

ORACLE  
PRESS



# JVM

# Performance Engineering

Inside OpenJDK and the  
HotSpot Java Virtual Machine

ORACLE

Monica Beckwith

# JVM Performance Engineering

---

# JVM Performance Engineering: Inside OpenJDK and the HotSpot Java Virtual Machine

## Table of Contents

Cover

Half Title

Title Page

Copyright Page

Contents

Preface

Acknowledgments

About the Author

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

A New Ecosystem Is Born

A Few Pages from History

Understanding Java HotSpot VM and Its Compilation Strategies

The Evolution of the HotSpot Execution Engine

Interpreter and JIT Compilation

Print Compilation

Tiered Compilation

Client and Server Compilers

Segmented Code Cache

Adaptive Optimization and Deoptimization

# **Table of Contents**

## **HotSpot Garbage Collector: Memory Management Unit**

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

Young Collections and Weak Generational Hypothesis

Old-Generation Collection and Reclamation Triggers

Parallel GC Threads, Concurrent GC Threads, and Their Configuration

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

Java 1.1 to Java 1.4.2 (J2SE 1.4.2)

Java 5 (J2SE 5.0)

Java 6 (Java SE 6)

Java 7 (Java SE 7)

Java 8 (Java SE 8)

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

Java 17 (Java SE 17)

## **Embracing Evolution for Enhanced Performance**

## **2 Performance Implications of Java's Type System Evolution**

Java's Primitive Types and Literals Prior to J2SE 5.0

Java's Reference Types Prior to J2SE 5.0

Java Interface Types

Java Class Types

Java Array Types

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

Enumerations

Annotations

Other Noteworthy Enhancements (Java SE 8)

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

Variable Handle Typed Reference

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

# **Table of Contents**

Switch Expressions

Sealed Classes

Records

## **Beyond Java 17: Project Valhalla**

Performance Implications of the Current Type System

The Emergence of Value Classes: Implications for Memory Management

Redefining Generics with Primitive Support

Exploring the Current State of Project Valhalla

Early Access Release: Advancing Project Valhallas Concepts

Use Case Scenarios: Bringing Theory to Practice

A Comparative Glance at Other Languages

## **Conclusion**

## **3 From Monolithic to Modular Java: A Retrospective and Ongoing Evolution**

### **Introduction**

### **Understanding the Java Platform Module System**

Demystifying Modules

Modules Example

Compilation and Run Details

Introducing a New Module

### **From Monolithic to Modular: The Evolution of the JDK**

### **Continuing the Evolution: Modular JDK in JDK 11 and Beyond**

### **Implementing Modular Services with JDK 17**

Service Provider

Service Consumer

A Working Example

Implementation Details

### **JAR Hell Versioning Problem and Jigsaw Layers**

# **Table of Contents**

Working Example: JAR Hell

Implementation Details

## **Open Services Gateway Initiative**

OSGi Overview

Similarities

Differences

## **Introduction to Jdeps, Jlink, Jdeprscan, and Jmod**

Jdeps

Jdeprscan

Jmod

Jlink

## **Conclusion**

Performance Implications

Tools and Future Developments

Embracing the Modular Programming Paradigm

## **4 The Unified Java Virtual Machine Logging Interface**

The Need for Unified Logging

Unification and Infrastructure

Performance Metrics

Tags in the Unified Logging System

Log Tags

Specific Tags

Identifying Missing Information

Diving into Levels, Outputs, and Decorators

Levels

Decorators

Outputs

Practical Examples of Using the Unified Logging System

# **Table of Contents**

Benchmarking and Performance Testing

Tools and Techniques

Optimizing and Managing the Unified Logging System

Asynchronous Logging and the Unified Logging System

Benefits of Asynchronous Logging

Implementing Asynchronous Logging in Java

Best Practices and Considerations

Understanding the Enhancements in JDK 11 and JDK 17

JDK 11

JDK 17

Conclusion

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

Introduction

Performance Engineering: A Central Pillar of Software Engineering

Decoding the Layers of Software Engineering

Performance: A Key Quality Attribute

Understanding and Evaluating Performance

Defining Quality of Service

Success Criteria for Performance Requirements

Metrics for Measuring Java Performance

Footprint

Responsiveness

Throughput

Availability

Digging Deeper into Response Time and Availability

The Mechanics of Response Time with an Application Timeline

The Role of Hardware in Performance

# **Table of Contents**

Decoding HardwareSoftware Dynamics

Performance Symphony: Languages, Processors, and Memory Models

Enhancing Performance: Optimizing the Harmony

Memory Models: Deciphering Thread Dynamics and Performance Impacts

Concurrent Hardware: Navigating the Labyrinth

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

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

Concurrent Memory Access and Coherency in Multiprocessor Systems

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

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

## **Performance Engineering Methodology: A Dynamic and Detailed Approach**

Experimental Design

Bottom-Up Methodology

Top-Down Methodology

Building a Statement of Work

The Performance Engineering Process: A Top-Down Approach

Building on the Statement of Work: Subsystems Under Investigation

Key Takeaways

## **The Importance of Performance Benchmarking**

Key Performance Metrics

The Performance Benchmark Regime: From Planning to Analysis

Benchmarking JVM Memory Management: A Comprehensive Guide

Why Do We Need a Benchmarking Harness?

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

Getting Started with Maven

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

Benchmark Phases: Warm-Up and Measurement



# Table of Contents

Loop Optimizations and @OperationsPerInvocation

Benchmarking Modes in JMH

Understanding the Profilers in JMH

Key Annotations in JMH

JVM Benchmarking with JMH

Profiling JMH Benchmarks with perfasm

Conclusion

## 6 Advanced Memory Management and Garbage Collection in OpenJDK

Introduction

Overview of Garbage Collection in Java

Thread-Local Allocation Buffers and Promotion-Local Allocation Buffers

Optimizing Memory Access with NUMA-Aware Garbage Collection

Exploring Garbage Collection Improvements

G1 Garbage Collector: A Deep Dive into Advanced Heap Management

Advantages of the Regionalized Heap

Optimizing G1 Parameters for Peak Performance

Z Garbage Collector: A Scalable, Low-Latency GC for Multi-terabyte Heaps

Future Trends in Garbage Collection

Practical Tips for Evaluating GC Performance

Evaluating GC Performance in Various Workloads

Types of Transactional Workloads

Synthesis

Live Data Set Pressure

Understanding Data Lifespan Patterns

Impact on Different GC Algorithms

Optimizing Memory Management

## 7 Runtime Performance Optimizations: A Focus on Strings,

# **Table of Contents**

## **Locks, and Beyond**

### **Introduction**

### **String Optimizations**

Literal and Interned String Optimization in HotSpot VM

String Deduplication Optimization and G1 GC

Reducing Strings Footprint

### **Enhanced Multithreading Performance: Java Thread**

#### **Synchronization**

The Role of Monitor Locks

Lock Types in OpenJDK HotSpot VM

Code Example and Analysis

Advancements in Javas Locking Mechanisms

Optimizing Contention: Enhancements since Java 9

Visualizing Contended Lock Optimization: A Performance Engineering  
Exercise

Synthesizing Contended Lock Optimization: A Reflection

Spin-Wait Hints: An Indirect Locking Improvement

### **Transitioning from the Thread-per-Task Model to More Scalable Models**

Traditional One-to-One Thread Mapping

Increasing Scalability with the Thread-per-Request Model

Reimagining Concurrency with Virtual Threads

### **Conclusion**

## **8 Accelerating Time to Steady State with OpenJDK HotSpot VM**

### **Introduction**

JVM Start-up and Warm-up Optimization Techniques

Decoding Time to Steady State in Java Applications

# **Table of Contents**

Ready, Set, Start up!

Phases of JVM Start-up

Reaching the Applications Steady State

An Applications Life Cycle

## **Managing State at Start-up and Ramp-up**

State During Start-up

Transition to Ramp-up and Steady State

Benefits of Efficient State Management

Class Data Sharing

Ahead-of-Time Compilation

## **GraalVM: Revolutionizing Javas Time to Steady State**

### **Emerging Technologies: CRIU and Project CRaC for Checkpoint/Restore Functionality**

### **Start-up and Ramp-up Optimization in Serverless and Other Environments**

Serverless Computing and JVM Optimization

Containerized Environments: Ensuring Swift Start-ups and Efficient Scaling

GraalVMs Present-Day Contributions

Key Takeaways

## **Boosting Warm-up Performance with OpenJDK HotSpot VM**

Compiler Enhancements

Segmented Code Cache and Project Leyden Enhancements

The Evolution from PermGen to Metaspace: A Leap Forward Toward Peak  
Performance

## **Conclusion**

## **9 Harnessing Exotic Hardware: The Future of JVM Performance Engineering**

Introduction to Exotic Hardware and the JVM

Exotic Hardware in the Cloud

# **Table of Contents**

Hardware Heterogeneity

API Compatibility and Hypervisor Constraints

Performance Trade-offs

Resource Contention

Cloud-Specific Limitations

The Role of Language Design and Toolchains

Case Studies

LWJGL: A Baseline Example

Aparapi: Bridging Java and OpenCL

Project Sumatra: A Significant Effort

TornadoVM: A Specialized JVM for Hardware Accelerators

Project Panama: A New Horizon

Envisioning the Future of JVM and Project Panama

High-Level JVM-Language APIs and Native Libraries

Vector API and Vectorized Data Processing Systems

Accelerator Descriptors for Data Access, Caching, and Formatting

The Future Is Already Knocking at the Door!

Concluding Thoughts: The Future of JVM Performance Engineering

Index