# Optimising Java Runtime Performance: A Deep Dive

Whether its dealing with memory constraints, CPU limitations, or trying to reduce latency, Java applications often require fine-tuning at the lower levels. This involves a deep dive into the runtime aspects of the application, focusing on optimizing the Java Execution Environment. This Document explores various experiments and techniques to enhance the performance of your Java applications such as:

* [Tuning the Java Code Cache](https://quip-amazon.com/Wyt6AiyHaJXy/Low-Level-Optimisations-for-Fortress#temp:C:DcI6bfaca814e6e44e69ba7b8dc4) (Preferred)
* [Optimizing the String Pool](https://quip-amazon.com/Wyt6AiyHaJXy/Low-Level-Optimisations-for-Fortress#temp:C:DcIa1c7cee66b13494b8d04ef049)  (Preferred)
* [Tuning the Garbage Collector](https://quip-amazon.com/Wyt6AiyHaJXy/Low-Level-Optimisations-for-Fortress#temp:C:DcI9946b3a33a16498da97822d8c)  (Preferred)
* [Selecting the appropriate Garbage Collector algorithm](http://Choosing Garbage Collector Algorithm)
* [Leveraging tools like Java Mission Control (JMC), Java Flight Recorder (JFR)](https://quip-amazon.com/Wyt6AiyHaJXy/Low-Level-Optimisations-for-Fortress#temp:C:DcI74dffc0f9ac24b84a1dcfbda7)
* [Java Microbenchmark Harness (JMH)](http://Benchmarking Java Applications)
* [Choosing between 32-bit and 64-bit JVMs](https://quip-amazon.com/Wyt6AiyHaJXy/Low-Level-Optimisations-for-Fortress#temp:C:DcI7e215a4e01144bb2bcc586803)

# **Experiment #1:  Tuning Java Code Cache**

### **Context**

**JIT Interpreter:**The interpreter in Java directly executes bytecode, one instruction at a time, without converting it to machine code, unlike the JIT compiler.

**JIT Compiler**: In the HotSpot JVM, which is the JVM implementation provided by Oracle and used by default in most distributions of Java, there are two Just-In-Time (JIT) compilers, known as C1 (Client compiler) and C2 (Server compiler). These compilers are responsible for converting Java bytecode into native machine code at runtime, which can then be executed directly by the hardware.

* **C1 Compiler (Client Compiler):**The C1 compiler uses a technique called "tiered compilation", where it initially compiles methods with minimal optimization to get them running quickly, and then recompiles them with more optimization if they are used frequently.
* **C2 Compiler (Server Compiler):**The C2 compiler is designed to optimize for maximum program throughput and is used in long-running server applications. It takes more time to compile methods, but it applies more aggressive optimizations, which can result in faster execution for code that is run frequently.

The C2 compiler uses a technique called "adaptive optimization", where it collects profiling information about the running program and uses this information to guide its optimizations.

#### Levels of Java Compilation:

|  |  |  |  |
| --- | --- | --- | --- |
|  | A | B | C |
| 1 | **Levels** | **Done By** | **Desciption** |
| 2 | 0 | JIT Interpreter | JVM interprets all Java code. |
| 3 | 1 | C1 | JVM compiles the code using the C1 compiler, but without collecting any profiling information. |
| 4 | 2 | C1 | JVM compiles the code using the C1 compiler with light profiling. |
| 5 | 3 | C1 | JVM compiles the code using the C1 compiler with full profiling. |
| 6 | 4 | C2 | JVM compiles the code using the C2 compiler for maximum long-term performance. |

### **Opportunities to Improve:**

When the code has been compiled to Level 4 using C2 compile, it means that it will be used a lot. So it will place that block of high performing compiled code into “Code Cache”. But, this ***code cache has limited size***.  And if there are lots of methods that are compiled to Level 4, then some of them will need to be removed from the code cache to make some space for the next one to be inserted. And the removed method could be recomposed and be added later.

So, the code cache size might not be sufficient and ***increasing the size of the code cache*** can lead to an improvement in our applications performance.

# **Experiment  #2:  Tuning the String Pool**

The string pool is implemented as a fixed capacity HashMap with each bucket containing a list of strings with the same hashCode. The default size of the table/pool is 1,009 buckets. It was a constant in the early versions of Java 6 and became configurable between Java6u30 and Java6u41.

**Until Java7u40**: String pool moved from PermGen space to heap space. You are limited only by a much higher heap size. It means that you can set the string pool size to a rather high value in advance (this value depends on your application requirements).

**Java8***:* String pool size was increased in Java7u40 (this was a major performance update) to 60013. This value allows you to have approximately 30,000 distinct strings in the pool before you start experiencing collisions.

Note: In java 8 The PermGen was eliminated and replaced by Metaspace.

### Opportunities to Improve:

Thus, the HashMap employed by the String Pool is of a **static size**. Despite our application having a larger heap size, the quantity of buckets allotted for our String Pool remains constant. This can potentially result in a higher rate of collisions.   
  
**Recommendations:**

* The StringTable statistics can be very useful for debugging performance issues in our Java application. They give you information about the number of buckets, the size of the table, the number of entries, etc. This can help you understand how the JVM is managing its memory and can help you optimize your application. Use -XX:+PrintStringTableStatistics to see all the details related to String Pool Buckets.
* If you see that your StringTable has a lot of collisions (i.e., multiple Strings hashing to the same bucket), that might be a sign that your table is too small. You can adjust the size of the StringTable using the -XX:StringTableSize JVM parameter. While passing the number, **use a Prime Number** bigger than the existing number of buckets.
* The Heap should be big enough to store the footprint required by the String Pool.

# **Experiment  #3:  Tuning Garbage Collector**

As we know in the structure of the Heap memory, we recognize two primary segments: the Young Generation and the Old Generation. The Young Generation is further divided into three parts: Eden, Survivor Space 0 (S0), and Survivor Space 1 (S1).

**Step 1:** Java Dynamically alter the size of heap sections at runtime. So, the first step will be to turn-off the dynamic resizing by using a flag: -XX:-UseAdaptiveSizePolicy  which will be enabled by default.

**Step2:** Garbage Collection that take place in young generation are better for performance than garbage collection on the old generation. So, based on the object behaviour in the memory (heap). We can minimise the garbage collections.

 Ways of trying to achieve this are:

* The Heap memory in Java, which includes the young and old generations, can be adjusted in size to optimize application performance. By increasing the memory allocation to the young generation and decreasing it for the old generation, the frequency of garbage collection in the young generation can be reduced. This results in the earlier collection of movable objects, thereby reducing their migration to the old generation. The JVM option -XX:NewRatio=n allows control over the size ratio between the Young and Old Generations. This can be used to enhance application performance based on the nature of the objects created. For applications generating many short-lived objects, increasing the Young Generation size can decrease minor garbage collections. On the other hand, for applications with many long-lived objects, expanding the Old Generation size can postpone major garbage collection.
* The JVM option -XX:SurvivorRatio=n allows you to adjust the size of the two survivor spaces (S0 and S1) in the Young Generation compared to the Eden space. For instance, -XX:SurvivorRatio=6 implies the Eden space is six times larger than a single Survivor space. This option is useful for optimizing Java application performance. If your application generates many short-lived objects, increasing the Eden space size can accommodate more objects before triggering minor garbage collection. However, if your application creates objects that survive longer but not long enough to move to the Old Generation, enlarging the Survivor spaces can be beneficial.
* The -XX:MaxTenuringThreshold=n is a JVM (Java Virtual Machine) option that you can use to control the maximum number of times that objects can be moved between the survivor spaces (S0 and S1) in the Young Generation before being promoted to the Old Generation. This option can be used to optimize the performance of your Java application. If your application creates many objects that live for a medium amount of time (long enough to survive several minor garbage collections, but not long enough to be worth moving to the Old Generation), you might want to increase the MaxTenuringThreshold to keep these objects in the Young Generation longer. Conversely, if your application creates many objects that live for a long time, you might want to decrease the MaxTenuringThreshold to move these objects to the Old Generation sooner.

**Note:** Like the -XX:NewRatio=n option, changes to -XX:SurvivorRatio=n  and -XX:MaxTenuringThreshold=n should be made cautiously and tested thoroughly, as incorrect settings can negatively impact performance or cause application failure.

# **Experiment  #4:  Choosing Garbage Collector Algorithm**

|  |  |  |  |
| --- | --- | --- | --- |
|  | A | B | C |
| 1 | **Garbage Collector** | **Description** | **Best Use Case** |
| 2 | Serial Garbage Collector | Simplest type, single-threaded, "stop-the-world" pauses. | Best for simple, single-threaded applications or for development and testing. |
| 3 | Parallel Garbage Collector (Throughput Collector) | Uses multiple threads to speed up the garbage collection process. | Best for applications with medium to large data sets and a large number of CPUs. |
| 4 | Concurrent Mark Sweep (CMS) Garbage Collector | Minimizes application pauses by doing most of its work concurrently with the application threads. | Best for applications that need to minimize latency. |
| 5 | G1 (Garbage-First) Garbage Collector | Divides the heap into regions and prioritizes collecting garbage from the regions that  contain the most garbage first. | Best for applications running on multi-core processors with large memory. |
| 6 | Z Garbage Collector (ZGC) | A scalable, low-latency garbage collector that performs all expensive work concurrently. | Best for applications that require low pause times and can spare plenty of memory. |
| 7 | Shenandoah Garbage Collector | A low-pause-time garbage collector that performs most of its work concurrently with running Java threads.  **Shenandoah** reduces GC pause times by compacting objects concurrently with running Java threads. This means that pause times with Shenandoah are independent of the heap size, providing consistent pause time whether your heap is 200 MB or 200 GB. This makes Shenandoah an excellent choice for applications that require responsiveness and predictable short pauses. | Best for applications with large heaps and significant amounts of live data. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | A | B | C | D |
| 1 | **Feature** | **Z Garbage Collector (ZGC)** | **Garbage-First (G1) Garbage Collector** | **Shenandoah Garbage Collector** |
| 2 | **Throughput** | High (Designed for server-style applications) | High (Designed for server-style applications) | High |
| 3 | **Latency** | Low (Designed for low pause times) | Moderate (Aims for predictable pause times) | Low (Designed for low pause times) |
| 4 | **Heap Size** | Handles heaps from a few hundred megabytes to multi-terabytes | Handles large heaps | Handles large heaps |
| 5 | **Concurrent Work** | Performs most work concurrently with application threads | Performs significant work concurrently with application threads | Performs most work concurrently with application threads |
| 6 | **Availability** | Available from JDK 11 | Available since JDK 7 (fully supported from JDK 9) | Available from JDK 12 |

# **Experiment #5: Java Mission Control and Java Flight Recorder- Profiling and Diagnostics**

**Java Mission Contro**l (JMC) is a profiling and diagnostics tools suite for the Java Virtual Machine (JVM). It includes tools to monitor, manage, profile, and eliminate memory leaks in Java applications. It's a part of JDK distribution.

JMC works in conjunction with the **Java Flight Recorder** (JFR), a low overhead data collection framework built into the JVM. Together, they provide a powerful toolset for detailed analysis of how the JVM and the Java application it hosts are behaving.

### Advantages of Java Mission Control:

* **Low overhead:** JMC and JFR are designed to have minimal impact on the performance of the system, making them suitable for use in production environments.
* **Detailed diagnostics:** JMC provides detailed telemetry data from a running JVM. This includes but is not limited to CPU usage, memory usage, thread activity, and garbage collection statistics.
* **Exception profiling:** JMC can provide information about exceptions that are thrown and caught, which can be useful in identifying potential issues in the code.
* **Flight Recorder:** The flight recorder tool allows you to record events in an application for future analysis. This can be very useful for post-incident analysis.

### **Key benefits of using Java Flight Recorder:**

* **Low Overhead:** JFR is designed to have a very low impact on performance, making it suitable for use in production environments.
* **Detailed Diagnostics:** JFR can collect a wealth of detailed diagnostic data from the JVM and the Java application it's running. This includes but is not limited to CPU usage, memory usage, garbage collection statistics, thread activity, and more.
* **Event-Based Tracing:** JFR uses an event-based model for collecting data, which allows it to capture a wide range of information about what's happening in your application and the JVM.
* **Long-Term Profiling:** Unlike some profiling tools that can only be used for short periods due to their impact on performance, JFR is designed to be used for long-term profiling of applications. This can provide valuable insights into the behavior of your application over time.
* **Integration with Java Mission Control:** JFR is designed to work with Java Mission Control (JMC), a suite of tools for managing, monitoring, profiling, and troubleshooting Java applications. JMC provides a user-friendly interface for viewing and analyzing the data collected by JFR.
* **Custom Events:** In addition to the standard events that JFR can record, you can define your own custom events to capture application-specific data.
* **Thread Profiling:** JFR provides detailed profiling information about individual threads, which can be invaluable for diagnosing performance issues and bottlenecks.
* **Exception Profiling:** JFR can provide information about exceptions that are thrown and caught, which can be useful in identifying potential issues in the code.
* **Flight Recordings:** The flight recordings can be extracted and analyzed offline, allowing developers to diagnose issues post-incident without affecting the running application.

# **Experiment  #6: Benchmarking Java Applications**

Java Microbenchmark Harness (JMH) is a toolkit that helps developers create, run, and analyze performance tests in Java and other languages running on the Java Virtual Machine (JVM). It is part of the OpenJDK project and is widely used for performance testing and optimization.

Here are some of the key benefits of using JMH:

1. **Accurate Microbenchmarking**: Microbenchmarking, or testing small units of code, can be tricky due to the optimizations performed by the JVM. JMH is designed to help you get accurate microbenchmark results despite these optimizations.
2. **Easy to Use**: JMH provides a set of annotations that make it easy to define benchmarks. You simply annotate a method with @Benchmark, and JMH takes care of the rest.
3. **Flexible Benchmarking Modes**: JMH supports various benchmarking modes, such as measuring average time, sample time, throughput, and more. This allows you to choose the most appropriate mode for your specific testing needs.
4. **Support for Multithreaded Benchmarks**: JMH supports multithreaded benchmarks, allowing you to measure performance in multi-threaded scenarios.
5. **Control Over JVM Options**: JMH allows you to control various JVM options and parameters, giving you more control over the benchmarking environment.
6. **Integration with Existing Tools**: JMH benchmarks are plain Java programs that can be packaged as JAR files and run from the command line or integrated with build tools like Maven or Gradle.
7. **Profiling Support**: JMH can be used in conjunction with various profilers to provide deeper insights into your code's performance.
8. **Warmup Phases**: JMH includes warmup phases in its benchmarks to ensure the JVM's Just-In-Time (JIT) compiler optimizes your code before measurements are taken, leading to more accurate results.

Remember, while JMH is a powerful tool, it's important to interpret the results correctly and understand that microbenchmarks are just one part of overall performance optimization and should be used in conjunction with other testing and profiling methods.

# **Experiment  #7:  32 bit vs 64 bit [JVM]**

If you are running a 64 bit Operating System, you can choose either 32 bit JVM or 64 bit JVM. So,

So, Why would one even consider choosing a 32-bit JVM over its advanced, muscular sibling, the 64-bit JVM?  
  
The general rule is that, if the heap size that our application need is **< 3 GB**, then *32 bit JVM will likely be faster* that 64 bit JVM.  The Reason for this is rather Technical, but it is based around the fact that, each pointer to an object in memory will be smaller (It will be 32 bit rather than 64 bit). So, manipulating these pointer will be quicker.  
  
However, to use a JVM of 32 bit, the total memory required for the application must not exceed 4 GB. Also, If our application is a heavy user of larger numeric types such as Longs and Doubles, then 32 bit JVM will be potentially slower than the 64 bit version.

**Recommendations:**

To be sure about performance improvements, it is recommended to test application in both 32 bit and 64 bit JVM.

Now If you choose 32 bit JVM, only C1 compiler will be available and If you choose 64 bit JVM, both C1 and C2 compiler will be available.

# Prioritization

|  |  |  |
| --- | --- | --- |
| **Experiment** | **Priority** | **Notes** |
| Experiment  #4:  Choosing Garbage Collector Algorithm | 1 | Experimentation with ZGC did not yield good result. We can experiment with shenendoha |
| Experiment  #2:  Tuning the String Pool | 2 | Just print table statistics |
| Experiment  #3:  Tuning Garbage Collector | 3 | Start by disabling adaptive size policy |
| Experiment #1:  Tuning Java Code Cache | 4 |  |
| Experiment #5: Java Mission Control and Java Flight Recorder- Profiling and Diagnostics | 5 | Need more info |
| Experiment  #6: Benchmarking Java Applications | 5 | Need more info |
| Experiment  #7:  32 bit vs 64 bit [JVM] | 6 | We always have memory greater than 3GB so may be not helpful |