Chapter 1: Introduction

Java performance is influenced by a multitude of factors, both internal and external to the code. These factors include:

- **The specific version of the Java Development Kit (JDK) being used.**
- **The efficiency and quality of the written code.**
- **The volume of code that needs to be compiled.**

Even seemingly minor version changes in the JDK can have a significant impact on performance, both positively and negatively. This highlights the importance of staying updated with JDK releases and understanding their potential performance implications. The concept of "**premature optimization**" is introduced, suggesting that optimization efforts should be targeted and based on actual performance analysis rather than assumptions or guesses.

Chapter 2: A Java Performance Toolbox

This chapter focuses on tools and techniques for measuring and analysing Java application performance:

Microbenchmarking

- **Ensure results are used:** The JVM may discard unused microbenchmark calculations.
- **Exclude unrelated operations:** Focus on the specific code being benchmarked.
- **Use realistic inputs:** Mimic real-world data for accurate performance assessments.

Macrobenchmarking

- **Measure the entire application:** The best performance indicator is the application itself, running with its external dependencies.
- **Mesobenchmarks:** Target specific application components for focused performance analysis.
- **Hardware impact:** Consider the influence of hardware on system performance.

Performance Measurement Techniques

- **Elapsed time (batch):** Measure performance over multiple iterations to account for system warm-up.
- **Throughput:** Report operations per second (TPS, RPS, OPS) after a warm-up period.
- **Response time:** Measure in percentiles to account for variations and user think time. Consider average response time for a broader user perspective.

Performance Testing Best Practices

- **Early and frequent regression testing:** Catch performance regressions early in the development cycle.
- **Automate everything:** Automate performance tests and baselining for consistent results.
- **Measure everything:** Collect comprehensive data for thorough performance analysis.
- **Run on target systems:** Different systems exhibit different performance characteristics.

Performance Monitoring Tools

- **Utilize JDK and OS tools:** Leverage tools like jcmd, jconsole, jhat, jmap, jinfo, jstack, jstat, and jvisualvm for monitoring various aspects of JVM performance.

CPU Usage

- **User time vs. system time:** Differentiate between time spent executing application code (user time) and time spent by the OS on behalf of the application (system time).
- **CPU utilization:** Higher CPU utilization indicates more effective code execution.
- **Ideal CPU time:** Leave some idle CPU time for handling future tasks.
- **Single CPU vs. multi-CPU:** Optimise for non-blocking threads in multi-CPU environments to maximize CPU utilization.
- **Processor queue length:** Monitor thread queue length to assess CPU load.

Disk I/O

- Disk I/O can be a significant performance bottleneck. Monitor disk usage and optimise data access patterns to minimise disk I/O overhead.

Key Takeaways
• CPU time is a primary performance indicator. • Optimise code to increase CPU usage (for shorter periods). • Investigate low CPU usage before attempting to tune an application. • Disk I/O is another common bottleneck.

Chapter 3: Working with the JIT Compiler

This chapter focuses on the Just-in-Time (JIT) compiler and its role in optimising Java code execution:

Interpreters vs. Compilers

- **Interpreters:** Execute code line by line, offering portability but potentially slower execution.
- **Compilers:** Convert code into native machine code, providing faster execution but limited portability.
- **Java's Approach:** Java uses an intermediate language (bytecodes) that is JIT-compiled into native code during execution, combining portability with performance.

JIT Compilation

  - **Hotspots:** Identify frequently executed code sections (hotspots) for optimisation.
  - **Compilation Trigger:** The JVM decides to compile code based on:
    - Number of method calls

        - Loop branch frequency
    - **On-Stack Replacement (OSR):** Compiles and replaces code while it's running, allowing loops to be optimised mid-execution.

JIT Compiler Flavours

- **Client Compiler:** Favours faster compilation times over highly optimised code.
- **Server Compiler:** Generates more optimised code for long-running applications, potentially at the cost of longer compilation times.
- **Tiered Compilation:** Uses the client compiler initially and switches to the server compiler for hotspots. Enabled by default in Java 8 and later.

JIT Compiler Tuning

- **Code Cache:** Adjust the size of the code cache (where compiled code is stored) to accommodate frequently used code.
- **Compilation Threshold:** Fine-tune the threshold that triggers compilation based on application behaviour.

Advanced Compiler Tuning

- **Compilation Threads:** Control the number of threads used for compilation.
- **Inlining:** Enable/disable inlining (replacing method calls with the method body) to improve performance.
- **Escape Analysis:** The JVM analyses object scope to optimise memory allocation.
- **Deoptimization:** The compiler can revert optimisations if they become invalid.

Tiered Compilation Levels

- Different levels of compilation (0-4) represent varying levels of optimisation.
    - 0: Interpreted Code
    - 1: Simple C1 compiled code
    - 2: Limited C1 complied code
    - 3: Full C1 complied code
    - 4: C2 compiled code
- Optimal path is typically 0 → 3 → 4.
- Monitor compilation logs to ensure efficient code compilation.

Key Takeaways

- Tiered compilation and sufficient code cache size are essential for JIT performance.
- Use compilation logs (PrintCompilation) to analyse code compilation behaviour.
- Advanced compiler tuning is usually not necessary, but understanding its options can be beneficial in specific scenarios.

Chapter 5: An Introduction to Garbage Collection

This chapter introduces the different garbage collection (GC) algorithms in Java and basic GC tuning:

Garbage Collection Basics

- **Automatic Memory Management:** Java automatically reclaims memory occupied by unused objects, relieving developers from manual memory management.
- **GC Process:** Involves finding unused objects, freeing their memory, and compacting the heap.
- **Generations:** The heap is divided into generations (young and old) to optimise GC efficiency.
- **Minor GC:** Collects garbage from the young generation.
- **Full GC:** Collects garbage from the entire heap.

Choosing a GC Algorithm

- **Consider**:
    - CPU resources
    - Acceptable pause times
    - Application performance requirements
- **Trade-offs:** Throughput vs. response time.

GC Algorithms

- **Serial Collector:** Single-threaded, suitable for single-CPU machines or small heaps.
- **Throughput Collector:** Parallel, default for server-class machines. Efficient for batch jobs.
- **CMS Collector:** Concurrent, minimises pause times but uses more CPU. Suitable for response-time-sensitive applications.
- **G1 Collector:** Concurrent, designed for larger heaps, divides the heap into regions.

Basic GC Tuning

- **Heap Sizing:**
    - Set appropriate initial (-Xms) and maximum (-Xmx) heap sizes.
    - Aim for 30% heap occupancy after full GC.
    - Avoid exceeding physical memory.
- **Generation Sizing:**
    - Tune the sizes of the young and old generations using parameters like NewRatio, NewSize, MaxNewSize, and -Xmn.
- **Permgen/Metaspace Sizing:**

    - Java 7 and earlier: Permgen
    - Java 8 and later: Metaspace
    - Tune using PermSize/MaxPermSize or MetaspaceSize/MaxMetaspaceSize.
- **Parallelism:** Control the number of GC threads using ParallelGCThreads.
- **Adaptive Sizing:** The JVM automatically adjusts generation sizes for optimal performance.

GC Tools

- **GC Logging:** Enable GC logs using -verbose:gc or -XX:+PrintGC.
- **GC Histogram:** Analyse GC logs for insights into GC behaviour.
- **jconsole:** Real-time monitoring of heap usage.
- **jstat:** Command-line tool for GC statistics.

Key Takeaways

- Balance heap size for optimal performance.
- Consider application characteristics (throughput vs. response time).
- Utilize GC logs and monitoring tools.
- Adaptive sizing is generally effective.
- Tune parallelism based on CPU resources.

Chapter 6: Garbage Collection Algorithms

This chapter delves deeper into the specific GC algorithms:

Throughput Collector

- Key operations: Minor collections and full GCs
- GC logs are essential for analysis.
- Adaptive Sizing: The collector adjusts heap size based on pause time goals (MaxGCPauseMillis, GCTimeRatio).

CMS Collector

- Key operations:
    - Young generation collection (stop-the-world)
    - Concurrent old generation cleanup
    - Full GC (if necessary)
- Types of failures
    - Concurrent Cycles: CMS cleans the old generation concurrently with application threads.
    - Concurrent Mode Failure: Occurs when the old generation fills up before a concurrent cycle completes.
    - Promotion Failure: Happens when the old generation is too fragmented to accommodate promoted objects.
- Permgen/Metaspace: CMS can collect Permgen/Metaspace (with specific flags).

G1 Collector

- Regions: The heap is divided into regions, allowing for more targeted GC.
- **Garbage First:** Prioritises regions with the most garbage for efficient collection.
- **Full GC Triggers:** Concurrent Mode Failure, Promotion Failure, Evacuation Failure, Humongous Allocation Failure.
- **Tuning Goals:**
- Avoid concurrent and evacuation failures
- Minimise pause times

  G1 Tuning Tips
    - Increase old generation size.
    - Increase background thread count.
    - Tune background activity frequency.
    - Increase work done in mixed GC cycles.

Key Takeaways

- Each GC algorithm has strengths and weaknesses.
- Understand the failure modes of each algorithm.
- Tune GC settings based on application behaviour and performance goals.

Chapter 7: Heap Memory Best Practices

This chapter focuses on best practices for managing heap memory effectively:

Analysing Memory Usage

- **Histograms:** Quickly identify memory issues caused by excessive instances of specific classes.
- **Heap Dumps:**
    - Generate using jcmd or jmap.
    - Analyse using tools like jhat, jvisualvm, or MAT (Memory Analyzer Tool).

OutOfMemoryError

- Causes:
    - Lack of native memory

    - Permgen/Metaspace exhaustion
    - Insufficient heap space for live objects
    - Excessive GC activity
- Solutions:
    - Increase heap size
    - Investigate memory leaks
    - Enable heap dumps on OutOfMemoryError
    - Using Less Memory
        - Reduce Object Size: Optimise object structure to minimise memory footprint.
        - Lazy Initialization: Delay object creation until needed.
        - Consider thread safety (use volatile).
        - Eager De-Initialization: Set objects to null when no longer needed.
        - Immutable and Canonical Objects:
        - Immutable objects can improve memory efficiency.
        - Use String.intern() to reduce duplicate strings.

Object Lifecycle Management

- Object Reuse:
    - Object pools: Manage reusable objects to reduce creation overhead.
    - Thread-local variables: Efficient reuse within a thread.
- Weak and Soft References:
    - Weak references allow objects to be collected more aggressively.
    - Soft references allow objects to be kept as long as memory is available.
    - Indefinite (Phantom) are weak and soft reference disguise as strong reference – they are often cached to reused later, to avoid incurring the cost of recalculating or re-fetching.

Key Takeaways

- Analyse heap dumps to understand memory usage patterns.
- Investigate and address OutOfMemoryError root causes.
- Employ techniques to reduce memory consumption.
- Consider object reuse and weak/soft references for efficient memory management.

Chapter 8: Native Memory Best Practices

This chapter addresses managing native memory used by the JVM and applications:

Native Memory Consumption

- **Non-Heap Memory**: Native memory used outside the Java heap.
- **Sources:**
    - Thread stacks
    - Code cache
    - Direct byte buffers (NIO)

Minimising Memory Footprint

- **Control Heap Size**: The heap is a major contributor to memory footprint.
- **Limit Thread Stack Size:** Adjust thread stack sizes based on application needs.
- **Minimise Direct Byte Buffer Usage:** Use direct byte buffers sparingly.
- **Set MaxDirectMemorySize:** Limit the maximum native memory allocated to direct byte buffers.

Native Memory Tracking (NMT)

- Introduced in Java 8: Provides insights into native memory allocation (summary or detail modes).
- **Use jcmd to get real-time information.**
- **Analyse memory commitment (total and individual).**

JVM Tuning for the OS

- **Large Pages**:
    - Utilize large memory pages (e.g., 2 MB) for improved memory management efficiency.
    - Configure the operating system to support large pages.
- **Compressed OOPs**:
    - Reduces the size of object pointers in 64-bit JVMs.
    - Effective for heap sizes up to around 32 GB.

JNI and malloc()

- JNI uses malloc() for native memory allocation.
- Key considerations:

    - Free allocated memory using free().
    - Avoid memory leaks.
    - Minimise memory holding time.

Key Takeaways

- Understand sources of native memory consumption.
- Monitor and manage native memory usage.
- Utilize NMT for insights into native memory allocation.
- Consider large pages and compressed OOPs for performance optimization.
- Follow JNI best practices to avoid memory leaks and other issues.

Chapter 9: Threading and Synchronization Performance

This chapter discusses performance considerations related to threading and synchronisation in Java applications:

Thread Pool Configuration

- **Maximum Threads:** Determine the optimal number based on workload and hardware.
- **Minimum Threads:** Prevent excessive thread creation but may not be beneficial if thread starvation occurs.
- **Task Queue Size:** Adjust the task queue depth to balance task waiting times and resource utilization.
- **ThreadPoolExecutor:** Configure the executor to manage thread pool behaviour effectively.

ForkJoinPool

- Designed for parallel processing with divide-and-conquer algorithms.
- Java 8 introduces a common pool.

Automatic Parallelization

- Java does not automatically parallelize all code.
- Developers need to structure code to leverage parallel processing capabilities.

Thread Synchronization

- Minimise Code in Synchronized Blocks: Reduce the amount of code executed under synchronization to improve performance.
- **False Sharing:** Avoid performance degradation caused by multiple threads accessing different variables in the same cache line. Use padding or data structures that minimise false sharing.

JVM Thread Tuning

- Thread Stack Sizes: Adjust stack sizes to optimise memory usage.
- **Biased Locking:** Improve performance for single-threaded access but may introduce overhead for thread pools.
- **Lock Spinning:** Control how threads handle contended locks.
- **Thread Priorities:** Hints to the OS about thread importance.

Java Thread Monitoring

- **Thread State:** Use tools like jstack and jcmd to monitor thread states.
- **Blocked Threads:** Identify threads blocked on locks or I/O operations.
- **Java Flight Recorder (JFR):** Capture detailed thread events for performance analysis.
- **Real-Time Monitoring:** Observe thread activity in real time using jconsole.

Key Takeaways

- Configure thread pools appropriately for your workload.
- Structure code to leverage parallel processing.
- Minimise synchronization overhead and false sharing.
- Tune JVM thread settings for performance.
- Utilize thread monitoring tools to diagnose and resolve threading issues.