wide array of hardware offerings, each with unique features and capabilities. For instance, newer Arm devices are equipped with cryptographic accelerators that significantly enhance the speed of cryptographic operations. In contrast, Intel's AVX-512 instructions expand the width of SIMD vector registers to 512 bits, facilitating parallel processing of more data.

Arm's technology spectrum also includes NEON and SVE. Whereas NEON provides SIMD instructions for Arm v7 and Arm v8, SVE allows CPU designers to select the SIMD vector length that best aligns with their requirements, ranging from 128 bits to 2048 bits in 128-bit increments. These variations in data widths and access patterns can significantly influence computational performance.

The diversity also extends to GPUs, FPGAs, and other accelerators, albeit with their availability varying significantly across cloud providers. This hardware heterogeneity necessitates software and runtimes to be adaptable and flexible, capable of optimizing performance based on the specific hardware in use. Adaptability in this context often involves taking different optimization paths depending on the hardware, thereby ensuring efficient utilization of the available resources.

API Compatibility and Hypervisor Constraints

Specialized hardware often requires specific APIs to be used effectively. These APIs provide a way for software to interact with the hardware, allowing for tasks such as memory management and computation to be performed on the hardware. One such widely used API for general-purpose computation on GPUs (GPGPU) is OpenCL. However, JVM, which is designed to be hardware-agnostic, may not natively support these specialized APIs. This necessitates the use of additional libraries or tools, such as the Java bindings for OpenCL,[11] to bridge this gap.

Moreover, the hypervisor used by the cloud provider plays a crucial role in managing and isolating the resources of the virtual machines, providing security and stability. However, it can impose additional constraints on API compatibility. The hypervisor controls the interaction between the guest operating system and the hardware, and while it is designed to support a wide range of operations, it may not support all the features of the specialized APIs. For example, memory management in NVIDIA's CUDA or OpenCL requires direct access to the GPU memory, which may not be fully supported by the hypervisor.

An important aspect to consider is potential security concerns, particularly with regard to GPU memory management. Traditional hypervisors and their handling of the input–output memory management unit (IOMMU) can prevent regular memory from leaking between different hosts. However, the GPU often falls outside this "scope." The GPU driver, which is usually unaware that it is running in a virtualized environment, relies on maintaining memory resident between kernel dispatches. This raises concerns about data isolation—namely, whether data written by one host is truly isolated from another host. With GPGPU, in which the GPU is not just processing images but is also used for general computation tasks, any potential data leakage could be significantly more consequential. Modern cloud infrastructures have made strides in addressing these issues, yet the potential risk still exists, underlining the importance of allowing only one host to have access to the GPU.

[11] www.jocl.org/

Although these constraints might seem daunting, it's worth reiterating the vital role that hypervisors play in maintaining secure and isolated environments. Developers, however, must acknowledge these factors when they're aiming to fully leverage the capabilities of specialized hardware in the cloud.

Performance Trade-offs

The use of virtualized hardware in the cloud is a double-edged sword. On the one hand, it provides flexibility, scalability, and isolation, making it easier to manage and deploy applications. On the other hand, it can introduce performance trade-offs. The overhead of virtualization, which is necessary to provide these benefits, can sometimes offset the performance benefits of the specialized hardware.

This overhead is often attributable to the hypervisor, which needs to translate calls from the guest operating system to the host hardware. Hypervisors are designed to minimize this overhead and are continually improving in efficiency. However, in some scenarios, the burden can still impact the performance of applications running on exotic hardware. For instance, a GPU-accelerated machine learning application might not achieve the expected speed-up due to the overhead of GPU virtualization, particularly if the application hasn't been properly optimized for GPU acceleration or if the virtualization overhead isn't properly managed.

Resource Contention

The shared nature of cloud environments can lead to inconsistent performance due to contention for resources. This is often referred to as the "noisy neighbor" problem, as the activity of other users on the same physical hardware can impact your application's performance. For example, suppose multiple users are running GPU-intensive tasks on the same physical server: They might experience reduced performance due to contention for GPU resources if a noisy neighbor can monopolize GPU resources, causing other users' tasks to run more slowly. This is a common issue in cloud computing infrastructure, where resources are shared among multiple users. To mitigate this problem, cloud providers often implement resource allocation strategies to ensure fair usage of resources among all users. However, these strategies are not always perfect and can lead to performance inconsistencies due to resource contention.

Cloud-Specific Limitations

Cloud environments can impose additional limitations that are not present in on-premises environments. For example, cloud providers typically limit the amount of resources, such as memory or GPU compute units, that a single virtual machine can use. This constraint can limit the performance benefits of exotic hardware in the cloud. Furthermore, the ability to use certain types of exotic hardware may be restricted to specific cloud regions or instance types. For instance, Google Cloud's A2 VMs, which support the NVIDIA A100 GPU, are available only in selected regions.

Both industry and the research community have been actively working on solutions to these challenges. Efforts are being made to standardize APIs and develop hardware-agnostic programming models. As discussed earlier, OpenCL provides a unified, universal standard environment for parallel programming across heterogenous computing systems. It's designed to harness the computational power of diverse hardware components within a single node, making it ideal for efficiently running domain-specific workloads, such as data parallel applications and big data processing. However, to fully leverage OpenCL's capabilities within the JVM, Java bindings for OpenCL are essential. Although OpenCL addresses the computational aspect of this challenge within a node, high-performance computing platforms, especially those used in supercomputers, often require additional libraries like MPI for inter-node communication.

During my exploration of these topics, I had the privilege of discussing them with Dr. Juan Fumero, a leading figure behind the development of TornadoVM, a plug-in for OpenJDK that aims to address these challenges. Dr. Fumero shared a poignant quote from Vicent Natol of HPC Wire: "CUDA is an elegant solution to the problem of representing parallelism in algorithms—not all algorithms, but enough to matter." This sentiment applies not only to CUDA but also to other parallel programming models such as OpenCL, oneAPI, and more. Dr. Fumero's insights into parallel programming and its challenges are particularly relevant when considering solutions like TornadoVM. Developed under his guidance, TornadoVM seeks to harness the power of parallel programming in the JVM ecosystem by allowing Java programs to automatically run on heterogeneous hardware. Designed to take advantage of the GPUs and FPGAs available in cloud environments, TornadoVM accelerates Java applications. By providing a high-level programming model and handling the complexities of hardware heterogeneity, API compatibility, and potential security concerns, TornadoVM makes it easier for developers to leverage the power of exotic hardware in the cloud.

The Role of Language Design and Toolchains

To effectively utilize the capabilities of exotic hardware, both language design and toolchains need to adapt to their new demands. This involves several key considerations:

- **Language abstractions:** The programming language should offer intuitive high-level abstractions, enabling developers to craft code that can execute efficiently on different types of hardware. This involves designing language features that can express parallelism and take advantage of the unique features of exotic hardware. For example, the Vector API in Project Panama provides a way for developers to express computations that can be efficiently vectorized on hardware that supports vector operations.
- **Compiler optimizations:** The toolchain, and in particular the compiler, plays a crucial role in translating high-level language abstractions into efficient low-level code that can run on exotic hardware. This involves developing sophisticated optimization techniques that can take advantage of the unique features of different types of hardware. For example, the TornadoVM compiler can generate OpenCL, CUDA, and SPIR-V code. Furthermore, it produces code optimized for FPGAs, RISC-V with vector instructions, and Apple M1/M2 chips. This allows TornadoVM to be executed on a vast array of computing systems, from IoT devices like NVIDIA Jetsons to PCs, cloud environments, and even the latest consumer-grade processors.

- **Runtime systems:** The runtime system needs to be able to manage and schedule computations on different types of hardware. This involves developing runtime systems that can handle the complexities of heterogeneous hardware, such as managing memory across different types of devices and scheduling computations to minimize data transfer. Consider the task of image processing, in which different filters or transformations are applied to an image. The way these operations are scheduled on a GPU can drastically affect performance. For instance, processing an image in smaller segments or "blocks" might be more efficient for certain filters, whereas other operations might benefit from processing the image as larger contiguous chunks. The choice of how to divide and process the image—whether in smaller blocks or in larger sections—can be analogous to the difference between a filter being applied in real time versus experiencing a noticeable delay. In the runtime system, such computations must be adeptly scheduled to minimize data transfer overheads and to fully harness the hardware's capabilities.
- **Interoperability:** The language and toolchain should provide mechanisms that allow for interoperability with the existing libraries and frameworks that are designed to work with exotic hardware. This can involve providing Foreign Function Interface (FFI) mechanisms, like those in Project Panama, that allow Java code to interoperate with native libraries.
- **Library support:** Although some libraries have been developed to support exotic hardware, these libraries are often specific to certain types of hardware and may not be available or optimized for all types of hardware offered by cloud providers. Specializations in these libraries can include optimizations for specific hardware architectures, which can lead to significant performance differences when running on different types of hardware.

These considerations significantly influence the design and implementation of the projects that aim to better leverage exotic hardware capabilities. For instance, the design of Aparapi, an API for expressing data parallel workloads, has been heavily influenced by the need for language abstractions that can express parallelism and take advantage of the unique features of exotic hardware. Similarly, the development of TornadoVM has been guided by the need for a runtime system that can manage and schedule computations on different types of hardware.

In the case studies that follow, we will delve into these projects in more detail, exploring how they address the challenges of exotic hardware utilization and how they are influenced by the considerations discussed earlier.

Case Studies

It's important to ground our discussion of exotic hardware and the JVM in real-world examples, and what better way to do that than through a series of case studies. Each of these projects—LWJGL, Aparapi, Project Sumatra, CUDA4J (the joint IBM–NVIDIA project), TornadoVM, and Project Panama—offers unique insights into the challenges and opportunities presented by hardware accelerators.

- **LWJGL (Lightweight Java Game Library)**[12] serves as a foundational case, demonstrating the use of the Java Native Interface (JNI) to enable Java applications to interact with

[12] www.lwjgl.org

native APIs for a range of tasks, including graphics and compute operations. It provides a practical example of how existing JVM mechanisms can be used to leverage specialized hardware.

- **Aparapi** showcases how language abstractions can be designed to express parallelism and take advantage of the unique features of heterogeneous hardware. It translates Java bytecode into OpenCL at runtime, enabling the execution of parallel operations on the GPU.
- **Project Sumatra**, although no longer active, was a significant effort to enhance the JVM's ability to offload computations to the GPU by leaning on Java 8's Stream API to express parallelism. The experiences and lessons from Project Sumatra continue to inform current and future projects.
- Running in parallel to Project Sumatra, **CUDA4J** emerged as a joint effort by IBM and NVIDIA.[13] This project leveraged the Java 8 Stream API, enabling Java developers to write GPU computations that the CUDA4J framework translated into CUDA kernels. It demonstrated the power of collaborative efforts in the community to enhance the compatibility between the JVM and exotic hardware, particularly GPUs.
- **TornadoVM** demonstrates how runtime systems can be developed to manage and schedule computations on different types of hardware. It provides a practical solution to the challenges of hardware heterogeneity and API compatibility.
- **Project Panama** provides a glimpse into the future of the JVM. It focuses on improving the connection between the JVM and foreign APIs, including libraries and hardware accelerators, through the introduction of a new FFM API and a Vector API. These developments represent a significant evolution in JVM design, enabling more efficient and streamlined interactions with exotic hardware.

These projects were chosen not only for their technical innovations but also for their relevance to the evolving landscape of JVM and specialized hardware. They represent significant efforts in the community to adapt the JVM to better leverage the capabilities of exotic hardware, and as such, provide valuable insights for our discussion.

LWJGL: A Baseline Example

LWJGL serves as a prime example of how Java interacts with exotic hardware. This mature and widely used library provides Java developers with access to a range of native APIs, including those for graphics (OpenGL, Vulkan), audio (OpenAL), and parallel computation (OpenCL). Developers can use LWJGL in their Java applications to access these native APIs. For instance, a developer might use LWJGL's bindings for OpenGL to render 3D graphics in a game or simulation. This involves writing Java code that calls methods in the LWJGL library, which in turn invoke the corresponding functions in the native OpenGL library.

[13] Despite its niche nature, CUDA4J continues to find application on IBM Power platforms with NVIDIA hardware: www.ibm.com/docs/en/sdk-java-technology/8?topic=only-cuda4j-application-programming-interface-linux-windows.

Here is a simple example of how one might use LWJGL to create an OpenGL context and clear the screen:

```
try (MemoryStack stack = MemoryStack.stackPush()) {
    GLFWErrorCallback.createPrint(System.err).set();
    if (!glfwInit()) {
        throw new IllegalStateException("Unable to initialize GLFW");
    }
    long window = glfwCreateWindow(800, 600, "Shrek: The Musical", NULL, NULL);
    glfwMakeContextCurrent(window);
    GL.createCapabilities();
    while (!glfwWindowShouldClose(window)) {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        // Set the color to swamp green
        glClearColor(0.13f, 0.54f, 0.13f, 0.0f);
        // Draw a simple 3D scene...
        drawScene();
        // Draw Shrek's house
        drawShrekHouse();
        // Draw Lord Farquaad's castle
        drawFarquaadCastle();
        glfwSwapBuffers(window);
        glfwPollEvents();
    }
}
```

In this example, we're creating a simple 3D scene for a game based on *Shrek: The Musical*. We start by initializing GLFW and creating a window. We then make the window's context current and create an OpenGL context. In the main loop of our game, we clear the screen, set the color to swamp green (to represent Shrek's swamp), draw our 3D scene, and then draw Shrek's house and Lord Farquaad's castle. We then swap buffers and poll for events. This is a very basic example, but it gives you a sense of how you might use LWJGL to create a 3D game in Java.

LWJGL and the JVM

LWJGL interacts with the JVM primarily through JNI, the standard mechanism for calling native code from Java. When a Java application calls a method in LWJGL, the LWJGL library uses JNI to invoke the corresponding function in the native library (such as `OpenGL.dll` or `OpenAL.dll`). This allows the Java application to leverage the capabilities of the native library, even though the JVM itself does not directly support these capabilities.

Figure 9.1 illustrates the flow of control and data between Java code, LWJGL, and the native libraries. At the top is the application that runs on top of the JVM. This application uses the Java APIs provided by LWJGL to interact with the native APIs. The LWJGL provides a bridge between the JVM and the native APIs, allowing the application to leverage the capabilities of the native hardware.

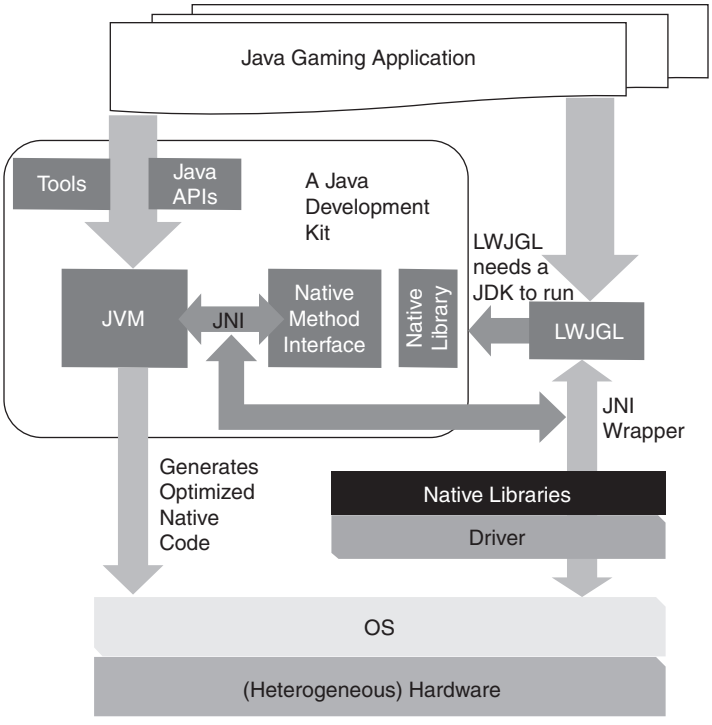

Figure 9.1 LWJGL and the JVM

The LWJGL uses JNI to call native libraries—represented by the JNI Wrapper in Figure 9.1. The JNI Wrapper is essentially "glue code" that converts variables between Java and C and calls the native libraries. The native libraries, in turn, interact with the drivers that control the hardware. In the case of LWJGL, this interaction encompasses everything from rendering visuals and processing audio to supporting various parallel computing tasks.

The use of JNI has both benefits and drawbacks. On the positive side, JNI allows Java applications to access a wide range of native libraries and APIs, enabling them to leverage exotic hardware capabilities that are not directly supported by the JVM. However, JNI also introduces overhead, as calls to native methods are generally slower than calls to Java methods. Additionally, JNI requires developers to write and maintain glue code that bridges the gap between Java and native code, which can be complex and error prone.

Challenges and Limitations

Although JNI has been successful in providing Java developers with access to native APIs, it also highlights some of the challenges and limitations of this approach. One key challenge is the complexity of working with native APIs, which often have different design philosophies and conventions than Java. This can lead to inefficiencies, especially when dealing with low-level details such as memory management and error handling.

Additionally, LWJGL's reliance on JNI to interface with native libraries comes with its own set of challenges. These challenges are highlighted by the insights of Gary Frost, an architect at Oracle, who brings a rich history in bridging Java with exotic hardware:

- **Memory management mismatch:** The JVM is highly autonomous in managing memory. It controls the memory allocation and deallocation life cycle, providing automatic garbage collection to manage object lifetimes. This control, however, can clash with the behavior of native libraries and hardware APIs that require manual control over memory. APIs for GPUs and other accelerators often rely on asynchronous operations, which can lead to a mismatch with the JVM's memory management style. These APIs commonly allow data movement requests and kernel dispatches to return immediately, with the actual operations being queued for later execution. This latency-hiding mechanism is crucial for achieving peak performance on GPUs and similar hardware, but its asynchronous nature conflicts with the JVM's garbage collector, which may relocate Java objects at any time. The pointers passed to the GPU through JNI may become invalid in mid-operation, necessitating the use of functions like `GetPrimitiveArrayCritical` to "pin" the objects and prevent them from being moved. However, this function can pin objects only until the corresponding JNI call returns, forcing Java applications to artificially wait until data transfers are complete, thereby hindering the performance benefits of asynchronous operations. It's worth noting that this memory management mismatch is not unique to LWJGL. Other Java frameworks and tools, such as Aparapi, and TornadoVM (as of this writing), also face similar challenges in this regard.
- **Performance overhead:** Transitions between Java and native code via JNI are generally slower than pure Java calls due to the need to convert data types and manage different calling conventions. This overhead can be relatively minor for applications that make infrequent calls to native methods. However, for performance-critical tasks that rely heavily on native APIs, JNI overhead can become a substantial performance bottleneck. The forced synchronization between the JVM and native libraries merely exacerbates this issue.
- **Complexity and maintenance:** Using JNI requires a firm grasp of both Java and C/C++ because developers must write "glue code" to bridge the two languages. This need for bilingual proficiency and the translation between differing paradigms can introduce complexities, making writing, debugging, and maintaining JNI code a challenging task.

LWJGL and JNI remain instrumental in allowing Java applications to access native libraries and leverage exotic hardware capabilities, but understanding their nuances is crucial, especially in performance-critical contexts. Careful design and a thorough understanding of both the JVM and the target native libraries can help developers navigate these challenges and harness the full power of exotic hardware through Java. LWJGL provides a baseline against which we can compare other approaches, such as Aparapi, Project Sumatra, and TornadoVM, which we will discuss in the following sections.

Aparapi: Bridging Java and OpenCL

Aparapi, an acronym for "A PARallel API," is a Java API that allows developers to implement parallel computing tasks. It serves as a bridge between Java and OpenCL. Aparapi provides a way

for Java developers to offload computation to GPUs or other OpenCL-capable devices, thereby leveraging the remarkable coordinated computation power of these devices.

Using Aparapi in Java applications involves defining parallel tasks using annotations and classes provided by the Aparapi API. Once these tasks are defined, Aparapi takes care of translating the Java bytecode into OpenCL, which can then be executed on the GPU or other hardware accelerators.

To truly appreciate Aparapi's capabilities, let's venture into the world of astronomy, where an adaptive optics technique is used in telescopes to enhance the performance of optical systems by adeptly compensating for distortions caused by wavefront aberrations.[14] A pivotal element of an adaptive optics system is the deformable mirror, a sophisticated device that is used to correct the distorted wavefronts.

In my work at the Center for Astronomical Adaptive Optics (CAAO), we used adaptive optics systems to correct for atmospheric distortions in real time. This task required processing large amounts of sensor data to adjust the shape of the telescope's mirror—a task that could greatly benefit from parallel computation. Aparapi could potentially be used to offload these computations to a GPU or other OpenCL-capable device, enabling the CAAO team to process the wavefront data in parallel. This would significantly speed up the correction process and allow the team to adjust the telescope more quickly and accurately to changing atmospheric conditions.

```
import com.amd.aparapi.Kernel;
import com.amd.aparapi.Range;

public class WavefrontCorrection {
    public static void main(String[] args) {
        // Assume wavefrontData is a 2D array representing the distorted wavefront
        final float[][] wavefrontData = getWavefrontData();

        // Create a new Aparapi Kernel, a unit of computation designed for wavefront correction
        Kernel kernel = new Kernel() {
          // The run method defines the computation. It will be executed in parallel on the GPU.
            @Override
            public void run() {
               // Get the global id, a unique identifier for each work item (computation)
                int x = getGlobalId(0);
                int y = getGlobalId(1);

                // Perform the wavefront correction computation.
                // This is a placeholder; the actual computation would depend on the specifics of
                   the adaptive optics system
                wavefrontData[x][y] = wavefrontData[x][y] * 2;
                // In this example, each pixel of the wavefront data is simply doubled.
```

[14] https://en.wikipedia.org/wiki/Adaptive_optics

```
                // In real-world applications, implement your wavefront correction algorithm here.
            }
        };

        // Execute the kernel with a Range representing the size of the wavefront data
        // Range.create2D creates a two-dimensional range equal to the size of the wavefront data.
        // This determines the work items (computations) that will be parallelized for the GPU.
        kernel.execute(Range.create2D(wavefrontData.length, wavefrontData[0].length));
    }
}
```

In this code, `getWavefrontData()` is a method that retrieves the current wavefront data from the adaptive optics system. The computation inside the `run()` method of the kernel would be replaced with the actual computation required to correct the wavefront distortions.

NOTE When working with Aparapi, it's essential for Java developers to have an understanding of the OpenCL programming and execution models. This is because Aparapi translates the Java code into OpenCL to run on the GPU. The code snippet shown here demonstrates how a wavefront correction algorithm can be implemented using Aparapi, but numerous intricacies must also be considered in such a use case. For instance, 2D Java arrays might not be directly supported in Aparapi due to the need to flatten memory for GPU execution. Additionally, this code won't execute in a standard sequential Java manner; instead, a wrapper method is required to invoke this code if the function is unoptimized. This highlights the importance of being aware of the underlying hardware and execution models when leveraging Aparapi for GPU acceleration.

Aparapi and the JVM

Aparapi offers a seamless integration with the JVM by converting Java bytecode into executable OpenCL code that can be executed on the GPU or other hardware accelerators. This translation is performed by the Aparapi compiler and runtime, which are optimized for OpenCL. The JVM then offloads the execution of this code to the GPU via Aparapi and OpenCL, enabling Java applications to leverage significant performance improvements for data parallel workloads.

Figure 9.2 illustrates the flow from a Java application using Aparapi to computation offloading. The left side of the diagram shows the JVM, within which our Java application operates. The application (on the right) utilizes Aparapi to define a *Kernel*[15] that performs the computation we want to offload. The "OpenCL-Enabled Compiler and Runtime" then translates this *Kernel* from Java bytecode into OpenCL code. The resultant OpenCL code is executed on the GPU or another OpenCL-capable device, as depicted by the OpenCL arrow leading to the GPU.

[15] https://www.javadoc.io/doc/com.aparapi/aparapi/1.9.0/com/aparapi/Kernel.html

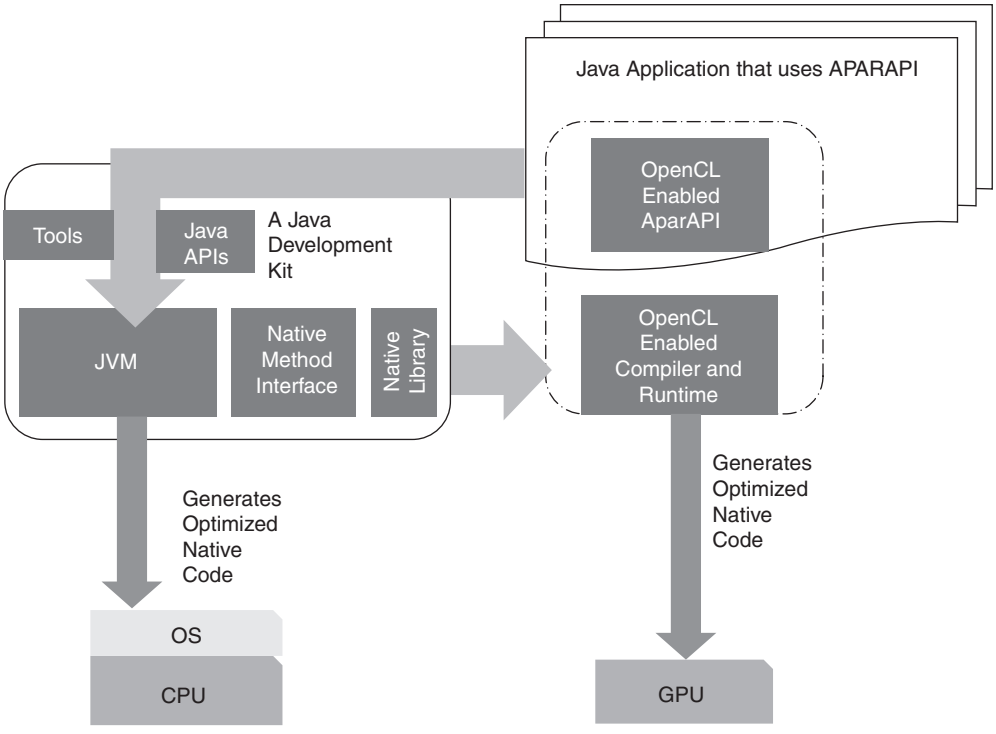

Figure 9.2 Aparapi and the JVM

Challenges and Limitations

Aparapi offers a promising avenue for Java developers to harness the computational prowess of GPUs and other hardware accelerators. However, like any technology, it presents its own set of challenges and limitations:

- **Data transfer bottleneck:** One of the main challenges lies in managing the data transfer between the CPU and the GPU. Data transfer can be a significant bottleneck, especially when working with large amounts of data. Aparapi provides mechanisms such as the ability to specify array access types (read-only, write-only, or read-write on the GPU) to help optimize the data transfer.
- **Explicit memory management:** Java developers are accustomed to automatic memory management courtesy of the garbage collector. In contrast, OpenCL demands explicit memory management, introducing a layer of complexity and potential error sources. Aparapi's approach, which translates Java bytecode into OpenCL, does not support dynamic memory allocation, further complicating this aspect.

- **Memory limitations:** Aparapi cannot exploit the advantages of constant or local memory, which can be crucial for optimizing GPU performance. In contrast, TornadoVM automatically leverages these memory types through JIT compiler optimizations.[16]
- **Java subset and anti-pattern:** Aparapi supports only a subset of the Java language. Features such as exceptions and dynamic method invocation are not supported, requiring developers to potentially rewrite or refactor their code. Furthermore, the Java kernel code, while needing to pass through the Java compiler, is not actually intended to run on the JVM. As Gary Frost puts it, this approach of using Java syntax to represent "intent" but not expecting the JVM to execute the resulting bytecode represents an anti-pattern in Java and can make it difficult to leverage GPU features effectively.

It's important to recognize that these challenges are not unique to Aparapi. Platforms such as TornadoVM, OpenCL, CUDA, and Intel oneAPI[17] also support specific subsets of their respective languages, often based on C/C++. In consequence, developers working with these platforms need to be aware of the specific features and constructs that are supported and adjust their code accordingly.

In conclusion, Aparapi represents a step forward in enabling Java applications to leverage the power of exotic hardware. Ongoing development and improvements are likely to address current challenges, making it even easier for developers to harness the power of GPUs and other hardware accelerators in their Java applications.

Project Sumatra: A Significant Effort

Project Sumatra was a pioneering initiative in the landscape of JVM and high-performance hardware. Its primary goal was to enhance the JVM's ability to work with GPUs and other accelerators. The project aimed to enable Java applications to offload data parallel tasks to the GPU directly from within the JVM itself. This was a significant departure from the traditional model of JVM execution, which primarily targeted CPUs.

Project Sumatra introduced several key concepts and components to achieve its goals. One of the most significant was the Heterogeneous System Architecture (HSA) and the HSA Intermediate Language (HSAIL). HSAIL is a portable intermediate language that is finalized to hardware ISA at runtime. This enabled the dynamic generation of optimized native code for GPUs and other accelerators.

Furthermore, the coherency between CPU and GPU caches provided by HSA obviated the need for moving data across the system bus to the accelerator, streamlining the offloading process for Java applications and enhancing performance.

Another key component of Project Sumatra was the Graal JIT compiler, which we briefly explored in Chapter 8, "Accelerating Time to Steady State with OpenJDK HotSpot VM." Graal

[16] Michail Papadimitriou, Juan Fumero, Athanasios Stratikopoulos, and Christos Kotselidis. *Automatically Exploiting the Memory Hierarchy of GPUs Through Just-in-Time Compilation*. Paper presented at the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'21), April 16, 2021. https://research.manchester.ac.uk/files/190177400/MPAPADIMITRIOU_VEE2021_GPU_MEMORY_JIT_Preprint.pdf.

[17] www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html

is a dynamic compiler that can be used as a JIT compiler within the JVM. Within Sumatra's framework, Graal was used to generate HSAIL code from Java bytecode, which could then be executed on the GPU.

Project Sumatra and the JVM

Project Sumatra was designed to be integrated closely with the JVM. It explored the integration of components such as the "Graal JIT Backend" and "HSA Runtime" into the JVM architecture, as depicted in Figure 9.3. These enhancements to the architecture allowed Java bytecode to be compiled into HSAIL code, which the HSA runtime could use to generate optimized native code that could run on the GPU (in addition to the existing JVM components that generate optimized native code for the CPU).

A prime target of Project Sumatra was the enhancement of the Java 8 Stream API to execute on the GPU, where the operations expressed in the Stream API would be offloaded to the GPU for processing. This approach would be particularly beneficial for compute-intensive tasks such as correcting numerous wavefronts in the astronomy scenario described earlier—the parallelized computation capabilities of the GPU could be leveraged to perform these corrections more quickly than would be possible on the CPU. However, to utilize these enhancements, specialized hardware that supports the HSA infrastructure, like AMD's Accelerated Processing Units (APUs), was required.

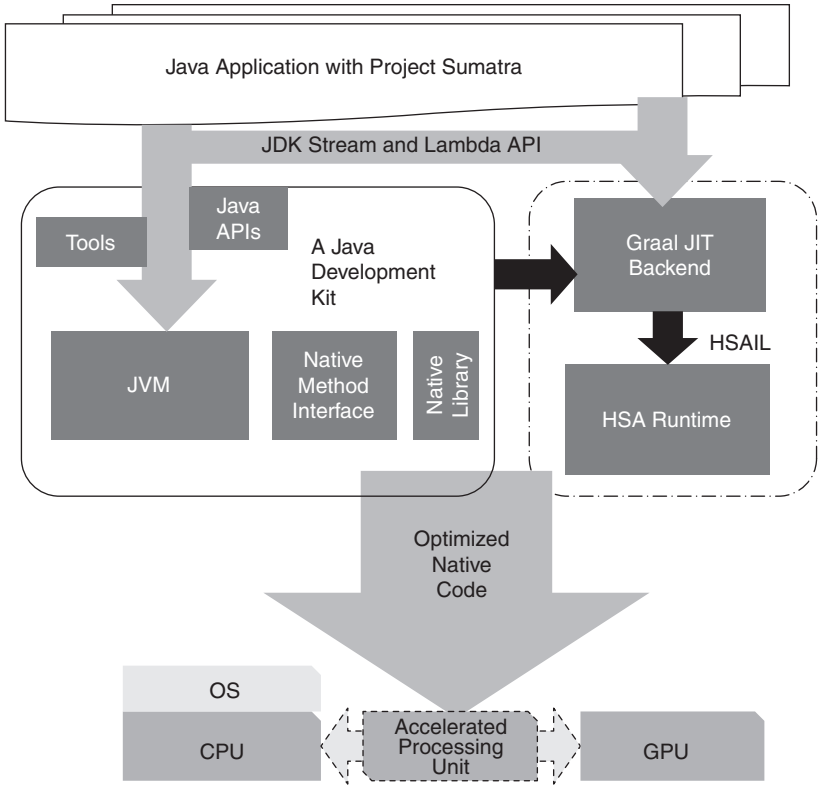

Figure 9.3 Project Sumatra and the JVM

In the following example, we use the Stream API to create a parallel stream from a list of wavefronts. Then we use a lambda expression, `wavefront -> wavefront.correct()`, to correct each wavefront. The corrected wavefronts are collected into a new list.

```
// Assume we have a List of Wavefront objects called wavefronts
List<Wavefront> wavefronts = ...;

// We can use the Stream API to process these wavefronts in parallel
List<Wavefront> correctedWavefronts = wavefronts.parallelStream()
    .map(wavefront -> {
        // Here, we're using a lambda expression to define the correction operation
        Wavefront correctedWavefront = wavefront.correct();
        return correctedWavefront;
    })
    .collect(Collectors.toList());
```

In a traditional JVM, this code would be executed in parallel on the CPU. However, with Project Sumatra, the goal was to allow such data parallel computations to be offloaded to the GPU. The JVM would recognize that this computation could be offloaded to the GPU, generate the appropriate HSAIL code, and execute it on the GPU.

Challenges and Lessons Learned

Despite its ambitious goals, Project Sumatra faced several challenges. One of the main challenges was the complexity of mapping Java's memory model and exception semantics to the GPU. Additionally, the project was closely tied to the HSA, which limited its applicability to HSA-compatible hardware.

A significant part of this project was led by Tom Deneau, Eric Caspole, and Gary Frost. Tom, with whom I had the privilege of working at AMD along with Gary, was the team lead for Project Sumatra. Beyond being just a colleague, Tom was also a mentor, whose deep understanding of the Java memory model and guidance played an instrumental role in Project Sumatra's accomplishments. Collectively, Tom, Eric, and Gary's extraordinary efforts allowed Project Sumatra to push the boundaries of Java's integration with the GPU, implementing advanced features such as thread-local allocation on the GPU and safe pointing from the GPU back to the interpreter.

However, despite the significant progress made, Project Sumatra was ultimately discontinued. The complexity of modifying the JVM to support GPU execution, along with the challenges of keeping pace with rapidly evolving hardware and software ecosystems, led to this decision.

In the realm of JVM and GPU interaction, the J9/CUDA4J offering deserves a noteworthy mention. The team behind J9/CUDA4J decided to use a Stream-based programming model similar to that included in Project Sumatra, and they continue to support their solution to date. Although J9/CUDA4J extends beyond the scope of this book, acknowledging its existence and contribution paints a more comprehensive picture of Java's journey toward embracing GPU computation.

Despite the discontinuation of Project Sumatra, its contributions to the field remain invaluable. It demonstrated the potential benefits of offloading computation to the GPU, while shedding light on the challenges that must be addressed to fully harness these benefits. The knowledge gained from Project Sumatra continues to inform ongoing and future projects in the area of JVM and hardware accelerators. As Gary puts it, we pushed hard on the JVM through Project Sumatra, and you can see its "ripples" in projects like the Vector API, Value Types, and Project Panama.

TornadoVM: A Specialized JVM for Hardware Accelerators

TornadoVM is a plug-in to OpenJDK and GraalVM that allows developers to run Java programs on heterogeneous or specialized hardware. According to Dr. Fumero, TornadoVM's vision actually extends beyond mere compilation for exotic hardware: It aims to achieve both code and performance portability. This is realized by harnessing the unique features of accelerators, encompassing not just the compilation process but also intricate aspects like data management and thread scheduling. By doing so, TornadoVM seeks to provide a holistic solution for Java developers delving into the realm of heterogeneous hardware computing.

TornadoVM offers a refined API that allows developers to express parallelism in their Java applications. This API is structured around tasks and annotations that facilitate parallel execution. The `TaskGraph`, which is included in the newer versions of TornadoVM, is central to defining the computation, while the `TornadoExecutionPlan` specifies execution parameters. The `@Parallel` annotation can be used to mark methods that should be offloaded to accelerators.

Revisiting our adaptive optics system scenario for a large telescope, suppose we have a large array of wavefront sensor data that requires real-time processing to correct for atmospheric distortions. Here's an updated example of how you might use TornadoVM in this context:

```
import uk.ac.manchester.tornado.api.TaskGraph;

public class TornadoExample {
    public static void main(String[] args) {
        // Create a task graph
        TaskGraph taskGraph = new TaskGraph("s0")
           .transferToDevice(DataTransferMode.FIRST_EXECUTION, wavefronts)
           .task("t0", TornadoExample::correctWavefront, wavefronts, correctedWavefronts)
           .transferToHost(DataTransferMode.EVERY_EXECUTION, correctedWavefronts);

        // Execute the task graph
        ImmutableTaskGraph itg = taskGraph.snapshot();
        TornadoExecutionPlan executionPlan = new TornadoExecutionPlan(itg);

        executionPlan.execute();
    }
```

```
    public static void correctWavefront(float[] wavefronts, float[] correctedWavefronts, int N) {
        for (@Parallel int i = 0; i < wavefronts.length; i++) {
           for (@Parallel int j = 0; j < wavefronts[i].length; j++) {
               correctedWavefronts[i * N + j] = wavefronts[i * N + j] * 2;
           }
        }
    }
}
```

In this example, a `TaskGraph` named `s0` is created; it outlines the sequence of operations and data transfers. The `wavefronts` data is transferred to the device (like a GPU) before computation starts. The `correctWavefront` method, added as a task to the graph, processes each wavefront to correct it. Post computation, the corrected data is transferred back to the host (like a CPU).

According to Dr. Fumero, TornadoVM (like Aparapi) doesn't support user-defined objects, but rather uses a collection of predefined ones. In our example, `wavefronts` would likely be a `float` array. Java developers using TornadoVM should be acquainted with OpenCL's programming and execution models. The code isn't executed in typical sequential Java fashion. If the function is deoptimized, a wrapper method is needed to invoke this code. A challenge arises when handling 2D Java arrays: The memory might require flattening unless the GPU can re-create the memory layout—a concern reminiscent of the issues with Aparapi.

TornadoVM and the JVM

TornadoVM is designed to be integrated closely with the JVM. It extends the JVM to exploit the expansive concurrent computation power of specialized hardware, thereby boosting the performance of JVM-based applications.

Figure 9.4 offers a comprehensive summary of the architecture of TornadoVM at a very high level. As you can see this diagram, TornadoVM introduces several new components to the JVM, each designed to optimize Java applications for execution on heterogenous hardware:

- **API (task graphs, execution plans, and annotations):** Developers use this interface to define tasks that can be offloaded to the GPU. Tasks are defined using Java methods, which are annotated to indicate that they can be offloaded, as shown in the earlier example.
- **Runtime (data optimizer + bytecode generation):** The TornadoVM runtime is responsible for optimizing data movement between the host (CPU) and the device (GPU). It uses a task-based programming model in which each task is associated with a data descriptor. The data descriptor provides information about the data size, shape, and type, which TornadoVM uses to manage data transfers between the host and the device. TornadoVM also provides a caching mechanism to avoid unnecessary data transfers. If the data on the device is up-to-date, TornadoVM can skip the data transfer and directly use the cached data on the device.

- **Execution engine (bytecode interpreter + OpenCL +PTX + SPIR-V/LevelZero drivers):** Instead of generating bytecode for execution on the accelerator, TornadoVM uses the bytecode to orchestrate the entire application from the host side. This approach, while an implementation detail, offers extensive possibilities for runtime optimizations, such as dynamic task migration and batch processing, while maintaining a clean design.[18]
- **JIT compiler + memory management:** TornadoVM extends the Graal JIT compiler to generate code optimized for GPUs and other accelerators. Recent changes in memory management mean each Java object now possesses its individual memory buffer on the target accelerator, mirroring Aparapi's approach. This shift enhances the system's flexibility, allowing multiple applications to share a single GPU, which is ideal for cloud setups.

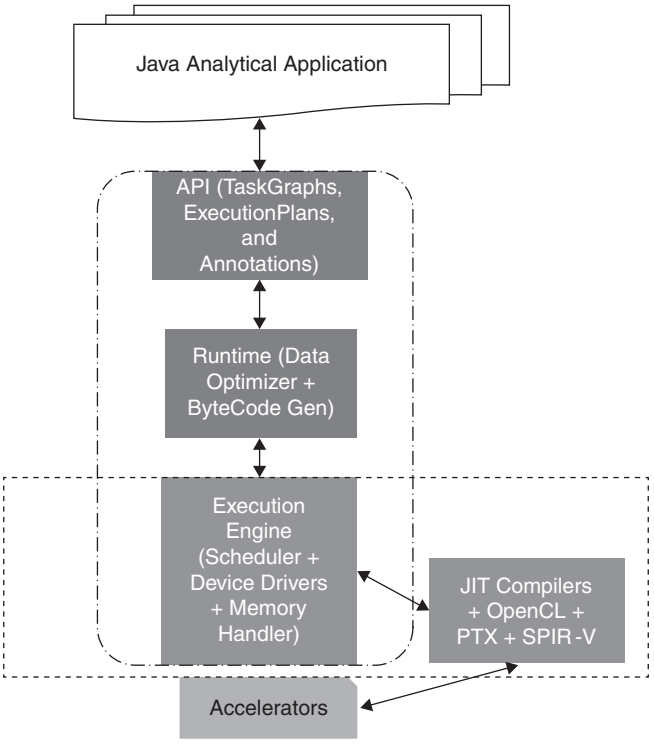

Figure 9.4 TornadoVM

[18] Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, James Clarkson, and Christos Kotselidis. "Dynamic Application Reconfiguration on Heterogeneous Hardware." In *Proceedings of the 15th ACMSIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '19)*, April 14, 2019. https://jjfumero.github.io/files/VEE2019_Fumero_Preprint.pdf.

In a traditional JVM, Java bytecode is executed on the CPU. However, with TornadoVM, Java methods that are annotated as tasks can be offloaded to the GPU or other accelerators. This allows TornadoVM to boost the performance of JVM-based applications.

Challenges and Future Directions

TornadoVM, like any pioneering technology, has faced its share of hurdles. One of the key challenges is the complexity of mapping Java's memory model and exception semantic to the GPU. Dr. Fumero emphasizes that TornadoVM shares certain challenges with Aparapi, such as managing nonblocking operations on GPUs due to the garbage collector. As with Aparapi, data movement between the CPU and accelerators can be challenging. TornadoVM addresses this challenge with a data optimizer that minimizes data transfers. This is possible though the Task-Graph API, which can accommodate multiple tasks, with each task pointing to an existing Java method.

TornadoVM continues to evolve and adapt, pushing the boundaries of what's possible with the JVM and exotic hardware. It serves as a testament to the potential benefits of offloading computation to the GPU and underscores the challenges that need to be surmounted to realize these benefits.

Project Panama: A New Horizon

Project Panama has made significant strides in the domain of Java performance optimization for specialized hardware. It is designed to enhance the JVM's interaction with foreign functions and data, particularly those associated with hardware accelerators. This initiative marks a notable shift from the conventional JVM execution model, historically centered on standard processors.

Figure 9.5 is a simplified block diagram showing the key components of Project Panama—the Vector API and the FFM API.

> **NOTE** For the purpose of this chapter, and this section in particular, I have used the latest build available as of writing this book: JDK 21 EA.

The Vector API

A cornerstone of Project Panama is the Vector API, which introduces a new way for developers to express computations that should be carried out over vectors (that is, sequences of values of the same type). Specifically, this interface is designed to perform vector computations that are compiled at runtime to optimal vector hardware instructions on compatible CPUs.

The Vector API facilitates the use of Single Instruction Multiple Data (SIMD) instructions, a type of parallel computing instruction set. SIMD instructions allow a single operation to be performed on multiple data points simultaneously. This is particularly useful in tasks such as graphics processing, where the same operation often needs to be performed on a large set of data.

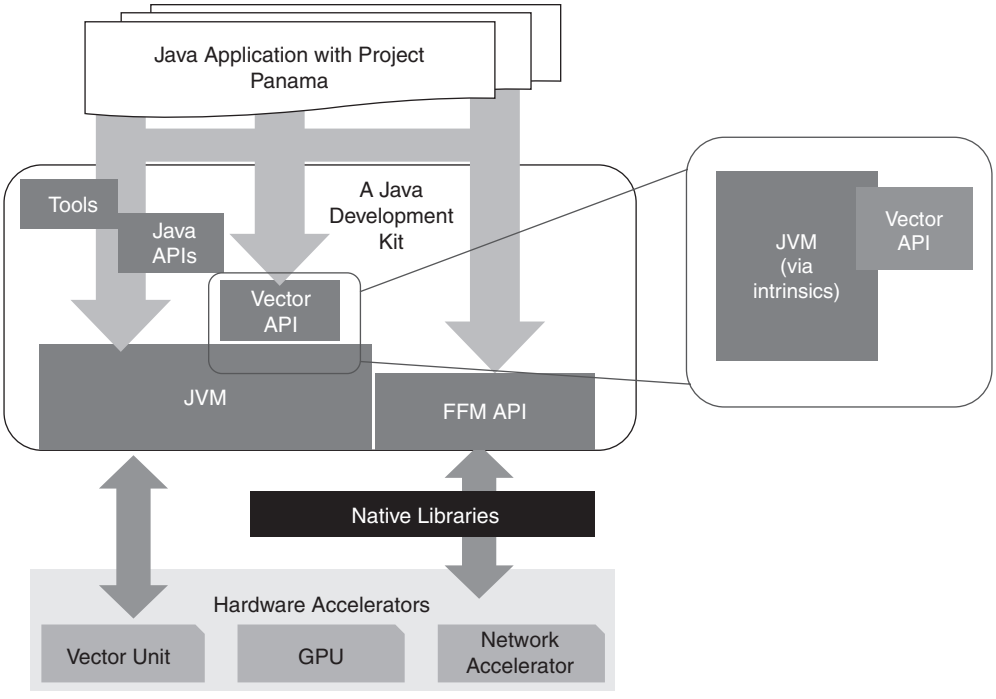

Figure 9.5 Project Panama

As of this book's writing, the Vector API resides in the `jdk.incubator.vector` module in the Project Panama early-access builds. It provides several benefits:

- **Vector computation expression:** The API offers a suite of vector operations that can be employed to articulate computations over vectors. Each operation is applied element-wise, ensuring uniformity in the processing of each element in the vector.
- **Optimal hardware instruction compilation → enhanced portability:** The API is ingeniously designed to compile these computations at runtime into the most efficient vector hardware instructions on applicable architectures. This unique feature allows the same Java code to leverage vector instructions across a diverse range of CPUs, without necessitating any modifications.
- **Leveraging SIMD instructions → performance boost:** The API harnesses the power of SIMD instructions, which can dramatically enhance the performance of computations over large data sets.
- **Correspondence of high-level operations to low-level instructions → increased developer productivity:** The API is architected in such a way that high-level vector operations correspond directly to low-level SIMD instructions via intrinsics. Thus, developers can write code at a high level of abstraction, while still reaping the performance benefits of low-level hardware instructions.

- **Scalability:** The Vector API offers a scalable solution for processing vast amounts of data in a concurrent manner. As data sets continue to expand, the ability to perform computations over vectors becomes increasingly crucial.

The Vector API proves particularly potent in the domain of graphics processing, where it can swiftly apply operations over large pixel data arrays. Such vector operations enable tasks like image filter applications to be executed in parallel, significantly accelerating processing times. This efficiency is crucial when uniform transformations, such as color adjustments or blur effects, are required across all pixels.

Before diving into the code, let's clarify the `factor` used in our example. In image processing, applying a filter typically involves modifying pixel values—`factor` represents this modification rate. For instance, a `factor` less than 1 dims the image for a darker effect, while a value greater than 1 brightens it, enhancing the visual drama in our *Shrek: The Musical* game.

With this in mind, the following example demonstrates the application of a filter using the Vector API in our game development:

```
import jdk.incubator.vector.*;

public class ImageFilter {
    public static void applyFilter(float[] shrekPixels, float factor) {
        // Get the preferred vector species for float
        var species = FloatVector.SPECIES_PREFERRED;
        // Process the pixel data in chunks that match the vector species length
        for (int i = 0; i < shrekPixels.length; i += species.length()) {
            // Load the pixel data into a vector
            var musicalVector = FloatVector.fromArray(species, shrekPixels, i);
            // Apply the filter by multiplying the pixel data with the factor
            var result = musicalVector.mul(factor);
            // Store the result back into the pixel data array
            result.intoArray(shrekPixels, i);
        }
    }
}
```

Through the code snippet shown here, `FloatVector.SPECIES_PREFERRED` allows our filter application to scale across different CPU architectures by leveraging the widest available vector registers, optimizing our SIMD execution. The `mul` operation then methodically applies our intended filter effect to each pixel, adjusting the image's overall brightness.

As the Vector API is evolving, it currently remains in the *incubator* stage—the `jdk.incubator.vector` package is part of the `jdk.incubator.vector` module in JDK 21. Incubator modules are modules that hold features that are not yet standardized but are made available for developers to try out and provide feedback. It is through this iterative process that robust features are refined and eventually standardized.

To get hands-on with the `ImageFilter` program and explore these capabilities, some changes to the build configuration are necessary. The Maven `pom.xml` file needs to be updated to ensure

that the `jdk.incubator.vector` module is included during compilation. The required configuration is as follows:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.11.0</version>
            <configuration>
                <source>21</source>
                <target>21</target>
                <compilerArgs>
                    <arg>--add-modules</arg>
                    <arg>jdk.incubator.vector</arg>
                </compilerArgs>
            </configuration>
        </plugin>
    </plugins>
</build>
```

The Foreign Function and Memory API

Another crucial aspect of Project Panama is the FFM API (also called the FF&M API), which allows a program written in one language to call routines or use services written in another program. Within the scope of Project Panama, the FFM API allows Java code to interoperate seamlessly with native libraries. This glue-less interface to foreign code also supports direct calls to foreign functions and is integrated with the JVM's method handling and linking mechanisms. Thus, it is designed to supersede the JNI by offering a more robust, Java-centric development model, and by facilitating seamless interactions between Java programs and code or data that resides outside the Java runtime.

The FFM API offers a comprehensive suite of classes and interfaces that empower developers to interact with foreign code and data. As of the writing of this book, it is part of the `java.lang.foreign` module in the JDK 21 early-access builds. This interface provides tools that enable client code in libraries and applications to perform the following functions:

- **Foreign memory allocation:** FFM allows Java applications to allocate memory space outside of the Java heap. This space can be used to store data that will be processed by foreign functions.
- **Structured foreign memory access:** FFM provides methods for reading from and writing to the allocated foreign memory. This includes support for structured memory access, so that Java applications can interact with complex data structures in foreign memory.

- **Life-cycle management of foreign resources:** FFM includes mechanisms for tracking and managing the life cycle of foreign resources. This ensures that memory is properly deallocated when it is no longer needed, preventing memory leaks.
- **Foreign function invocation:** FFM allows Java applications to directly call functions in foreign libraries. This provides a seamless interface between Java and other programming languages, enhancing interoperability.

The FFM API also brings about several nonfunctional benefits that can enhance the overall development experience:

- **Better performance:** By enabling direct invocation of foreign functions and access to foreign memory, FFM bypasses the overhead associated with JNI. This results in performance improvements, especially for applications that rely heavily on interactions with native libraries.
- **Security and safety measures:** FFM provides a safer avenue for interacting with foreign code and data. Unlike JNI, which can lead to unsafe operations if misused, FFM is designed to ensure safe access to foreign memory. This reduces the risk of memory-related errors and security vulnerabilities.
- **Better reliability:** FFM provides a more robust interface for interacting with foreign code, thereby increasing the reliability of Java applications. It mitigates the risk of application crashes and other runtime errors that can occur when using JNI.
- **Ease of development:** FFM simplifies the process of interacting with foreign code. It offers a pure-Java development model, which is more intuitive to use and understand than JNI. This facilitates developers in writing, debugging, and maintaining code that interacts with native libraries.

To illustrate the use of the FFM API, let's consider another example from the field of adaptive optics. Suppose we're working on a system for controlling the secondary mirror of a telescope, and we want to call a native function to adjust the mirror. Here's how we could do this using FFM:

```
import java.lang.foreign.*;
import java.util.logging.*;

public class MirrorController {
public static void adjustMirror(float[] secondaryMirrorAdjustments) {
        // Get a lookup object from the native linker
        var lookup = Linker.nativeLinker().defaultLookup();
        // Find the native function "adjustMirror"
        var adjustMirrorSymbol = lookup.find("adjustMirror").get();
        // Create a memory segment from the array of adjustments
        var adjustmentArray = MemorySegment.ofArray(secondaryMirrorAdjustments);
        // Define the function descriptor for the native function
        var function = FunctionDescriptor.ofVoid(ValueLayout.ADDRESS);
```

```
        // Get a handle to the native function
        var adjustMirror=Linker.nativeLinker().downcallHandle(adjustMirrorSymbol,function);
        try {
            // Invoke the native function with the address of the adjustment array
            adjustMirror.invokeExact(adjustmentArray.address());
        } catch (Throwable ex) {
            // Log any exceptions that occur
            Logger.getLogger(MirrorController.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

In this example, we're adjusting the secondary mirror using the `secondaryMirrorAdjustments` array. The code is written using the FFM API in JDK 21.[19] To use this feature, you need to enable preview features in your build system. For Maven, you can do this by adding the following code to your `pom.xml`:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.11.0</version>
            <configuration>
                <compilerArgs>--enable-preview</compilerArgs>
                <release>21</release>
            </configuration>
        </plugin>
    </plugins>
</build>
```

The evolution of the FFM API from JDK 19 to JDK 21 demonstrates a clear trend toward simplification and enhanced usability. A notable change is the shift from using the `SegmentAllocator` class for memory allocation to the more streamlined `MemorySegment.ofArray()` method. This method directly converts a Java array into a memory segment, significantly reducing the complexity of the code, enhancing its readability, and making it easier to understand. As this API continues to evolve, further changes and improvements are likely in future JDK releases.

Challenges and Future Directions

Project Panama is still evolving, and several challenges and areas of future work remain to be addressed. One of the main challenges is the task of keeping pace with the ever-evolving hardware and software technologies. Staying abreast of the many changes in this dynamic field

[19] The FFM API is currently in its third preview mode; that means this feature is fully specified but not yet permanent, so it's subject to change in future JDK releases.

requires continual updates to the Vector API and FFM API to ensure they remain compatible with these devices.

For example, the Vector API needs to be updated as new vector instructions are added to CPUs, and as new types of vectorizable data are introduced. Similarly, the FFM API needs to be updated as new types of foreign functions and memory layouts are introduced, and as the ways in which these functions and layouts are used and evolved.

Another challenge is the memory model. The Java Memory Model is designed for on-heap memory, but Project Panama introduces the concept of off-heap memory. This raises questions about how to ensure memory safety and how to integrate off-heap memory with the garbage collector.

In terms of future work, one of the main areas of focus is improving the performance of the Vector and FFM APIs. This includes optimizing the JIT compiler to generate more efficient code for vector operations and improving the performance of foreign function calls and memory accesses.

Another area under investigation is improving the usability of the APIs. This includes providing better tools for working with vector data and foreign functions and improving the error messages and debugging support.

Finally, ongoing work seeks to integrate Project Panama with other parts of the Java ecosystem. This includes integrating it with the Java language and libraries, and with tools such as the Java debugger and profiler.

Envisioning the Future of JVM and Project Panama

As we stand on the precipice of technological advancement, the horizon promises that a transformative era for the JVM and Project Panama lies ahead. Drawing from my experience and understanding of the field, I'd like to share my vision for the future. Figure 9.6 illustrates one possible use case, in which a gaming application utilizes accelerators via the FFM and Vector APIs with the help of high-level JVM-language APIs.

High-Level JVM-Language APIs and Native Libraries

I anticipate the development of advanced JVM-language APIs designed to interface directly with native libraries. A prime example is the potential integration with NVIDIA's RAPIDS project, which is a suite of software libraries dedicated to data science and analytics pipelines on GPUs.[20] RAPIDS leverages NVIDIA's CUDA technology, a parallel computing platform and API model, to optimize low-level compute operations on CUDA-enabled GPUs. By developing JVM APIs that can interface with such native APIs, we could potentially simplify the development process, ensuring efficient hardware utilization. This would allow a broader range of developers—including those who may not have deep expertise in low-level hardware programming—to harness the power of hardware accelerators.

[20] "RAPIDS Suite of AI Libraries." *NVIDIA Developer*. https://developer.nvidia.com/rapids.

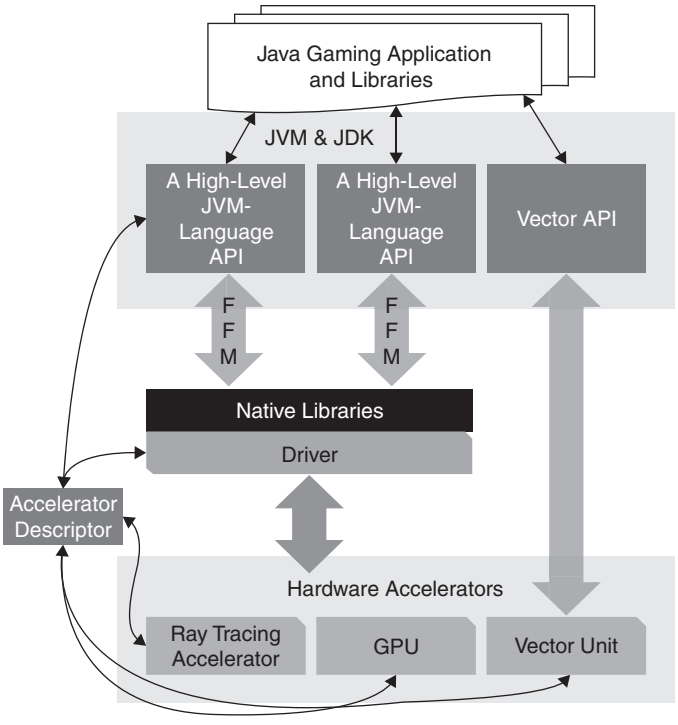

Figure 9.6 A Gaming Application Utilizing Accelerators via the FFM API and Vector API with the Help of High-Level JVM-Language APIs

Vector API and Vectorized Data Processing Systems

The Vector API, when synergized with vectorized data processing systems such as Apache Spark and vector databases, has the potential to revolutionize analytical processing. By performing operations on whole data vectors concurrently rather than on discrete data points, it can achieve significant speed improvements. This API's hardware-agnostic optimizations promise to further enhance such processing, allowing developers to craft efficient code suitable for a diverse array of hardware architectures:

- **Vector databases and Vector API:** Vector databases offer scalable search and analysis of high-dimensional data. The Vector API, with its platform-agnostic optimizations, could further scale these operations.
- **Analytical queries, Apache Spark, and Vector API:** Spark utilizes vectorized operations within its query optimizer for enhanced performance. Integrating the Vector API into this process could further accelerate the execution of analytical queries, capitalizing on the API's ability to optimize across different hardware architectures.

- **Parquet columnar storage file format:** The Parquet columnar storage file format for the Hadoop ecosystem could potentially benefit from the Vector API. The compressed format of Parquet files could be efficiently processed using vector operations, potentially improving the performance of data processing tasks across the Hadoop platform.

Accelerator Descriptors for Data Access, Caching, and Formatting

Within this envisioned future, accelerator descriptors stand out as a key innovation. These metadata frameworks aim to standardize data access, caching, and formatting specifications for hardware accelerator processing. Functioning as a blueprint, they would guide the JVM in fine-tuning data operations to the unique traits and strengths of each accelerator type. The creation of such a system is not just a technical challenge—it requires anticipatory thinking about the trajectory of data-centric computing. By abstracting these specifications, developers could more readily tailor their applications to exploit the full spectrum of hardware accelerators available, simplifying what is currently a complex optimization process.

To realize this vision, meticulous design and collaborative implementation are essential. The Java community must unite in both embracing and advancing these enhancements. In doing so, we can ensure that the JVM not only matches the pace of technological innovation but also redefines benchmarks for performance, usability, and flexibility.

This endeavor is about harmonizing the JVM's solid foundation with the capabilities of cutting-edge hardware accelerators, aiming for significant performance gains without complicating the platform's inherent versatility. Maintaining the JVM's general-purpose nature while optimizing for modern hardware is a delicate balance and a prime focus for Project Panama and the JVM community at large. With the appropriate developments, we are poised to enter a new epoch where the JVM excels in high-performance scenarios, fully leveraging the potential of modern hardware accelerators.

The Future Is Already Knocking at the Door!

As I ventured deeper into the intricate world of JVM and hardware acceleration, it was both enlightening and exhilarating to discover a strong synergy between my vision and the groundbreaking work of Gary Frost and Dr. Fumero. Despite our diverse research trajectories, the collective revelations at JVMLS 2023 showcased a unified vision of the evolution of this domain.

Gary's work with Hardware Accelerator Toolkit (HAT)[21] was a testament to this forward-thinking vision. Built on the foundation of the FFM API, HAT is more than just a toolkit—it's an adaptable solution across various environments. It includes the *ndrange* API,[22] FFM data-wrapping patterns, and an abstraction for vendor-specific runtimes to facilitate the use of hardware accelerators.

The *ndrange* API in HAT, drawing inspiration from TornadoVM, further enhances these capabilities. Complementing HAT's offerings, Project Panama stands out for its powerful functional API and efficient data management strategies. Considering that GPUs require specific data layouts, Project Panama excels in creating GPU-friendly data structures.

[21] www.youtube.com/watch?v=IbKBu3lTftc

[22] https://man.opencl.org/ndrange.html

Project Babylon[23] is an innovative venture emerging as a pivotal development for enhancing the integration of Java with diverse programming models, including GPUs and SQL. It employs code reflection to standardize and transform Java code, enabling effective execution on diverse hardware platforms. This initiative, which complements Project Panama's efforts in native code interoperation, signifies a transformative phase for Java in leveraging advanced hardware capabilities.

Another highlight of the discussion at JVMLS 2023 was Dr. Fumero's presentation on TornadoVM's innovative vision for a Hybrid API.[24] As he explained, this API seamlessly merges the strengths of native and JIT-compiled code, enabling developers to tap into speed-centric, vendor-optimized libraries. The seamless integration of Project Panama with this Hybrid API ensures uninterrupted data flow and harmonizes JIT-compiled tasks with library calls, paving the way for a more cohesive and efficient computational journey.

In conclusion, our individual journeys in exploring JVM, Project Panama, and hardware acceleration converge on a shared destination, emphasizing the immense potential of this frontier. We stand on the brink of a transformative era in computing, one in which the JVM will truly unlock the untapped potential of modern hardware accelerators.

Concluding Thoughts: The Future of JVM Performance Engineering

As we draw this comprehensive exploration of JVM performance engineering to a close, it's essential to reflect on the journey we've undertaken. From the historical evolution of Java and its virtual machine to the nuances of its type system, modularity, and memory management, we've traversed the vast landscape of JVM intricacies. Each chapter has been a deep dive into specific facets of the JVM, offering insights, techniques, and tools to harness its full potential.

The culmination of this journey is the vision of the future—a future where the JVM doesn't just adapt but thrives, leveraging the full might of modern hardware accelerators. Our discussions of Project Panama, the Vector API, and more have painted a vivid picture of what lies ahead. It's a testament to the power of collaboration and shared vision.

The tools, APIs, and frameworks we've discussed are the building blocks of tomorrow, and I encourage you to explore them while applying your own expertise, insights, and passion. The JVM community is vibrant, and ever evolving. Your contributions, experiments, and insights we collectively share will shape its future trajectory. Engage with this community. Dive deep into the new tools, push their boundaries, and share your findings. Every contribution, every line of code, every shared experience adds a brick to the edifice we're building.

Thank you for joining me on this enlightening journey.
Let's continue to explore, innovate, and shape the future of
JVM performance engineering together.

[23] https://openjdk.org/projects/babylon/

[24] Juan Fumero. *From CPU to GPU and FPGAs: Supercharging Java Applications with TornadoVM*. Presentation at JVM Language Summit 2023, August 7, 2023. https://github.com/jjfumero/jjfumero.github.io/blob/master/files/presentations/TornadoVM-JVMLS23v2-clean.pdf.

Index

Numerics

Symbols

A

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X - Y - Z

This page intentionally left blank

ORACLE

Goodbye, database complexity. Hello, simplicity.

Reduce complexity and cost with the one database for all your workloads. Cloud native, converged, and automated, Oracle Autonomous Database eliminates manual maintenance hassles and delivers top performance, security, and automatic scaling for your changing needs. It also supports the full development lifecycle for modern applications.

Get simplicity in the cloud or on-premises.

Oracle Autonomous Database
Learn more at
oracle.com/autonomous-database

ORACLE

Oracle VM VirtualBox

Oracle's open source, cross-platform, desktop hypervisor helps deliver code faster

Deploy your applications quickly and securely, on-premises and to the cloud, with more productivity and simpler operations, at a lower cost.

- Cloud integration
- Runs on any desktop
- Full VM encryption

oracle.com/virtualbox

Register Your Product at informit.com/register

Access additional benefits and save up to 65%* on your next purchase

- Automatically receive a coupon for 35% off books, eBooks, and web editions and 65% off video courses, valid for 30 days. Look for your code in your InformIT cart or the Manage Codes section of your account page.
- Download available product updates.
- Access bonus material if available.**
- Check the box to hear from us and receive exclusive offers on new editions and related products.

InformIT—The Trusted Technology Learning Source

InformIT is the online home of information technology brands at Pearson, the world's leading learning company. At informit.com, you can

- Shop our books, eBooks, and video training. Most eBooks are DRM-Free and include PDF and EPUB files.
- Take advantage of our special offers and promotions (informit.com/promotions).
- Sign up for special offers and content newsletter (informit.com/newsletters).
- Access thousands of free chapters and video lessons.
- Enjoy free ground shipping on U.S. orders.*

** Offers subject to change.*

*** Registration benefits vary by product. Benefits will be listed on your account page under Registered Products.*

Connect with InformIT—Visit informit.com/community

Pearson

informIT

Addison-Wesley • Adobe Press • Cisco Press • Microsoft Press • Oracle Press • Peachpit Press • Pearson IT Certification • Que