Performance in HotSpot JVM

for latency-sensitive applications

Thomas Pliakas @ JHUG

Agenda

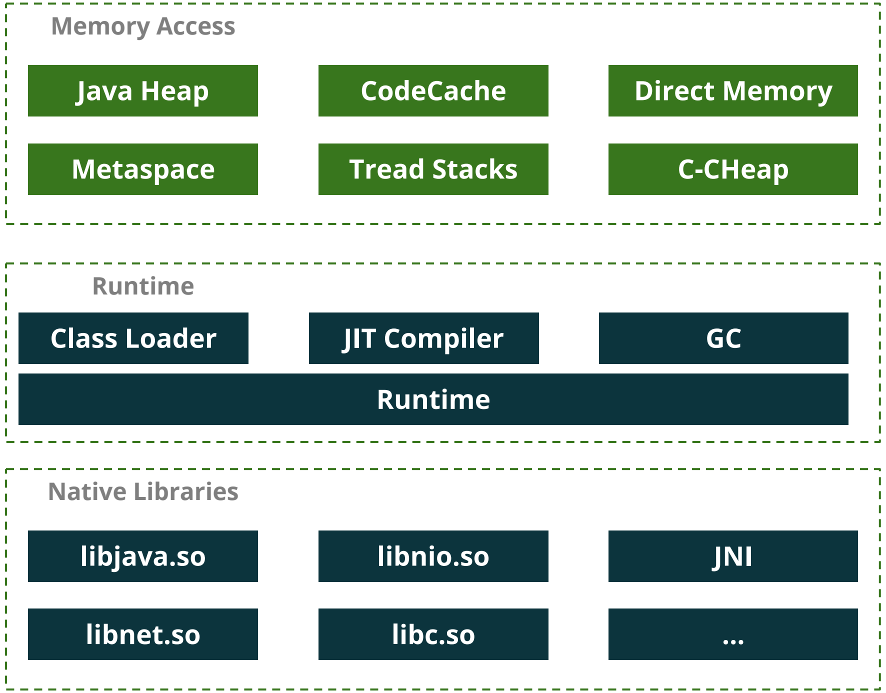
- Introduction
- Memory
- Classloader
- Just-In-Time Compiler
- Threads
- Garbage Collectors (G1C, ZGC, Shenandoah GC)
- Containers

Introduction

The current VM options are based on AdoptOpenJDK 64-Bit Server VM can be found:

```
java -XX:+UnlockDiagnosticVMOptions  \
     -XX:+UnlockExperimentalVMOptions \
     -XX:+PrintFlagsFinal -version
```

JVM Hotspot Architecture



Memory :: General

1. **`-XX:+UseCompressedOops (default = true)`**
   It allows references to be 32-bit in a 64-bit JVM for heap sizes less than 32 GB, typically increasing the performance. (Not supported by ZGC).

2. **`-XX:+UseComporessedClassPointers (default = true)`**
   It enables 32-bit offset to represent the class pointer in a 64-bit JVM for heaps < 32Gb

3. **`-XX:+UseLargePages (default = false)`**
   Enable to use the large page memory. Translation-Lookaside Buffers (TLB) optimization.

4. **`-XX:+UseNUMA (default = true)`**
   Enables the NUMA aware Collector to allocate objects in memory node local to a processor, increasing the application's use of lower latency memory.

5. **`LD_PRELOAD (default = malloc)`**
   Changing the default malloc allocation to avoid possible native memory fragmentation. Possible values are: `jemalloc`, `tcmalloc`, `mimalloc`.

Memory :: Heap

1. **`Set -Xms to be equals to -Xmx`**

   Setting the initial heap to be equals to max heap size, you are avoiding resizing.

2. **`-XX:+AlwaysPretouch`**

   Trigger pre-zeroed memory-mapped pages at startup, during JVM initialization.

Memory :: Metaspace

1. `-XX:MetaspaceSize (default = 21,807,104)`
   Set the size of the allocated class metadata space that will trigger a GC when it is exceeded.

2. `-XX:InitialBootClassLoaderMetaspaceSize (default = 4,194,304)`
   Consider a higher value to increase the boot class loader Metaspace.

3. `-XX:MaxMetaspaceExpassion (default = 5,541,776)`
   Represents the maximum size to expand a Metaspace without triggering Full GC.

4. `-XX:MinMetaspaceFreeRatio (default = 40)`

5. `-XX:MaxMetaspaceFreeRatio (default = 70)`

Memory :: CodeCache

1. **`-XX:InitialCodeCacheSize (default = 2,555,904)`**
   The initial code cache size.

2. **`-XX:ReservedCodeCacheSize (default = 251,658,304)`**
   The reserved code cache size (can be considered as the max code cache size)

3. **`-XX:+UseCodeCacheFlushing (default = true)`**
   Attempt to sweep the CodeCache before shutting down the compiler. Be sure that is always true.

4. **`-XX:+SegmentedCodeCache (default = true)`**
   Divide the CodeCache into distinct segments (e.g. non-method, profiled, and non-profiled code) to improve the code locality (i.e. better iTLB and iCache behavior), to decrease fragmentation of highly-optimized code and to better control JVM memory footprint. Please make sure this option is enabled.

5. **`-XX:-TieredCompilation (default = true)`**
   It disables the intermediate compilation tiers (Tier 1, Tier 2, and Tier 3) so that a method is either Interpreted or compiled at the maximum optimization level by C2 JIT (basically it uses only Tier 0, and Tier 4). Note: disabling TieredCompilation will (i) minimize the number of Compiler threads, (ii) simplify the compilation policy (i.e. based on method invocation and backedge counters but without detailed profiled information), and (iii) reduce the amount of JIT-compiled code, hence minimizing CodeCache usage.

Memory :: DirectBuffers

1. `-XX:MaxDirectMemorySize (default = 0)`
   Setting the initial heap to be equals to max heap size, you are avoiding resizing.

2. `-Djdk.nio.maxCachedBufferSize`
   Consider limiting ***maxCachedBufferSize*** *to avoid these kinds of problems.*

Classloader :: Dynamic Class-Data Sharing

1. **`-XX:ArchiveClassesAtExit (default = dynamic_archive_file.jsa)`**
   Dynamically creates the application shared archive when the application exits.

2. **`-XX:SharedArchiveFile (default = dynamic_archive_file.jsa)`**
   Specifies the name of the dynamic archive file.

JIT :: General

1. `-Xbatch`
   Enabling this will switch from a background to a foreground compilation process across JIT threads, resulting to a better deterministic JIT behaviour.

2. `-Xverify:none`
   It disables bytecode verification resulting to faster JVM startup.

3. `-XX:TierStopAtLevel=1`
   Stops the compilation at C1 and limits optimization of JIT compiler. Both C2 JIT or GRAAL will not be triggered.

JIT :: C1 + C2 Compiler

1. `-XX:+TieredCompilation (default = true)`
   Enable tiered compilation.
2. `-XX:InlineSmallCode (default = 2,000)`
   Inline a previously compiled method only if its generated native code size is less than ***InlineSmallCode****.*
3. `-XX:+MaxInlineSize (default = 35)`
   Maximum size of method bytecode which gets inlined if reaching -XX:MinInliningThreshold.
4. `-XX:+MaxInlineLevel (default = 9)`
   The maximum number of nested calls that gets inlined.
5. `-XX:FreqInlineSize (default = 325)`
   Maximum bytecode size of a frequently executed method to be inlined.
6. `-XC:MinInliningThreshold (default = 250)`
   The minimum number of invocations for a method to be inlined.

JIT :: Graal Compiler

1. **`-XX:+UnlockExperimentalVMOptions -XX:+UseJVMCICompiler (experimental, default false)`**
   To enable the Graal JIT Compiler (e.g. Interpreter -> C1 JIT -> Graal JIT).

Threads :: Stack

1. **`-XX:+OmitStackTraceInFastThrow (default true)`**
   For performance reasons, consider throwing pre-allocated exceptions that do not provide a stack trace.

2. **`-XX:+StackTraceInThrowable (default true)`**
   For performance reasons, consider removing stack traces from thrown exceptions.

Garbage Collectors :: General

1. `-XX:+ExplicitGCInvokesConcurrent (default false)`
   Avoid a lengthy pause in response to a System.gc() or Runtime.getRuntime().gc() by enabling concurrent GCs..

2. `-XX:-UseGCOverheadLimit (default true)`
   It is designed to prevent applications from running for an extended period of time while making little or no progress because the heap is too small. .

3. `-XX:-DisableExplicitGC (default false)`
   It might have the hidden side effect of not reclaiming the unused off-heap memory used by direct ByteBuffers. For example, in case of direct ByteBuffer allocations fails, under the hood, the JDK code explicitly calls System.gc() which suppose to reclaim the unused memory. Disabling the explicit GC will disable this mechanism.

Garbage Collectors :: G1C :: Latency

1. **`-XX:-UseTransparentHugePages (default false)`**
   Consider keeping Transparent Huge Pages (THP) disabled unless there is a proven benefit..

2. **`set -Xms equals to -Xmx`**
   To minimize heap resizing work by disabling it.

3. **-XX:+AlwaysPreTouch**
   Pre-touch and set to zero all virtual memory pages during VM startup time.

4. **`-XX:+UseNUMA (default = true)`**
   Enables the NUMA aware Collector to allocate objects in memory node local to a processor, increasing the application's use of lower latency memory.

5. **`-XX:+ParallelRefProcEnabled (default true)`**
   If the time taken to process reference objects is high or increasing (i.e. ref-proc and ref-enq is the major contributor) enable parallelization of these phases.

6. **`-XX:+ParallelRefProcEnabled (default true)`**
   If the Evacuate Collection Set phase (i.e. Object Copy sub-phase) during a Young GC takes too long, consider decreasing the G1NewSizePercent (i.e. the percentage of the heap to use as the minimum for the Young Generation size)..

Garbage Collectors :: G1C :: Latency

1. **`-XX:G1NewSizePercent (default 5)`**
   Consider keeping Transparent Huge Pages (THP) disabled unless there is a proven benefit.

2. **`-XX:G1MaxNewSizePercent (default 60)`**
   This limits the maximum size of the Young Generation and so the number of objects that need to be processed during the pause. If the amount of objects surviving a Collection suddenly changes it might cause spikes in the GC pause time. Consider decreasing G1MaxNewSizePercent (i.e. the percentage of the heap size to use as the maximum for Young Generation size).

3. **`-XX:G1HeapRegionSize=n (default 2,097,152)`**
   It directly affects the number of cross-region references and as well as the size of the remembered set. Handling the remembered sets for regions may be a significant part of Garbage Collection work, so this has a direct effect on the achievable maximum pause time. Larger regions tend to have fewer cross-region references, so the relative amount of work spent in processing them decreases, although, at the same time, larger regions may mean more live objects to evacuate per region, increasing the time for other phases.

Garbage Collectors :: G1C :: Throughput

1. `set -Xms equals to -Xmx`
   To minimize heap resizing work by disabling it.

2. `-XX:+AlwaysPreTouch`
   Pre-touch and set to zero all virtual memory pages during VM startup time.

3. `-XX:+UseLargePages`
   Enabling this feature will also improve performance.

4. `-XX:G1NewSizePercent (experimental, default 5)`

5. `-XX:G1MaxNewSizePercent (experimental, default 60)`

6. `-XX:GCPauseIntervalMillis (default = 201)`
   You can specify the length of the time period during which the pause can occur.

7. `-XX:MaxGCPauseMillis=n (default 200)`
   Attempt to keep Garbage Collection induced pauses shorter than n milliseconds. The generation sizing heuristics will automatically adapt the size of the Young Generation, which directly determines the frequency of pauses. Hence, increasing the maximum pause time will potentially decrease the frequency of the pauses, improving the throughput.

8. `-XX:G1RSetUpdatingPauseTimePercent (default 10)`
   *Try to decrease the amount of concurrent work, in particular, concurrent remembered set updates, which requires a lot of CPU resources. By decreasing **G1RSetUpdatingPauseTimePercent** it will move the work from the concurrent operation into the Garbage Collection pause, potentially increasing the throughput.*

Garbage Collectors :: Z GC

1. **`-XX:+UseZGC (experimental, default false)`**
   Enable Z GC..

2. **`-Xmx`**
   Setting an appropriate max heap size is the most important tuning option for ZGC. Since ZGC is a concurrent Collector, a max heap size must be selected such that (i) the heap can accommodate the live-set of your application and (ii) there is enough headroom in the heap to allow allocations to be serviced while the GC is running. In general, the more memory you give to ZGC the better..

3. **`-XX:+ConcGCThreads`**
   The number of concurrent GC threads is automatically selected, nevertheless, depending on the characteristics of the application this might need to be adjusted.

4. **`-XX:+UseNUMA`**
   GC has basic NUMA support, which means it will try it's best to direct Java heap allocations to NUMA-local memory.

5. **`-XX:-XX:+UseTransparentHugePages`**
   An alternative to using explicit large pages (as described above) is to use transparent huge pages. The use of transparent huge pages is usually not recommended for latency-sensitive applications because it tends to cause unwanted latency spikes. However, it might be worth experimenting with to see if/how the workload is affected by it.

6. **`-XX:-XX:+UseLargePages`**
   Use large pages will generally yield better performance (in terms of throughput, latency and startup).

Garbage Collectors :: Shenadoah GC

1. `-XX:+UseShenandoahGC (default false)`
   Enable Shenandoah GC.

2. **`Set -Xms equals to -Xmx`**
   Minimize heap resizing work.

3. **-XX:+Always Pretouch**
   The number of concurrent GC threads is automatically selected, nevertheless, depending on the characteristics of the application this might need to be adjusted.

4. **`-XX:+UseNUMA`**
   *While Shenandoah does not support NUMA explicitly, it is a good idea to enable this to also enable NUMA interleaving (**-XX:UseNUMAInterleaving** (default false)) on multi-socket hosts. When coupled with **-XX:+AlwaysPreTouch**, it provides better performance than the default out-of-the-box configuration*

5. `-XX:+UseTransparentHugePages`
   *Will enable the large pages transparently. It is recommended to set /sys/kernel/mm/transparent_hugepage/enabled and /sys/kernel/mm/transparent_hugepage/defrag to "madvise". When coupled with -**XX:+AlwaysPreTouch**, then init/shutdown would be faster, because it will pre-touch with larger pages. It will also pay the defrag costs upfront, at startup.*

6. `-XX:+UseLargePages`
   *Using large pages greatly improves performance on large heaps. This would enable hugetlbfs (Linux) or Windows (with appropriate privileges) support.When coupled with -**XX:+AlwaysPreTouch**, then init/shutdown would be faster, because it will pre-touch with larger pages. It will also pay the defrag costs upfront, at startup..*

Containers :: General

1. `-XX:+UseContainerSupport (default true)`
   Make sure container support is enabled. It allows the JVM to read cgroup limits like available CPUs and RAM.

2. `-XX:+PreferContainerQuotaForCPUCount (default true)`
   *If the flag **PreferContainerQuotaForCPUCount** is set to true, use the **cpu_quota** instead of **cpu_shares** for picking the number of cores, without exceeding the number of physical CPUs in the system. The JVM will use this count to make decisions such as how many compiler threads, GC threads, and sizing of the fork-join pool. if the flag **PreferContainerQuotaForCPUCount** is false, use the minimum of cpu_shares or cpu_quotas, if set, without exceeding the number of physical CPUs in the system. If only one of cpu_shares or cpu_quotas is provided, then use the specified value limited by the number of physical processors in the system.*

3. `-XX:InitialRAMPercentage (default 1.5)`
   Consider increasing the number of minimum heap size as a percentage of the available container memory.

4. `-XX:MaxRAMPercentage (default 25)`
   Consider increasing the number of maximum heap size as a percentage of available container memory..

JVM Optimization :: Resources

1. Book :: Java Performance, in depth advice of tuning and programming Java 8, 11 and Beyond
2. Book :: Java :: Optimizing Java, practical techniques for improving JVM application performance, Ben Evans, James Gough, Chris Newland
3. Web :: HotSpot Virtual Machine Garbage Collection Tuning Guide
4. Web :: OpenJDK Wiki – Shenandoah Garbage Collector
5. Web :: OpenJDK Wiki – Z Garbage Collector
6. Web :: OpenJDK Wiki – Server Compiler Inlining Messages
7. Blog :: Oracle Blogs – Never disable bytecode verification in a production system
8. Blog :: Using jemalloc to get to the bottom of a memory leak
9. Resource :: VM Options Explorer
10. *Blog : Hotspot JVM performance tuning guidelines.*

QUESTIONS ?