JVM Performance Optimisation Training Summary

# Performance KPI (Key Performance Index)

1 – Throughput

* Percentage of work-related tasks done. E.g. in 24 hours, a system spends a total of 5 minutes for GC. ∴throughput = (100 - ( 5 / ( 24 x 60)) x 100)% = 99.652%.

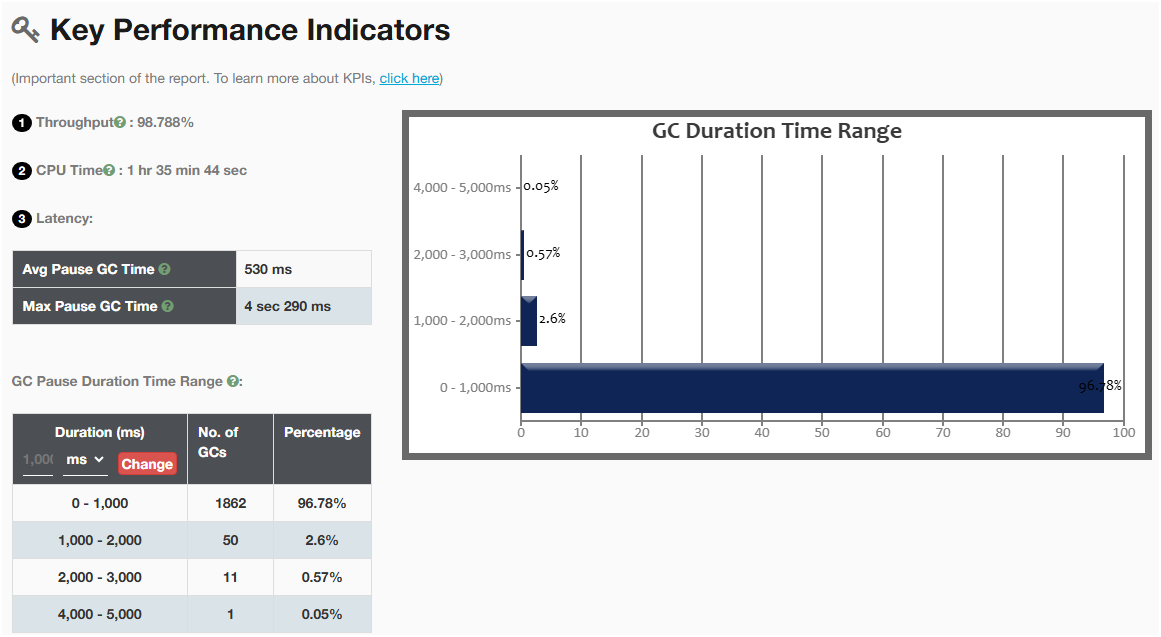
2 – Latency

* Amount of time taken for GC: maximum, average,distribution.

3 – Footprint

* Amount of CPU time taken for GC.

[GCEasy](https://gceasy.io/) can be used to display the information based on GC log as shown below. See [GC log section](#_GC_log:) for more details on GC log).



# Performance Problems

## CPU Spike

### Why

### How to Solve

## OutOfMemoryError

### Why

### How to Solve

## StackOverflowError

### Why

### How to Solve

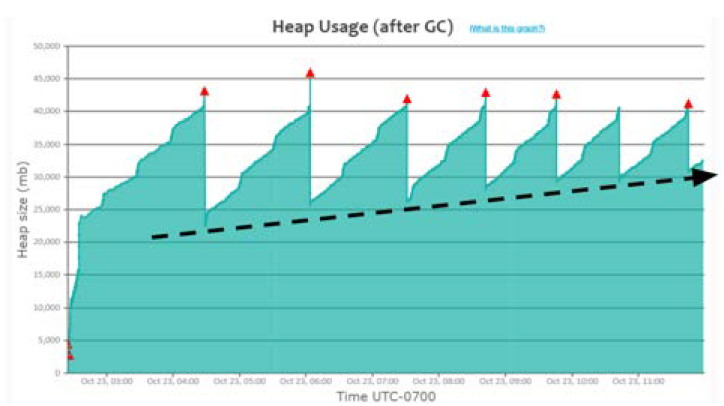
# GC Patterns

## Healthy GC (sawtooth)

A graph showing the amount of air in the air

Description automatically generated with medium confidence

## Acute Memory Leak (uptrend)

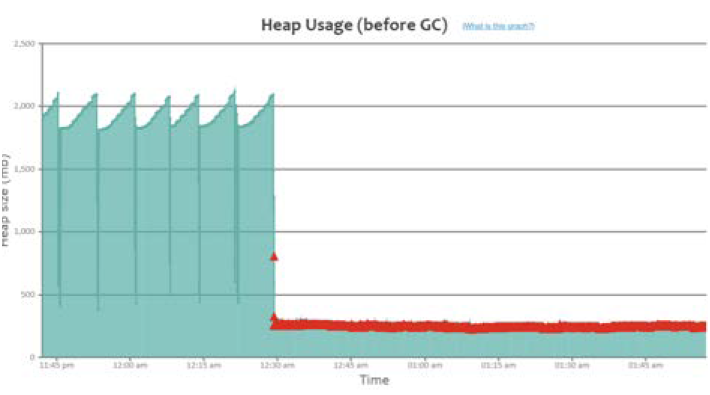


## Heavy Caching

A graph showing a number of blue dots

Description automatically generated with medium confidence

## Metaspace memory problem



# Useful JVM Arguments for Optimisation

## Heap

Use one of the followings:

|  |  |
| --- | --- |
| **JVM Arguments** | **Remarks** |
| -Xmx |  |
| -XMaxRAMFraction |  |
| -XMaxRAMPercentage |  |

## Metaspace

|  |  |
| --- | --- |
| **JVM Arguments** | **Remarks** |
| -Xx:MaxMetaspaceSize |  |

## Stack: -Xss

|  |  |
| --- | --- |
| **JVM Arguments** | **Remarks** |
| -Xss |  |

## GC selection

Use one of the followings:

|  |  |
| --- | --- |
| **JVM Arguments** | **Remarks** |
| -XX:+UseSerialGC |  |
| -XX:+UseParallelGC |  |
| -XX:+UseConcMarkSweepGC |  |
| -XX:+ UseG1GC |  |
| -XX:+ UseShenandoahGC |  |
| -XX:+ UseZGC |  |

## Timeouts

Use any of the followings if needed:

|  |  |
| --- | --- |
| **JVM Arguments** | **Remarks** |
| ‑Dsun.net.client.defaultConnectTimeout | Timeout to connect to host.  Example:  ‑Dsun.net.client.defaultConnectTimeout=2000 |
| -Dsun.net.client.defaultReadTimeout | Timeout when reading from input stream.  Example:  -Dsun.net.client.defaultReadTimeout=2000 |

# Useful JVM Arguments for Troubleshooting

## GC log:

* Analyse GC log for period of 24 hours during weekdays for high and low traffic monitoring.
* Can be used to troubleshoot GC-related problems: long GC pauses, irresponsive application, low throughput, memory leak indication GC pattern

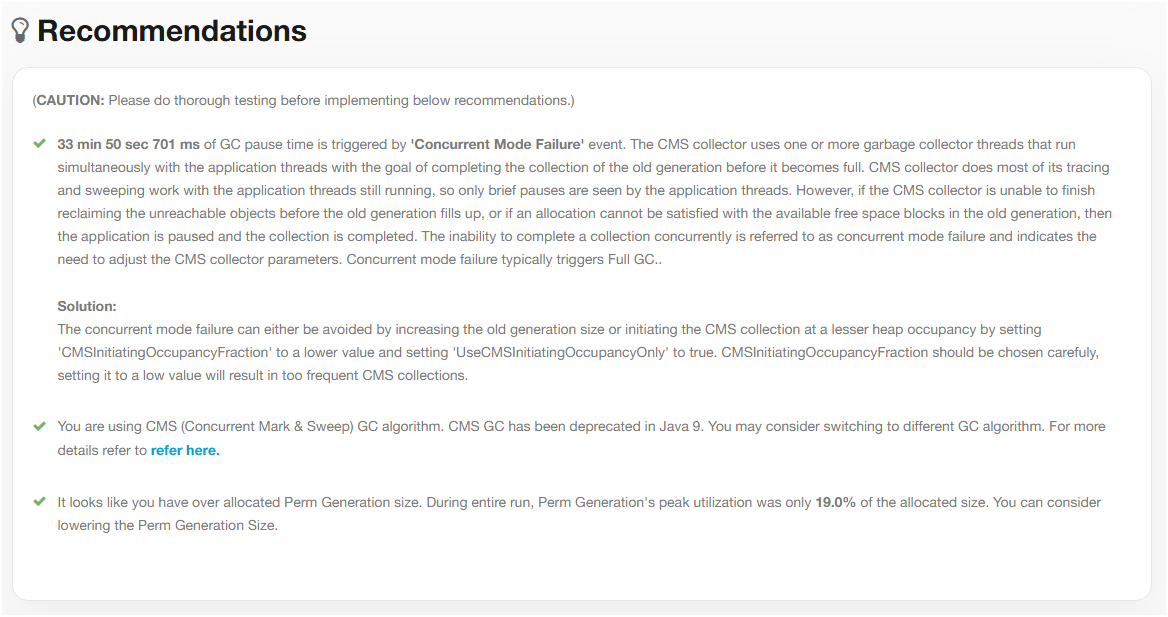
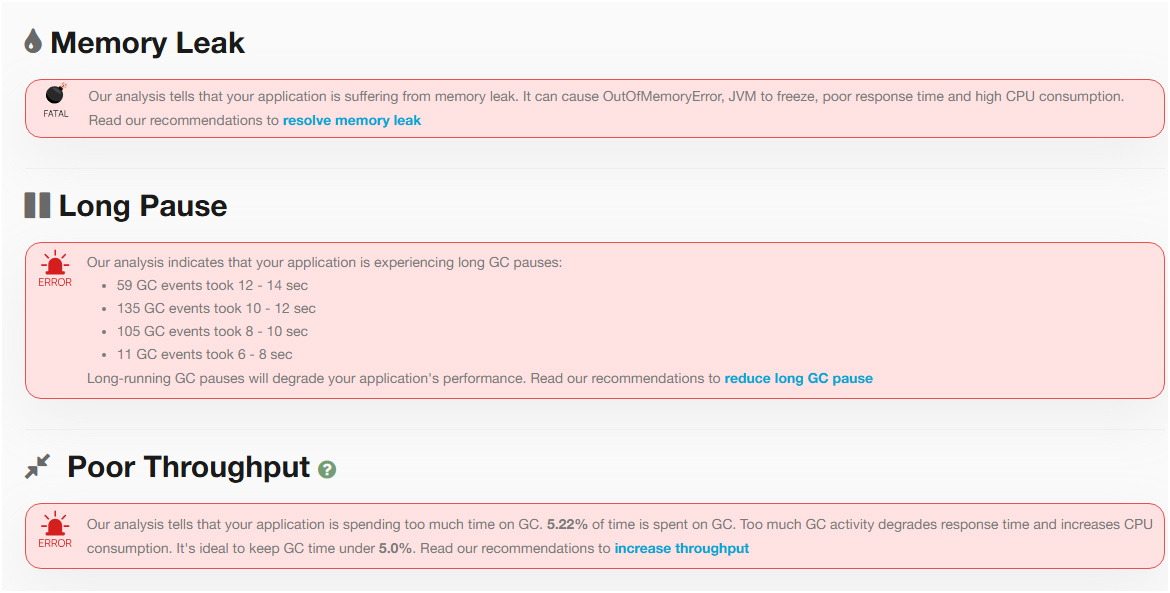
|  |  |
| --- | --- |
| **JVM Arguments** | **Remarks** |
| ‑XX:+PrintGCDetails  ‑XX:+PrintGCDateStamps  -Xloggc:<file-path>  -verbose:gc  -Xloggc:<log\_file\_path>  -XX:+PrintGCDetails  -XX:+PrintGCDateStamps | For Java 7 and below  Example:  *java -verbose:gc*  *-Xloggc:/var/log/myapp/gc.log*  *-XX:+PrintGCDetails*  *-XX:+PrintGCDateStamps -jar myapp.jar* |
| -XX:+PrintGC  -XX:+PrintGCDetails  -XX:+PrintGCDateStamps  -Xloggc:<log\_file\_path>  -XX:+UseGCLogFileRotation  -XX:NumberOfGCLogFiles=<number\_of\_files>  -XX:GCLogFileSize=<size>[k|m|g] | For Java 8: it has additional log rotation option.  Example:  *java -XX:+PrintGC –*  *XX:+PrintGCDetails*  *-XX:+PrintGCDateStamps*  *-Xloggc:/var/log/myapp/gc.log*  *-XX:+UseGCLogFileRotation*  *-XX:NumberOfGCLogFiles=5*  *-XX:GCLogFileSize=10m -jar myapp.jar* |
| -Xlog:gc\*:  file=<log\_file\_path>:  time,uptime,level,tags:  filecount=<number\_of\_files>,  filesize=<size>[k|m|g] | For Java 9 and above: it is using unified logging -Xlog  Example:  *java -Xlog:gc\*:*  *file=/var/log/myapp/gc.log:*  *time,uptime,level,tags:*  *filecount=5,*  *filesize=10m -jar myapp.jar* |

### Tools

To analyse GC log, we can use:

* [GCEasy](https://gceasy.io/) by yCrash
* JDK Mission Control
* JVisualVM
* [IBM Health Centre and/or IBM GC and Memory Visualizer](https://www.ibm.com/support/pages/java-sdk-monitoring-and-post-mortem)
* [Garbage Cat](https://github.com/mgm3746/garbagecat)

With [GCEasy](https://gceasy.io/), summary and recommendations are provided like shown below:



## Heap Dump

* Can be used to troubleshoot memory-related problems: slow memory leaks, GC problems, OutOfMemoryError

|  |  |
| --- | --- |
| **JVM Arguments** | **Remarks** |
| -XX:+HeapDumpOnOutOfMemoryError  -XX:HeapDumpPath=<file\_path> | Example:  XX:+HeapDumpOnOutOfMemoryError  -XX:HeapDumpPath=/opt/tmp/heapdump.bin |

### Tools

1 - [HeapHero](https://heaphero.io/)

2 – jcmd.exe: available with JDK

* $ jcmd <pid> GC.heap\_dump <file\_path>. E.g.: $ jcmd 37320 GC.heap\_dump /opt/tmp/heapdump.bin

3 – JVisualVM

## Thread Dump

* It’s a snapshot of all the threads running in a Java process.
* Can be used to troubleshoot: CPU spikes, unresponsiveness, poor response time, hung threads, high memory consumption.
* No noticeable overhead in capturing thread dumps on every 5 minutes or 2 minutes interval.

|  |  |
| --- | --- |
| **JVM Arguments** | **Remarks** |
|  |  |

### Tools

1 – jstack.exe: available with JDK

* $ jstack -l <pid> > <file\_path>. E.g.: jstack -l 37320 > /opt/tmp/threadDump.txt

2 – kill-3 <pid>: available with JRE

* $ kill -3 <pid>. E.g.: $ kill -3 37320

3 – jcmd.exe: available with JDK

* $ jcmd <pid> Thread.print > <file\_path>. E.g.: $ jcmd 37320 Thread.print > /opt/tmp/threadDump.txt

4 - JVisualVM

# Useful JVM Arguments as Reactive Actions

## When OutOfMemoryError

Use any of the followings:

|  |  |
| --- | --- |
| **JVM Arguments** | **Remarks** |
| -XX:+HeapDumpOnOutOfMemoryError  -XX:HeapDumpPath={heap-dump-file-path} |  |
| -XX:OnOutOfMemoryError=<script\_path> | Execute a script when OOME occurs.  Example:  -XX:OnOutOfMemoryError=/scripts/restart-myapp.sh |
| -XX:+CrashOnOutOfMemoryError |  |
| -XX:+ExitOnOutOfMemoryError |  |

# Recommended Practice for Optimum Performance

## Set Max Heap Size and Metaspace Size accordingly

Heap size and Metaspace size plays a role in determining the frequency of GC events for your application.

### Setup

|  |  |
| --- | --- |
| **JVM Arguments** | **Remarks** |
| -Xmx | Example:  Setting heap size to 2GB:  -Xmx2g |
| -XX:MaxMetaspaceSize | Example:  Setting Metaspace size to 256MB:  -XX:MaxMetaspaceSize=256m |

## Use ZGC for Java 11+ Application

ZGC is known for its sub-millisecond pauses, allowing latency-sensitive systems to thrive.

### Setup

|  |  |
| --- | --- |
| **JVM Arguments** | **Remarks** |
| -XX:+UseZGC |  |

## Always Enable GC Logging

GC Logging has (close to) no-impact to application performance but very useful for troubleshooting purposes. See [GC log section](#_GC_log:) for more details.

### Setup

|  |  |
| --- | --- |
| **JVM Arguments** | **Remarks** |
| -XX:+PrintGCDetails  -XX:+PrintGCDateStamps  -Xloggc: <file\_path> | Up to JDK 8  Example:  -XX:+PrintGCDetails  -XX:+PrintGCDateStamps  -Xloggc:/opt/workspace/myAppgc.log |
| -Xlog:gc\*:file=<file\_path> | JDK 8 +  Example:  -Xlog:gc\*:file=/opt/workspace/myAppgc.log |

## Create Heap Dump on Out of Memory Error

Heap dump is very useful in troubleshooting OutOfMemoryError in application when it happens.

### Setup

|  |  |
| --- | --- |
| **JVM Arguments** | **Remarks** |
| -XX:+HeapDumpOnOutOfMemoryError  -XX:HeapDumpPath=<file\_path> | Example:  -XX:+HeapDumpOnOutOfMemoryError  -XX:HeapDumpPath=/dmp/my-heap-dump.hprof |

## Increase Stack Memory Only When Needed

Each thread will have its own stack. When not enough memory in stack, StackOverflowError is thrown.

### Setup

|  |  |
| --- | --- |
| **JVM Arguments** | **Remarks** |
| -Xss | Example:  Setting stack size to 256KB:  -Xss256k |

## Set timeout for connection

This is to avoid unresponsiveness in your application caused by remote applications and safeguard your applications high availability.

### Setup

|  |  |
| --- | --- |
| **JVM Arguments** | **Remarks** |
| -Dsun.net.client.defaultConnectTimeout  -Dsun.net.client.defaultReadTimeout | Example:  -Dsun.net.client.defaultConnectTimeout=2000  -Dsun.net.client.defaultReadTimeout=2000 |

## Set Time Zone for Your Application

This is particularly useful for sensitive business requirements in an application running in a distributed environment.

### Setup

|  |  |
| --- | --- |
| **JVM Arguments** | **Remarks** |
| -Duser.timezone | Example:  -Duser.timezone=”Asia/Kolkata” |

## Stop Wasting Memory in Your Code

1. Lazy Initialisation of your collection
2. Apply Goldilocks principle to your collection size
3. Stop String duplication

With G1GC (Java 8+), use XX:+UseStringDeduplication. With others, use ‘intern()’. However, beware of the performance impact with the latter. Thus, apply this on troublesome code that has been pointed out by your monitoring tool as a source of String duplication.

# Tools