Chapter 1: Introduction

Java Performance Turing is Part Art and Part Science.

Java Performance depends upon multiple factors both internal and external factor influence the performance of the code.

- The JDK version
- How well/efficient the code is written
- How many lines of code that needs to be complied
- Even in the minor version change can significantly impact the performance of the code (both in positive and in negative ways).

"**Premature optimization**"

Chapter 2: A Java performance toolbox

- **Microbenchmark:**
  - Microbenchmarking should use the results – else JVM will discard the calculation and smart JVM will skip the calculation.
  - Microbenchmarking should not be including extraneous (irreverent /unrelated) operation
  - Microbenchmarking with correct inputs – the inputs that mimics real work data.
- **Macrobenchmark:** The best thing to use to measure performance of an application is the application itself, in conjunction with any external resources it uses.

Mesobenchmarks are not limited to the Java EE arena, it is a term I use for benchmarks that do some real work, but are not full-fledged applications.

System under test mostly consider software system, but its also important to note that the hardware component also places a significant impact on the system-under-test.

Elapsed Time (Batch) Measurements – most of the system (application servers) usually optimized itself over first few iteration thus it's important to measure the Elapsed Time (Batch) Measurements – in case of JPA it cache the initial results which improves the subsequent results.

Throughput Measurement: Usually, this number is reported as the number of operations per second, rather than the total number of operations over the measurement period. This measurement is frequently referred to as transactions per second (TPS), requests per second (RPS), or operations per second (OPS). The throughput measure is done after a suitable warm-up period , as during the warm-up the system would have varying throughput.

Response Time: The response time is closer to user action, which includes 'Think Time', because of which will have varying response. Thus, response time is measured in percentile. 90% of response is less than 1 sec. Also, its better to consider average response time, as it would cater a larger user base.

Early and frequent regression testing is advisable.

- Automate Everything – performance testing, baselining .
- Measure Everything - must gather every conceivable piece of data that will be useful for later analysis.
- Run on Target system – different system behaves differently. Large machine will be threading effect.

The focus on the performance monitoring toolbox , specially the tools that are ready available with JDKs, and tools that are available with Operating systems – Windows / Unix distributions.

CPU Usages

1. CPU usages is denoted by two separate parameters

1) User Time (time when the OS is running application code)

2) System Time (time the OS is running Kernel code)

2. CPU Usages number – how effectively the application code is using CPU, higher the number better its.

3. If the Application is wating for resource, then CPU will spend cycle executing the application code. It's always advisable to leave some ideal CPU time for the next upcoming tasks.

Java and single CPU Usages – Most application would work on request based, when there is an incoming load, it processes the workload and CPU usages skips up momentarily, and the rest of the time it remains ideal. Optimizing the application to take less CPU resource, would show some improvement in overall (average) CPU usages. But when the application is receiving constant load, then same improvement cannot be seen.

Java and multiple CPU Usages – The goal of the Multiple CPU with multiple thread is drive the CPU higher by making sure individual threads are non-blocking. Each thread is design to perform a single task at a time, if the thread is block as its waiting for the resource, no new task can be picked even if there are tasks to be executed – resulting in idea CPU time. IN this case, the thread-pool size should be increased.

CPU threads – In windows, the processor queue length does not include the threads that are currently running; in Unix it also includes the threads that are currently running. Hence for windows system idea processor queue length is Zero, while in Unix its desired to be less than equal to number of processors.

Key take aways

1.CPU time is the first thing to examine when looking at performance of an application.
2. The goal in optimizing code is to drive the CPU usage up (for a shorter period of time), not down.
3. Understand why CPU usage is low before diving in and attempting to tune an application.

Disk I/O another bottleneck.

Java Monitoring Tool

jcmd	Prints basic class, thread, and VM information for a Java process.
jconsole	Provides a graphical view of JVM activities, including thread usage, class usage, and GC activities.
jhat	Reads and helps analyze memory heap dumps. This is a postprocessing utility.
jmap	Provides heap dumps and other information about JVM memory usage. Suitable for scripting, though the heap dumps must be used in a postprocessing tool.
jinfo	Provides visibility into the system properties of the JVM, and allows some system properties to be set dynamically. Suitable for scripting.
jstack	Dumps the stacks of a Java process
jstat	Provides information about GC and class-loading activities
jvisualvm	A GUI tool to monitor a JVM, profile a running application, and analyze JVM heap dumps (which is a postprocessing activity, though jvisualvm can also take the heap dump from a live program).

Keey monitoring area for Java

- Basic VM information
- Thread information
- Class information
- Live GC analysis
- Heap dump postprocessing
- Profiling a JVM

The jFlags are not supposed to be change , without a significant reason. jinfo can change few flags at the runtime, but it not necessary the change value is picked by JVM as many of the flags are loaded during startup.

Default flag values can be found by including -XX:+PrintFlags Final on a command line. This is useful for determining the default ergonomic settings of flags on a particular platform. Often the defeault value of the flags are denoted by : at the start of the flag.

Thread Information

jconsole and jvisualvm display information (in real time) about the number of threads running in an application. It can be very useful to look at the stack of running threads to determine if they are blocked.

Heap Dump

The heap dump is a snapshot of the heap that can be analyzed.

Java Profiling Tool

Almost all Java profiling tools are themselves written in Java and work by “attaching” themselves to the application to be profiled–meaning that the profiler opens a socket (or other communication channel) to the target application. The target application and the profiling tool then exchange information about the behavior of the target application.

There are two profiling methods – Sampling Method and instrumented method.

Sampling Profile method is common, as it has a relatively low profile – Different sampling profile behaves differently; each may be better for a particular application.

Instrumented method can be bit more *intrusive*, as it altered the bytecode sequence of classes as they are loaded (inserting code to count the invocations, and so on).

Threads

Threads that are blocked may or may not be a source of a performance issue; it is necessary to examine why they are blocked. Blocked threads can be identified by the method that is blocking, or by a timeline analysis of the thread.

Native Profilers

Native profiling tools are those that profile the JVM itself. This allows visibility into what the JVM is doing, and if an application includes its own native libraries, it allows visibility into that code as well. Any native profiling tool can be used to profile the C code of the JVM (and of any native libraries), but some native-based tools can profile both the Java and C/C++ code of an application.

Java Mission Control

It available with commercially available Oracle Java versions, Java mission control not only monitors what is happing with the JVM but also, provide an insight on what is happing at the OS level.

Java Flight Recorder

Java Mission Control is the Java Flight Recorder (JFR). The main function of a JFR is a history of events in the JVM that can be used to diagnose the past performance and operations of the JVM. The data stream is held in a circular buffer, so only the most recent events are available. Java Mission Control can then display those events–either taken from a live JVM or read from a saved file. By default, JFR is set up so that it has very low overhead: an impact below 1% of the program’s performance. That overhead will change as more events are enabled, or as the threshold at which events are reported is changed, and so on.

JFR Memory view:

The data capture by JFR Memory view is extensive, it let us understand CMS bailouts , performance of the full GC. How JVM adjust to tenuring threshold, virtual any piece of data how GC behaved can be answered by the data captured by JFR capture data.

JFR Code view:

The Code page in Java Mission Control shows basic profiling information from the recoding. Unlike other profiles, JFR offers other node of visibility into code. The throwable tab provides a view into the exception processing of the application. There are also tab that provide the information on the complier doing, include a peek into the code cache. Thread I/O, and system events provides a more insights on the JFR recording.

JFR Events :

In general there are many events that is been generated by the JVM , depending upon the configuration of the JFR these events can be monitored , not all events are enabled from the starting.

Java 7u40, there are 77 event types that can be monitored with JFR .

Classloading

- Number of classes loaded and unloaded.
- Which classloader load class, time required to load an individual class.

Thread statistics

- Number of threads created and destroyed , thread dumps.
- Which threads are block on locks, (and the specific locks they are blocked on).

Throwables

- Throwable classes used by the application.
- How many exceptions and error are thrown and the stack trace of their creation.

TLAB allocations

- The number of allocations in the heap and size of thread-local allocation buffers (TLABs)
- The number of allocations in the heap and the stack trace where they are allocated.

File and socket I/O

- Time spent performing I/O.
- Time spent read/write call, the specific file or socket taking a long time to read or write.

Monitoring blocked

- Thread waiting for a monitor.
- Specific thread blocked on specific monitors and the length of time they blocked.

Code Cache

- Size of code cache and how much it contains.
- Methods removed from the code cache; code cache configuration.

Code Compilation

- Which methods are complied, OSR compilation, and length of time to compile.

Garbage Collection

- Times for GC, including individual phases; size of generations

Profiling

- Instrumenting and sampling profiles
- Each profiler have something different to share, JFR profile provides a good high-order overview.

Enabling JFR

The JFR is disable by default, it needs to be enabled, to enable it in commercially enabled Oracle JVM , need to add a flag -XX:+UnlockCommercialFeatures -XX:+FlightRecorder to the command line of the application. There are different ways to enable JFRs through Java mission control GUI Or through jcmd command line.

JFR event

There are 77 events that can be enabled in JVM. Collecting event involved some overhead. By default, expensive events are not enabled. The events and the threshold of the event that JFR capture are defined in two template – default template and profile template. Where default template has an overhead of 1% , enabling profile template has overhead of 2%. One can also define their own custom template, it is an xml file where one need to define the jfr event that need to capture and its threshold.

Chapter 3: Working with the JIT Compiler

Just-in-Time Compiler

Interpreter: Line by line conversion of the code into Binary code, which can be understand by the complier and can run it.

Complier: Converts a chunk of code into Binary code, which can be understand by the complier and can run it. The disadvantage of the compilers is its portability, code written for one CPU type cannot be used for other CPU types.

Java attempts to find a middle ground here. Java applications are compiled—but instead of being compiled into a specific binary for a specific CPU, they are compiled into an idealized assembly language. This assembly language (know as Java bytecodes) is then run by the java binary (in the same way that an interpreted PHP script is run by the. This gives Java the platform independence of an interpreted language. Because it is executing an idealized binary code, the java program is able to compile the code into the platform binary as the code executes. This compilation occurs as the program is executed: it happens “just in time.”

The manner in which the Java Virtual Machine compiles this code as it executes is the focus of this chapter.

In a typical program, only a small subset of code is executed frequently, and the performance of an application depends primarily on how fast those sections of code are executed. These critical sections are known as the hot spots of the application; the more the section of code is executed, the hotter that section is said to be.

- If the code is going to be execute only once – Interpretating is more advisable.
- If the code is going to be repeatedly call/need to be executed – Compiling the code is advisable.

Java is designed to take advantage of the platform independence of scripting languages and the native performance of compiled languages. A Java class file is compiled into an intermediate language (Java bytecodes) that is then further compiled into assembly language by the JVM. Compilation of the bytecodes into assembly language performs a number of optimizations that greatly improve performance

Hotspot of an application is that piece of the code that keeps getting referred again and again, it depends on the executing of that code how fast the code runs.

JIT Compiler comes with two flavours - Client & Server

The choice between two compliers is their aggressiveness in compiling the code. In general, the, code produced by the server compiler runs faster then that of the client compiler.

For long running application , it always better to use JIT server configuration in conjunction with tried compilation.

JVM starts with the client compiler and then use server compiler as its get hotter – this technique is know as tiered compilation. Since its hard to optimized tiered compilation, its not enabled by default in java 7, java 8 onwards its enabled by default.

There are three different JIT compiler version available

- 32 Bit client version (-client)

- 32 Bit server version (-server)
- 64 Bit server version (-d64)

When to used what?

- For 32-bit OS used 32-bit client version
- For 64-bit OS use 64-bit server / if heap is less then used 32-bit version.

Important Point - 32-bit JVM cannot exceed the total process size beyond 4 GB – that includes space for all the code, any native memory the application allocates, and the code cache.

32-bit and 64-bit installation are available separately, one can download both the installer and have them installed in separate paths.

Other JIT tuning

- Code Cache tuning: Since the code cache size is fixed, in case there are more complied code then the size of the cache, then even with complier code the code will run slower.

  The default allocated size can be altered with reserve memory configuration, once reversed , memory is not allocated until needed. Reserved memory is getting allocated to code cache, java heap various other native memory

- Compilation Triger: When JVM will deem that code need to complied is based on the follow two triggers
  - Number of times a method been called
  - Number of times a loop in the method have branched back.

    On-stack replacement (OSR) , the JVM start executing the complied version of the loop while the code is still running, the JVM once completion the compilation replace the code (stack) with complied version of the code, making the code run much faster.

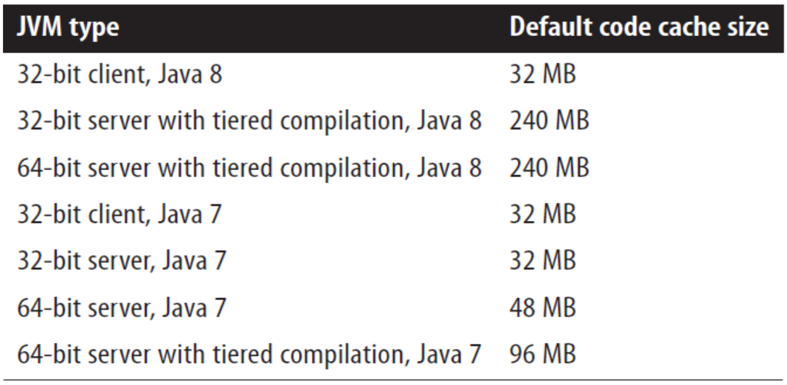
    Its common practice to change the CompileThreshold flag, a slight variance may have positive/negative impact on the overall executing time.

The attributes field is a series of five characters that indicates the state of the code being compiled.

- %: The compilation is OSR.
- s: The method is synchronized.
- !: The method has an exception handler.
- b: Compilation occurred in blocking mode.
- n: Compilation occurred for a wrapper to a native method.

The best way to gain visibility into how code is being compiled is by enabling PrintCompilation. Output from enabling PrintCompilation can be used to make sure that compilation is proceeding as expected.

COMPILE SKIPPED indicates there is an issue , it can be because of *Code cache filled OR Concurrent classloading.*



JVM type	Default code cache size
32-bit client, Java 8	32 MB
32-bit server with tiered compilation, Java 8	240 MB
64-bit server with tiered compilation, Java 8	240 MB
32-bit client, Java 7	32 MB
32-bit server, Java 7	32 MB
64-bit server, Java 7	48 MB
64-bit server with tiered compilation, Java 7	96 MB

Advance Compiler Tuning: Mostly this configuration is rarely needed.

Compilation Threads

There are one or more compilation threads, compilation is done asynchronously; while compilation is done using standard compilation, then the next method invocation will execute the compile method. If a loop is getting complied using OSR then the next iteration of the loop will execute the compile code.

-XX:CICompilerCount=*N* flag (with a default value given in the previous table). That is the total number of threads the JVM will use to process the queue(s); for tiered compilation, one-third of them (but at least one) will be used to process the client compiler queue, and the remaining threads (and also at least one) will be used to process the server compiler queue. This parameter has less overall effect on the performance .

-XX:+BackgroundCompilation flag , by default its true ; incase its made to change to false; in which case when a method is eligible for compilation , code that wants to execute it will wait until the method is complied. Background computation is disable when -Xbatch is specified.

Inlining

The overhead for invoking a method call like this is quite high, especially relative to the amount of code in the method. JVMs now routinely perform code inlining. Inlining is enable by default, disabling Inline reduce the performance. A given methos is eligiable for Inline , if its hot (frequently called) and the bytecode size is then defined by -XX:MaxFreqInlineSize=*N* flag. Its often recommende to increase the size of the -XX:MaxFreqInlineSize=*N* flag, so that more method are Inlined .

When tuning -XX:MaxFreqInlineSize=*N* flag; its also need to be taken into account frequent Inlining, else their wouldn't be much improvement in overall performance (we may see some performance boot during warmup).

Escape Analysis

-XX:+DoEscapeAnalysis, is true by default, server compiler will perform some aggressive optimization when flag is set. In rare scenarios, it been observed that disabling this flag would provide performance benefits, but in those particular cases it advisable to simplify the code.

Deoptimization

Deoptimization allows the compile to back out previous version of compile code; Code is deoptimized when the pervious optimization is no longer valid. These is a slight impact of the deoptimization, but soon the new code gets warmed up & get optimized.

Under tried compilation , code gets deoptimized if previously the code is compiled by client compiler and then later the code compiled by server compiler.

Tried Compilation Levels

There are four (4) different level of compilation

0: Interpreted Code
1: Simple C1 compiled code
2: Limited C1 complied code
3: Full C1 complied code
4: C2 compiled code
Typical compilation path 0 → 3 → 4 (Most common and efficient)

Alternative paths

- If the server compiler queue is full: 0 → 2 → 3 → 4
- If client compiler queue is full: 0 → 2 → 4 (method that is scheduled for compilation at level 3, will be eligible for compilation at level 4. In that case method get quickly complied at level 2 then get complied directly at level 4)

- For trivial methods: 0 → 2/3 → 1 (due to the nature of the trivial methods after getting compiled through level 2 or 3 it may go back and get compiled by level 1)
- Deoptimization: Any level → 0

Best performance is achieved when the code is getting compiled by 0 → 3 → 4, if the code is getting complied by 4, then consider increasing the CPU compiler threads.

NOTE:

1. Common performance parameters with reference to JIT configuration; use tiered compilation , ensure that code cache is large enough.

2. Presence Or absence of the final keyword will not affect the performance of an application.

Chapter 5: An Introduction to Garbage Collection

There are four (4) types of garbage collector – serial collector (use for single-CPU machines), the throughput (parallel) collector, the concurrent (CMS) collector m and the G1 collector.

One of the most attractive features of Java is that developers need not explicitly manage the lifecycle of objects: objects are created when needed, and when the object is no longer in use, the JVM automatically frees the object.

JVM identifies the objects that are not been used and then make them qualified for garbage collection.

JVM GC basic operation → finding unused objects, making their memory available, and compacting the heap. Different collector taking different approach to meet the above operations.

There are different generation – old generation and younger generation. The younger generation is further divided into many younger generations as a java object can made to use in for a very short period of time.

Garbage collection process

- The objects are more to the younger generation of object.
- Minor GC
  - Clear younger generation.
  - Move live objects to the superior or older generation.
  - Discard unused objects.
- Full GC : clear old generation.

GC Algorithms:

- Simple algorithms: stop-the world for full GC (stor app application threads to collect GC).
- Concurrent collectors (CSM, G1)
  - Find the unused objects while the app thread is running
  - Minimized pause time
  - Use more CPU overall.

Choosing right GC, based on the application type right GC algorithm is important.

- For response time-sensitive applications.
  - Concurrent collector algorithm minimized log pause.
  - Better reduce outline response time.
- For average response optimization
  - Throughput collector often yields better response time.
- For batch jobs
  - Concurrent collector if CPU available.
  - Throughput collector if CPU limited.

Concurrent collectors: shorter pauses, higher CPU usage.

Throughput collector: longer pauses, lover CPU limited.

GC Algorithms

- **The serial garbage collector** – This is the simplest GC, use in single processor / 32-bit windows machine, where single GC thread would do the GC either minor or full. During full GC the older generation gets fully compacted.
- **The throughput collector** – This is default collector for 64-bit server-class machine , in this case there are multiple threads to collect the younger generation , making minor GC faster, it can also use multiple threads to collect from the older generation , default this feature is enabled – can be changed through -XX:+UseParallelOldGC flag. It stops all application threads during both minor and full GCs. This GC is good for batch processing jobs.
- **The CMS Collector** – this is used to minimized the long pauses. Stops threads for doing minor GC, for collecting old generation it uses background threads, higher CPU uses. Since there is no compact of the memory, it can lead to fragmentation of the memory.
- **The C1 Collector** – ideal for the larger heap size (> 4GB) , there head is divided into several regions. Concurrent collector with background threads. During collection , it move the object from one region to other, it partially compacts the heap during processing. It has higher CPU usages.

Point to remember

- Serial GC simplest collector.
- Throughput GC for batch jobs.
- CMS and C1 offer concurrent collectors with shorter pause time but higher CPU usages.
- The main trade off in selecting the GC is Pause time Vs CPU usages.
- The choice of the GC depended on the application need – response time Vs throughput.
  NOTE: Throughput and response time are inversely proportional.

Choosing a GC Algorithm

Serial Collector:

- Best for application using less than 100 MB heap.
- Suitable for small heaps where parallel collection is not beneficial.

Throughput Collector:

- Throughput collector Pause affects the overall performance of the application.
- Uses multiple GC threads during collection.

Concurrent Collector:

- Background threads periodically consumes CPU.
- Can lead to higher overall CPU usages.
- Shorter but more frequent pause.
- In multiple-CPU scenarios , Concurrent collector beneficial if extra CPU is available (in single CPU, it may not offer any advantages).

The key factor for selecting the GC algorithm

- Available of CPU resources.
- Acceptable pause times.
- Overall application performance requirement of the application.

Trade off

- Throughput collector: Longer pauses, potentially better overall performance.
- Concurrent collectors: Shorter pauses, higher CPU usage, potentially faster for multi-CPU systems.

Choosing between CMS and G1 Collector algorithms. CMS outperformed G1 collector when the heap size is relatively small, as it need to first scan the heap before claiming any freed objects , while G1 breaks the heap into smaller region so that its easier for the multiple background threads to work in parallel. However, G1 Collector algorithm is relatively newer than other algorithms.

Basic GC Tuning

- **Sizing the Heap** : If the size of the heap is too small , most of the time performing GC and less time spending on running application logic, equally if the heap is too large then the OS will offer some of the heap through RAM and some through OS, this will lead to constant swapping which will result in performance degradation. The size of the heap can be configured by -XmsN and -XmxN parameters.

  Setting of heap size is optional, if the application is doing too much GC the JVM will automatically increase the heap size to 'correct' value. Its advisable to keep 30% heap occupied. If the required application heap size is known, then JVM need to figure out the size of the heap it need to re-configure to, this makes the GC more efficient. The JVM will attempt to find a reasonable minimum and maximum heap size based on the machine it is running on.

- Sizing the Generations : Its equally important to have generation rightly sized, if the young generation is too small, then there need a frequent GC , if the size of the young generation is too small; then more objects will need to move to older generation leading to performance efficiently.

  Each algorithm has it own way to size the region within heap, but the parameters that governing the sizing is same.

  -XX : NewRatio = N |The ratio of the new and old generation.
  -XX: NewSize = N | TMhe size of the new generation. The old generation would get the remain heap.
  -XX : MaxNewSize = N | Maximum size of the young generation.
  -XXmnN | shorthand for setting both NewZie and MaxNewSize to the same value.

- **Sizing Permgen and Metaspace** When JVM loads classeses, it must also keep track of certain metadata about those classes, in Java 7 the space was called as premgen , Java 8 its known as metaspace. Premgen and metaspace are the same thing, some of the unrelated metadata class information were move to heap in Java 8, keeping only class metadata information alone on metadata space. There isn't a good way to calculate permgen space .
- Controlling Parallelism Except serial collector perform GC operation in parallel , the number of these threads is controlled by -XX:ParallelGCThreads=*N* flag. The value of this flag affects the number of threads used for the following operations:

  - Collection of the young generation when using -XX:+UseParallelGC
  - Collection of the old generation when using -XX:+UseParallelOldGC
  - Collection of the young generation when using -XX:+UseParNewGC
  - Collection of the young generation when using -XX:+UseG1GC
  - Stop-the-world phases of CMS (though not full GCs)
  - Stop-the-world phases of G1 (though not full GCs)

  The total number of the parallel thread can be counted by the below formula , where N is the number of CPU core

  ParallelGCThreads = 8 + ((N - 8) * 5 / 8)

Adaptive Sizing: Using Adaptive sizing JVM will find the optimal size of the young generation and old generation, without needed to set it explicitly, however this can be disabled by setting -XX:-UseAdaptiveSize Policy flag (which is true by default) or by setting maximum and minimum heap size to same value or by setting initial and maximum size of the new generation to the same value.

GC Tools

GC log provides an insight on GC operation, one need to enable the GC logs its stop by default. Also, there are tool like GC histogram can present the log content in more readable format. Apart from monitoring logs, there are other tools like jconsole & jstat (preferred for scripting)

Important Notes

1. Heap Sizing:
    - Initial size: -XmsN
    - Maximum size: -XmxN
    - Rule of thumb: Size heap so it's 30% occupied after full GC
    - Never set larger than physical memory
    - Consider leaving 1GB for OS and other apps
2. Generation Sizing:
    - Young generation controlled by:
        - -XX:NewRatio=N (default 2, meaning 1/3 young, 2/3 old)
        - -XX:NewSize=N (initial size)
        - -XX:MaxNewSize=N (maximum size)
        - -XmnN (shorthand for setting both NewSize and MaxNewSize)
    - Old generation gets remaining heap space
3. Permgen/Metaspace:
    - Java 7 and earlier: Permgen
    - Java 8+: Metaspace
    - Sizing flags:
        - Permgen: -XX:PermSize=N, -XX:MaxPermSize=N
        - Metaspace: -XX:MetaspaceSize=N, -XX:MaxMetaspaceSize=N
4. Controlling Parallelism:
    - -XX:ParallelGCThreads=N
    - Default: 1 thread per CPU up to 8, then 5/8 thread per additional CPU
5. Adaptive Sizing:
    - Enabled by default (-XX:+UseAdaptiveSizePolicy)
    - Can be disabled for finely tuned applications
6. GC Logging:
    - Enable with -verbose:gc or -XX:+PrintGC
    - Detailed logging: -XX:+PrintGCDetails
    - Add timestamps: -XX:+PrintGCTimeStamps or -XX:+PrintGCDateStamps
    - Specify log file: -Xloggc:filename
    - Log rotation: -XX:+UseGCLogFileRotation, -XX:NumberOfGCLogFiles=N, -XX:GCLogFileSize=N
7. GC Monitoring Tools:
    - GC Histogram: For analyzing GC logs
    - jconsole: Real-time heap monitoring
    - jstat: Command-line tool for GC statistics

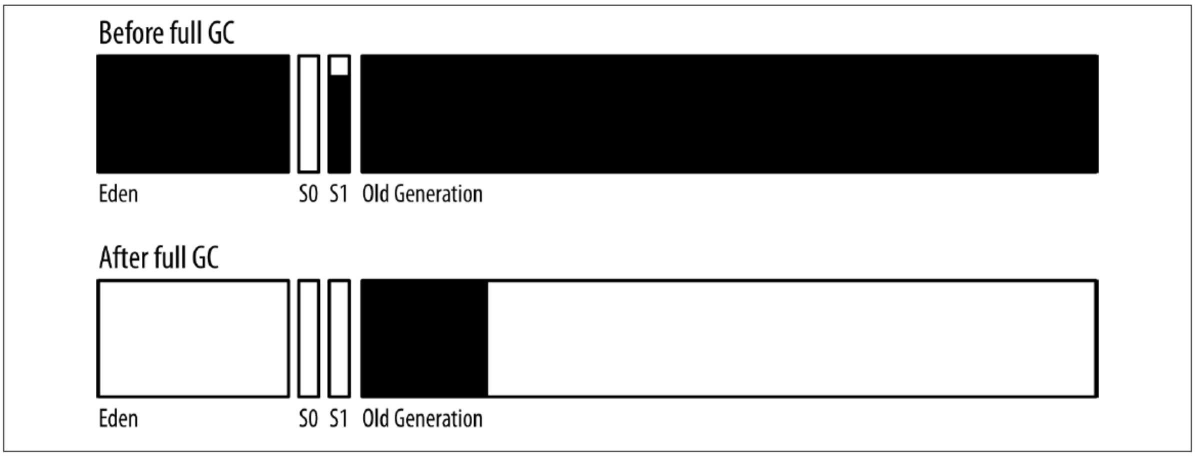
Remember:

- Balance heap size for performance
- Consider application characteristics when tuning
- Use GC logs and monitoring tools for optimization
- Adaptive sizing works well for many applications
- Tune parallelism based on available CPUs and workload

Chapter 6: Garbage Collection Algorithms

Throughput Collector

The main operation of throughput collector are – minor collections and full GCs. GCLogs is an effective way to determine the overall impact of the GC on an application using throughput collector.

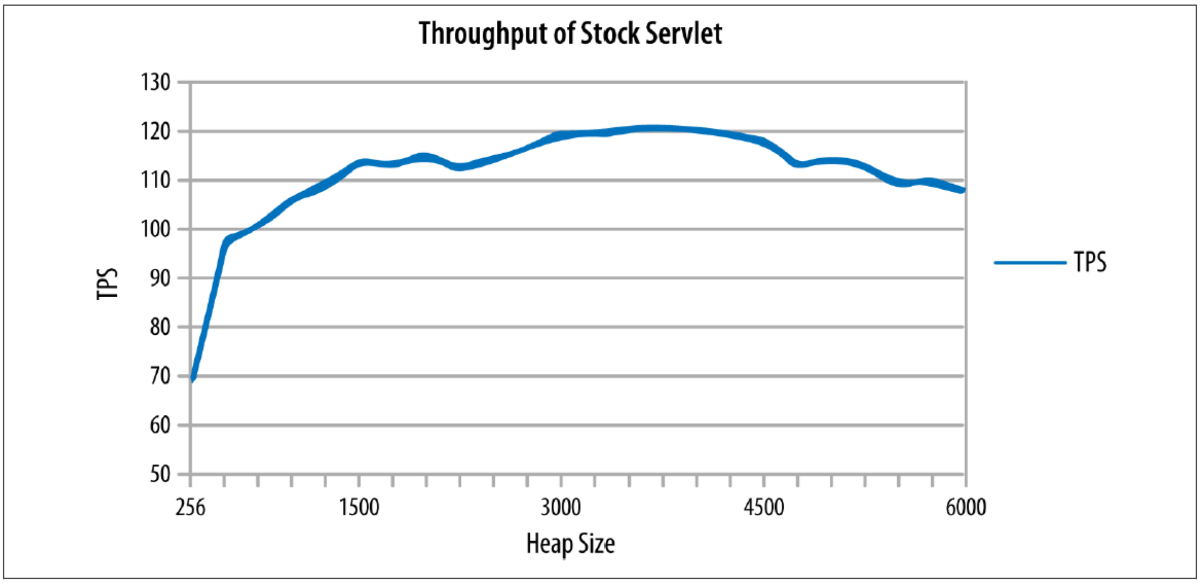


Adaptive and Static Heap Size Tuning

Trade-offs

1. Time Vs Space:
2. Time it takes to perform GC

As the heap size increases the application throughput increases too, until a point where with the increase in the heap size application throughput does not improves largely. Beyond this point, the application performance decreases as the heap size increases.



The adaptive sizing of the throughput collector will resize the heap to meet it pause time goa. The goal is set through these flags -XX:MaxGCPauseMillis=*N* and -XX:GCTimeRatio=*N*.

Where MaxGCPauseMillis specifies the time application is will tolerate pause time, and GCTimeRatio is the time application is will to spend in GC. Both these values are connect to throughput by the below formula.

Throughput = 1 – (1 / 1 + GCTimeRatio) ; lesser the GCTimeRatio value higher the throughput value is.

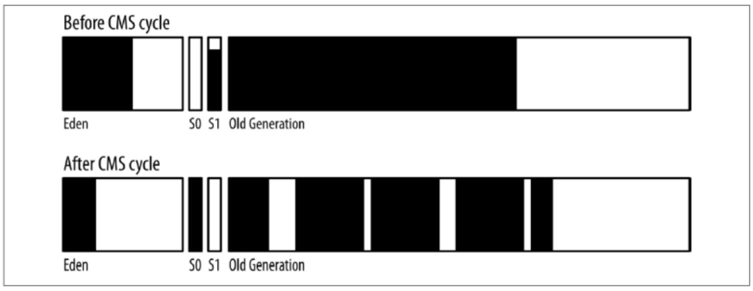
OR *GCTimeRatio* = *Throughput* /(1 - *Throughput*)

CMS Collector

CSM has three basic operations

- CMS collects the young generation (stopping all application threads)

- CMS run a concurrent cycle to clean data out of the old generation.
- If necessary, CMS perform a full GC.



Like throughput collector, eden objects are move to survivor space or to the older generation. When the heap is sufficiently full JVM will trigger concurrent cycle to remove the objects, but in this process the older generation will not be compacted.

- Marking phase will mark the objects, (application thread run concurrently with this phase)
- PreClean phase will trigger when the young generation is 50% full this is done to avoid back-to-back pauses. (*CMS uses past behaviour to calculate when the next young collection is likely to occur)*
- Sweep phase, this phase run concurrently with young generation collection. This is the last concurrent phase.
- CMS cycle ends – the unreferenced object found in the old generation are freed.

CMS collector encounter issues under following circumstances when concurrent mode or promotion failure.

- When the young generation grows and there is not enough space in old generation to hold all the objects that are expected to be promoted. CMS executes a full GC – this pauses all application thread and clean up any dead object from the old generation. CMS full GC execution happens as a single threaded application.
- When the old generation is too fragmented to hold all the objects promoted from the younger generation. As a result, CMS stopped all the threads and start collecting and compacting the entire old generation.
- When the sized of the permgen is full, CMS does not collect permgen/metaspace, when its full, full GC is required to clear the permgen/metaspace.

When its 70% full, CMS start scanning the old generation for dead object through a concurrent cycle before the old generation fills up. There are multiple ways to avoid concurrent mode failure.

- Make old generation larger – shift portion of young generation to new generation OR add more heap space.
- Run the background thread more often Using flags -XX:CMSInitiatingOccupancyFraction=N and -XX:+UseCMSInitiatingOccupancyOnly flags makes CMS easier to determine when to start the background thread based only on the percentage of the old generation that is filled. (Instead of 70% fill , run it when 60% fill)
- Use more background thread. If the number of ConcGCThread is set too high, they will take most of the CPU Cycle; and the application need to wait for running it application threads.

Tuning CMS for Permgem

- CMS Collector default don't collect Permgem
- When Permgem is full , it triggers full GC
- -XX:+CMSPermGenSweepingEnabled (default value false) if this flag is turned on then permgen is collected like old generation object.
- -XX:CMSInitiatingPermOccupancyFraction=*N*,(default value 80%) , only when this thrushold is meet the permgem collection is triggered.
- -XX:+CMSClassUnloadingEnabled flag is not set then, CMS permgen sweeping will manage to free a few miscellaneous objects, but no class metadata will be freed.
  NOTE : In java 8 CMS does clean unloaded classes from metaspace by default; hence -XX:+CMSClassUnloadingEnabled need not to be set.

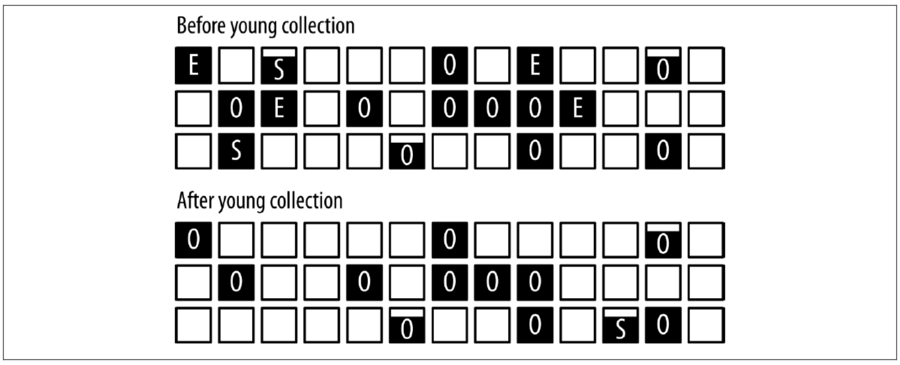
Incremental CMS

- Deprecated in Java 8, availability of this option in Java 9 onwards (need to check this).
- This single core / supper busy CPUs – where the background thread paused periodically to allow application thread to run.

Understanding G1 Collector

G1 concurrent collector operates on discrete regions within the heap. Heap can have many region (by default there are 2048 region) , each region has different level of garbage (some have more then others), G1 collector stop the threads during garbage collection – but focus on regions which are mostly garbage and spend less time in emptying those regions. Hence it called Garbage First OR G1.

Any live data in those regions was moved to another region (just as live data was moved from the young generation into regions in the old generation). This is why G1 ends up with a fragmented heap less often than CMS—moving the objects like this is compacting the heap as G1 goes along.



E = Eden
O = Old Generation
S = Survivor Space

Under following scenarios when full GC is observed in GC logs, it is an indication that tunning or increasing of the heap can be beneficial.

- **Concurrent Mode Failure**: When C1 start the marking , but the old generation is getting filled up before cycle can complete. Resulting in aborting the marking process. Remedy – Increase memory, G1 background processing must begin sooner OR Cycle must be tuned to more quickly.
- **Promotion Failure**: G1 complete the marking process and start performing mixed GCs to clean up old regions, but old generation runs out of space before enough memory is reclaimed. Remedy – collection needs to happened more quickly, and you collection needs to process more region in the old generation.
- **Evacuation Failure**: When performing young collection, there is not enough room in survivor space and old generation to hold all the surviving objects. This indicates that the heap is largely full or fragmented. Remedy increases heap size.
- **Humongous allocation failure**: Applications that allocates very large objects can trigger another kind of full GC in G1 ; there is no clear way to identify this.

G1 Tuning Goals

1. Avoid Concurrent failures & Evacuation failures.
2. Minimized pauses

G1 Tuning Tips

1. Increase sized of the old generation – by increasing heap OR by adjusting the ratio between the generation.
2. Increase the number of background threads.
3. Tune to perform G1 background activities more frequently.
4. Increase the amount of work-done in mixed GC cycles.

G1 Trade off

By reducing the young generation size to meet the pause time goal, this will also decrease the collection of old generation by mixed GC which will increase the chance of concurrent mode failure.

Tuning G1 background (marking) threads, this can be achieved by tuning ParallelGCThreads and ConcGCThreads flags . Default value of the concurrent thread is define by

Concurrent Thread = (ParallelGCThreads + 2 / 4)

Tuning G1 to run more frequently , this can be achieved by setting right value to -XX:InitiatingHeapOccupancyPercent=*N*, the occupancy ratio specified by this flag determines when the G1 will be running . default its set to 45%. Setting this value too high will result in running fewer G1 process – resulting in filling up of the heap more frequently, setting this value too small will resulting in running too many G1 background threads which will stop the application threads.

Tuning G1 mixed GC cycles

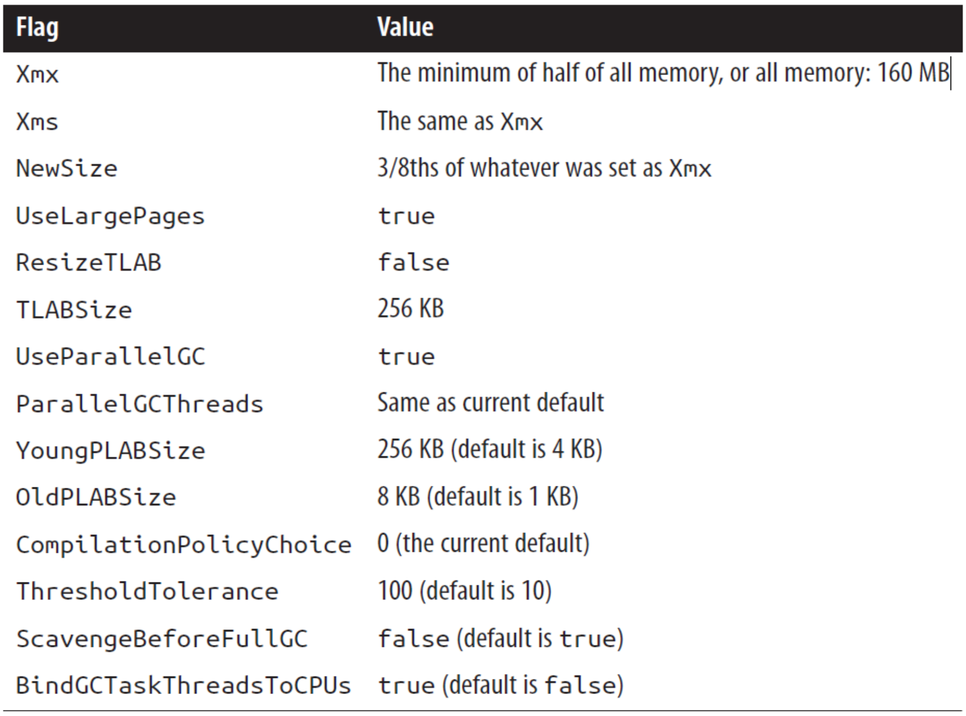
- **Concurrent Cycle Limitation:** G1 cannot start a new concurrent cycle until all previously marked regions in the old generation are collected. This means that efficient collection is crucial for timely GC cycles.
- **Mixed GC cycle work:** the effectiveness of the mixed GC cycle depends upon
  1. Amount of garbage in the region (-XX: G1MixedGCLiveThresholdPercent=*N)*, a region is eligible for garbage collection if its at 35% garbage. Currently this parameter is not tunable in Java 7 or 8.
  2. Maximum number of mixed GC (-XX: G1MixedGCCount Target=N.), default its set to 8.
  3. Maximum desired length of the GC pauses (MaxGCPauseMillis).

Advance Tunings

- Tenuring and Survivor space
  - The **tenuring threshold** determines when objects are promoted from the young generation to the old generation. This threshold can range from 1 to the maximum defined by MaxTenuringThreshold. Even if the JVM starts with the maximum threshold, it may decrease it based on the application's behaviour.
  - **The Always Tenure** Flag - If certain objects are expected to survive young collections for a long time, you can use the -XX:+AlwaysTenure flag. This effectively promotes these objects directly to the old generation, bypassing the survivor space. However, this scenario is quite rare.
  - **Survivor Spaces.** The JVM uses survivor spaces to hold objects that survive young collections. The size of these spaces can be fixed by setting the SurvivorRatio and disabling the UseAdaptiveSizePolicy flag. By default, the JVM aims to keep survivor spaces about 50% full after a GC cycle, but this can be adjusted using the -XX:TargetSurvivorRatio=N flag.
  - **Promotion to Old Generation** Objects are moved to the old generation under two conditions: When survivor spaces are small and fill up during a young collection, remaining live objects in the Eden space are promoted directly to the old generation. There is a limit on how many GC cycles an object can remain in the survivor spaces, known as the tenuring threshold**.**
- **Allocating Large Objects**
  - In the JVM, "large" objects are relative and depend on the size of the Thread-Local Allocation Buffer (TLAB). For example, in a 2 GB heap, objects larger than 512 MB are considered very large.
  - Allocating large objects can significantly affect the performance of GC algorithms. TLAB sizing is crucial for all GC algorithms, but G1 has specific considerations for very large objects.
  - When allocating very large arrays (e.g., an array of **3.1 MB** with a G1 region size of **1 MB**), the JVM must find **four contiguous regions** in the old generation. This can lead to wasted space in the last region and may require a full GC to find the necessary contiguous regions.
  - The need for contiguous regions for large objects can defeat G1's usual compaction strategy, which is to free arbitrary regions based on their fullness. This can complicate memory management and lead to inefficiencies.

The **AggressiveHeap** flag is designed for optimizing JVM performance on large machines with significant memory. It applies only to 64-bit JVMs and is no longer recommended due to potential performance issues and lack of transparency in the tunings it applies. Default Setting: The flag is set to false by default. he flag may hide actual tunings, making it difficult to understand JVM settings. Its effectiveness can vary with JVM upgrades, requiring reevaluation of its benefits.

Tunings Enabled with Aggressive Heap.



Flag	Value
Xmx	The minimum of half of all memory, or all memory: 160 MB
Xms	The same as Xmx
NewSize	3/8ths of whatever was set as Xmx
UseLargePages	true
ResizeTLAB	false
TLABSize	256 KB
UseParallelGC	true
ParallelGCThreads	Same as current default
YoungPLABSize	256 KB (default is 4 KB)
OldPLABSize	8 KB (default is 1 KB)
CompilationPolicyChoice	0 (the current default)
ThresholdTolerance	100 (default is 10)
ScavengeBeforeFullGC	false (default is true)
BindGCTaskThreadsToCPUs	true (default is false)

Heap Size

- The JVM's heap size is crucial for performance and is determined by the amount of physical memory available on the machine.
- Default values for initial and maximum heap sizes depend on the JVM and the machine's memory.
- For Systems having large available memory, the Max Size is set default to 32 GB , size of the heap is calculated using the following formula.

  Default Xmx (heap size) = MaxRAM (max memory allocated) / MaxRAMFraction (defaulted to 4).

  For example for 128 GB Max RAM allocation, 32 GB of Heap size will be allocated as -XX:MaxRAMFraction=*N* flag is defaulted to 4. (Quarter of full memory).

  Also -XX:ErgoHeapSizeLimit=*N* flag can be used to set the maximum heap allocated to an JVM. Default its set to 0 (*meaning to ignore it*), otherwise the limit should be less then MaxRAM / MaxRAMFraction

The following are the several critical aspects of optimizing Java Virtual Machine (JVM) performance.

1. **Compilation Thresholds**:
   - Tuning can affect compilation thresholds, which are based on two counters: the number of times a method has been called and the number of times loops in the method have branched back. This tuning is essential for optimizing method performance, particularly in frequently called methods.
2. **Memory Management**:
   - The performance gap in 64-bit JVMs arises from the larger object references (8 bytes) compared to 32-bit references (4 bytes). This increased size can lead to more frequent garbage collection (GC) cycles due to reduced available heap space. To mitigate this, the JVM can use compressed ordinary object pointers (oops), allowing it to reference larger memory spaces efficiently.
3. **Process Limitations**:

- On Unix-style systems, the maximum number of processes configured for a user can impact performance. Reducing stack size may help, but it won't resolve issues related to process limits. This is an important consideration when encountering JVM errors.

4. **Large Pages**:
   - Memory allocation is managed in units called pages, which are the minimum allocation units for the operating system. When memory is allocated, an entire page is reserved, and further allocations come from that page until it is filled. The operating system uses paging to manage memory efficiently, moving pages in and out of physical memory as needed. This involves a global page table and translation lookaside buffers (TLBs) for faster access.

Chapter 7: Heap Memory Best Practices

Knowing which objects consume memory is crucial for optimization. Histograms can quickly identify memory issues caused by excessive instances of certain object types.

For deeper analysis, heap dumps are essential. They provide a detailed view of memory usage. You can generate heap dumps using commands like:

Purpose of Heap Dumps: While histograms can identify issues related to excessive instances of specific classes, heap dumps are necessary for deeper analysis of memory usage. They provide a comprehensive view of all objects in the heap at a specific moment.

- jcmd process_id GC.heap_dump /path/to/heap_dump.hprof
- jmap -dump:live,file=/path/to/heap_dump.hprof process_id

The live option forces a full garbage collection (GC) before the dump, while -all includes dead objects.
Several tools can analyze heap dumps:

- **jhat**: Analyzes heap dumps and provides a web interface for exploration.
- **jvisualvm**: Takes heap dumps from running programs and allows browsing of the heap.
- **MAT (Memory Analyzer Tool)**: Loads multiple heap dumps and generates analysis reports.

While tuning the garbage collector is important, better performance often comes from implementing good programming practices. This includes optimizing memory usage and object allocation.

Out of Memory

- **Out of Memory Errors** can occur for various reasons; it is essential not to assume that heap space is always the issue. Common causes include:
  - No native memory available for the JVM.
  - Permgen (Java 7 and earlier) or Metaspace (Java 8 and later) running out of memory.
  - Java heap being out of memory due to too many live objects for the configured heap size.
  - Excessive time spent on Garbage Collection (GC).
- **Common Error Messages:** When the heap is out of memory, the error message appears as:
  - Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
  - Exception in thread "main" java.lang.OutOfMemoryError: Metaspace (In Java8+)
  - Exception in thread "main" java.lang.OutOfMemoryError: PermGen space (In Java7)
- **The JVM throws an Out of Memory Error under the following circumstances:**
  - Lack of native memory.
  - Exhaustion of permgen or metaspace.
  - Insufficient heap space for live objects.
  - High GC activity, which may indicate underlying memory issues.
- **Solution:** If the application has too many live objects, consider:

  - **Increasing the heap size** to accommodate more objects.
  - Investigating for **memory leaks**, where the application allocates more objects without releasing them. Increasing heap size in this case may only delay the error.
  - Setting Automatic heapDump using following JVM flags
    - -XX:+HeapDumpOnOutOfMemoryError (default value false)
    - -XX:HeapDumpPath=<path> (default is *java_pid<pid>.hprof* in the application’ s current working directory.)
    - -XX:+HeapDumpAfterFullGC
    - -XX:+HeapDumpBeforeFullGC
  - If there are increasingly large number of objects found in target heap dump as compare to baseline heap dump, its then advisable to change the logic of the application code to proactively discarded items from the collection that are no longer needed OR use weak or soft references can automatically discard the items when nothing else in the application is referencing them
- GC Overhead limit reached : Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded.
  This exception is seen when all following conditions are meet.
  - Amount of time spent in full GC is exceed then the value specified in -XX:GCTimeLimit=*N* flag.
  - Amount of memory reclaimed by fullGC is less than the value specified in -XX:GCHeapFreeLimit=*N* flag.
  - The above two condition have held true for five consecutive full GC cycles. [***This is not configurable parameter***]
  - When -XX:+UseGCOverheadLimit flag is true. (by default this flag is set true).

Using Less Memory
The primary goal is to use less heap memory, which leads to fewer instances of the heap filling up and consequently reduces the frequency of garbage collection (GC) cycles. This can enhance overall application performance.

Impact of Reducing Memory Usage

- By minimizing memory usage, the young generation collections occur less frequently. This means that objects are less likely to be promoted to the old generation, which can lead to a decrease in the number of full GC cycles.
- Fewer full GC cycles can help clear more memory, further reducing the frequency of these cycles, creating a positive feedback loop for memory management.

Methods to Reduce Memory Usage:

- **Reducing Object Size**: Making objects smaller is a straightforward way to decrease memory consumption. For instance, a 20% reduction in the size of half the objects can have a significant impact on overall memory usage, potentially achieving the same effect as increasing the heap size by 10%

Lazy Initialization section discusses a design pattern that optimizes resource usage by delaying the creation of an object until it is needed. This technique is particularly useful when an instance variable is not always required.

Synchronization, can be bottle neck for lazy initiation object, when a rarely used object suddenly been used by multiple threads; to solve this problem the lazy initiation objects should be thread-safe.

Thread Safety Considerations: When implementing lazy initialization in a multi-threaded environment, it is crucial to declare the instance variable as volatile to ensure visibility across threads. Additionally, assigning the instance variable to a local variable can provide a slight performance benefit. This approach helps to avoid threading issues that can arise when multiple threads attempt to initialize the variable simultaneously.

```
NOTE: Volatile keyword is a modifier that ensures that a variable's value is always read from
and written to main memory, rather than from a thread's local cache.
```

Egger De-Initialize, the opposite of lazy initialization is egger de-initialization where an object is set to null; making it qualify for early garbage collection. Use of local variable instead of instance variable also help, as the code moves out of the method the local variables are fall out of scope and qualifies for the garbage collection.

In collection classes where class holds the data reference for a long time, anything that are no longer needed, should be actively set to null this free up the memory held by the object making it early qualify for garbage collection.

Immutable and Canonical Objects

Immutable objects are those that can't be updated, java have many immutable objects – its advisable to use immutable objects to define custom classes; though it may sound counter productive as the objects can't be change after creating – but its been observed that object that are quickly create and discarded have less impact on young memory which leads to the better utilization of the heap memory.

String Interning

If the application is using large number of duplicate Strings, then using String Intern () method can be helpful in improving the memory allocation and the overall application performance. TO identify if application has too many duplicate String, first load the heap dump into Eclipse Memory Analyzer, then calculate the retained size of the string objects – string with retained size same are mostly duplicate string.

The intern() method in Java's String class is used to store only one copy of each distinct string value in a pool of strings. This helps conserve memory and can improve performance in certain scenarios.

When using String Intern method save the String value in native memory under String Table (which is a fixed size hashtable). The sized of the string hashtable is between 1009 to 60,013 buckets, the string hashtable needs to be correctly sized by -XX:StringTableSize=N , the performance of the string intern method is dependent on the how well String Table Size is tuned.

One can view the performance of the String Table by printing the String table statistics through -XX:+PrintStringTableStatistics argument (default value is set to false). The number of interned string can also be obtained by jmap command.

Object Lifecycle Management

Java typically manages object lifecycles automatically. Developers create objects as needed, and when they go out of scope, they are cleared by the garbage collector. n scenarios where object creation is costly, it may be beneficial to alter the standard lifecycle. This can involve reusing objects or maintaining special references to them, which can improve performance despite requiring more effort from the garbage collector. By managing the lifecycle of expensive objects effectively, applications can achieve better efficiency and resource utilization, which is crucial for performance-sensitive applications.

Object Reuse: Object reuse is a technique used to improve performance by reusing existing objects instead of creating new ones.

The common methods of object reuse are:

1. **Object Pools**: A collection of reusable objects that can be borrowed and returned, reducing the overhead of object creation. Commonly used in scenarios like JDBC connection pooling.

   The performance of the object pool is impacted by

   - GC Impact: Reuse objects tend to stay longer in memory which largely impact GC efficiency.
   - Synchronization: Pool object needs to be synchronized, if the object are frequently removed & replaced, the pool have lot of contamination which leads to decrease the efficiency of using pool object as the object creation takes less time then using object from the pool.
   - Throttling: Pool leads to throttle when there is increasingly large number of request then the system can handle. This leads to degradation of the performance of the system.

2. **Thread-Local Variables**: Variables that are local to a thread, allowing for efficient reuse without interference from other threads.

   The following tread-off using thread local variable need to consider

   - Lifecycle management – unlike pool objects, the thread local object don't need to b returned to the pool. Thread-local object are available within the thread and needn't to be explicitly returned.

- Cardinality: There is no direct one-to-one correspondence between the number of threads and the save objects, unlike object pool the thread local do not provide any throttling feature.
- Synchornization: Since the thread local object are used within a same thread there is no need to synchronized objects.

Key consideration while reusing an object

1. Object initialization cost: Reusing to be consider when the cost of the object creation is very high.
2. Garbage collection impact: While the object reusing can improve the performance of the application, but it also adds up to the longer GC cycle as the reuse objects are tend to stay longer in memory.

To address the Synchronization problem – create new instance of the Random class each time , this would help to resolve the synchronization problem with next() method. ThreadLocalRandom class have better performance from creating a new instance.

NOTE: if initialization of object is taking longer time, then, one should explore pooling Or use of thread-local variable to reuse those expensive-to-create objects.

Weak and soft references in Java are special types of references that help manage memory more efficiently, especially for objects that are expensive to create or maintain. Here's a simple breakdown:

1. **Weak References**:

   - These references allow the garbage collector to reclaim the referenced object more aggressively. If an object is only weakly referenced, it can be collected during any garbage collection cycle, even if it is still in use by other threads. This is useful when you want to ensure that memory is freed up quickly. 9

2. **Soft References**:

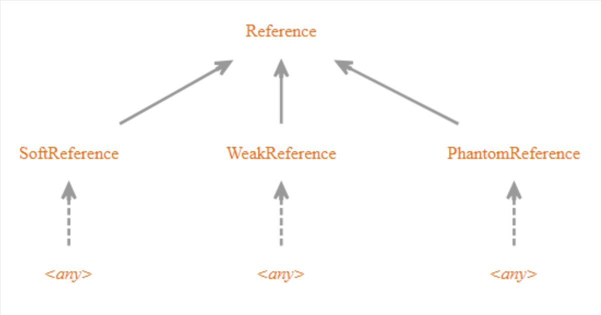
   - Soft references are used for objects that you want to keep around as long as there is enough memory. If the JVM needs memory, it can reclaim these objects, but they will be kept if the memory is available. This is like having a cache that holds onto objects that are likely to be reused, but can be cleared when memory is tight. For example, a stock history cache can use soft references to store data that may be requested again. 10

3. **Indefinite (Phantom) References**:

   - This term encompasses both weak and soft references, distinguishing them from strong references (ordinary object references). Indefinite references are often used to cache results from expensive operations, like calculations or database queries. This way, if the result is needed again, it can be reused without incurring the cost of recalculating or re-fetching it. 11

In summary, weak, and soft references help manage memory by allowing objects to be reclaimed when they are no longer needed, while still providing the option to reuse them when necessary. This balance helps improve application performance and memory efficiency.

Reference Class Hierarchy



1.Weak Reference

Weak references are objects that do not prevent their referents from being garbage collected. If an object is only weakly reachable (i.e., all references to it are weak references), it will be garbage collected in the next collection cycle.

```
import java.lang.ref.WeakReference;

public class WeakReferenceExample {
    public static void main(String[] args) {
        // Create a strong reference
        Integer num = new Integer(50);

        // Create a weak reference to num
        WeakReference<Integer> weakNum = new WeakReference<>(num);

        // Remove strong reference
        num = null;

        // Try to get the object through weak reference
        System.out.println("Before GC: " + weakNum.get());

        // Suggest garbage collection
        System.gc();

        // Try to get the object again
        System.out.println("After GC: " + weakNum.get());
    }
}
```

In this example, after we set num to null and suggest garbage collection, the weakly referenced Integer object is likely to be collected, and weakNum.get() will return null.

2. Soft Reference

Soft references are similar to weak references but are collected less aggressively. Objects with only soft references are garbage collected only when the JVM is low on memory.

```
import java.lang.ref.SoftReference;

public class SoftReferenceExample {
    public static void main(String[] args) {
        // Create a strong reference
        Integer num = new Integer(100);

        // Create a soft reference to num
        SoftReference<Integer> softNum = new SoftReference<>(num);

        // Remove strong reference
        num = null;

        // Try to get the object through soft reference
        System.out.println("Before GC: " + softNum.get());
        // Suggest garbage collection
        System.gc();

        // Try to get the object again
        System.out.println("After GC: " + softNum.get());
    }
}
```

In this case, even after garbage collection, the softly referenced Integer object may still be available if the JVM isn't under memory pressure.

3. Phantom Reference
Phantom references are the weakest type of reference. An object with only phantom references will be finalized and collected immediately. They are often used for more flexible memory management or to be notified when an object has been collected.

```
import java.lang.ref.PhantomReference;
import java.lang.ref.ReferenceQueue;

public class PhantomReferenceExample {
    public static void main(String[] args) {
        // Create a strong reference
        Integer num = new Integer(200);

        // Create a reference queue
        ReferenceQueue<Integer> refQueue = new ReferenceQueue<>();

        // Create a phantom reference to num
        PhantomReference<Integer> phantomNum = new PhantomReference<>(num, 
refQueue);

        // Remove strong reference
        num = null;

        // Suggest garbage collection
        System.gc();

        // Check if the phantom reference is enqueued
        System.out.println("Is phantom reference enqueued? " + 
phantomNum.isEnqueued());

        // Try to get the object (always returns null for phantom references)
        System.out.println("Phantom reference get(): " + phantomNum.get());
    }
}
```

In this example, after garbage collection, the phantom reference will be enqueued in the refQueue. Note that phantomNum.get() always returns null; phantom references are used for notification of object finalization, not for accessing the object.

Phantom references are indeed quite different from weak and soft references, and their primary purpose is not to access objects but to be notified when an object has been finalized and is about to be garbage collected.

Unlike weak and soft references, phantom references always return null when you call their get() method. This is by design.

An object that is only phantom-reachable will have its finalize() method called (if it has one), but it won't be immediately collected.

After finalization, the phantom reference is added to its associated ReferenceQueue (if one was provided when creating the reference).

Different use cases of Weak Reference Object , Soft Reference Object and Phantom Reference Objects.

Real-world use cases for weak references:

1. Caching: In the example above, WeakHashMap is used to implement a cache. This allows the JVM to reclaim memory if it's needed elsewhere, without the risk of OutOfMemoryErrors due to the cache growing too large.
2. Listener/Observer pattern: Weak references can prevent memory leaks in event handling systems. If a listener is no longer in use but forgot to unregister, a weak reference allows it to be garbage collected.
3. Object pooling: Weak references can be used in object pools to allow unused objects to be collected when memory is tight.

```
// Use Case 1: Caching
import java.lang.ref.WeakReference;
import java.util.WeakHashMap;

class DataProvider {
    private WeakHashMap<String, WeakReference<ExpensiveObject>> cache = new
WeakHashMap<>();

    public ExpensiveObject getExpensiveObject(String key) {
        WeakReference<ExpensiveObject> ref = cache.get(key);
        if (ref != null) {
            ExpensiveObject obj = ref.get();
            if (obj != null) {
                return obj;
            }
        }

        // Create new object if not in cache or has been collected
        ExpensiveObject newObj = new ExpensiveObject(key);
        cache.put(key, new WeakReference<>(newObj));
        return newObj;
    }
}

// Use Case 2: Listener management
class EventManager {
    private List<WeakReference<EventListener>> listeners = new ArrayList<>();

    public void addListener(EventListener listener) {
        listeners.add(new WeakReference<>(listener));
    }

    public void notifyListeners() {
        listeners.removeIf(ref -> ref.get() == null);
        for (WeakReference<EventListener> ref : listeners) {
            EventListener listener = ref.get();
            if (listener != null) {
                listener.onEvent();
            }
        }
    }
}
```

Real-world use cases for soft references:

1. Memory-sensitive caches: In the example above, an image cache uses soft references. This allows the system to keep images in memory as long as there's enough available, but allows them to be collected if memory becomes scarce.
2. Temporary buffer pools: Soft references can be used to maintain a pool of reusable buffers. The buffers will remain in memory if possible, improving performance, but can be collected if memory is needed elsewhere.
3. Large data structures in memory-constrained environments: For applications dealing with large datasets that ideally should stay in memory but can be recomputed/reloaded if necessary.

```
// Use Case 1: Memory-sensitive cache
import java.lang.ref.SoftReference;
import java.util.HashMap;
import java.util.Map;

class ImageCache {
    private Map<String, SoftReference<BufferedImage>> imageCache = new
HashMap<>();

    public BufferedImage getImage(String path) {
        SoftReference<BufferedImage> ref = imageCache.get(path);
        if (ref != null) {
            BufferedImage img = ref.get();
            if (img != null) {
                return img;
            }
        }

        // Load image if not in cache or has been collected
        BufferedImage newImg = loadImage(path);
        imageCache.put(path, new SoftReference<>(newImg));
        return newImg;
    }

    private BufferedImage loadImage(String path) {
        // Load image from file
        // ...
    }
}

// Use Case 2: Temporary buffer pool
class BufferPool {
    private List<SoftReference<byte[]>> buffers = new ArrayList<>();

    public byte[] getBuffer(int size) {
        for (Iterator<SoftReference<byte[]>> i = buffers.iterator();
i.hasNext();) {
            SoftReference<byte[]> ref = i.next();
            byte[] buffer = ref.get();
            if (buffer != null && buffer.length >= size) {
                i.remove();
                return buffer;
            } else if (buffer == null) {
                i.remove();
            }
        }
        return new byte[size];
    }

    public void returnBuffer(byte[] buffer) {
        buffers.add(new SoftReference<>(buffer));
    }
}
```

Real-world use cases for phantom references:

1. Resource cleanup: In the example above, phantom references are used to perform cleanup actions after an object has been finalized but before it's collected. This is useful for managing non-memory resources that aren't automatically cleaned up by the garbage collector.
2. Detecting memory leaks: Phantom references can be used to track objects and detect potential memory leaks. If objects aren't being collected when expected, it could indicate a memory leak.
3. Monitoring object lifecycle: For debugging or profiling, phantom references can be used to precisely track when objects are being collected.
4. Custom memory management: In some advanced scenarios, phantom references can be used to implement custom memory management schemes, allowing precise control over when objects are removed from memory.

```
// Use Case 1: Resource cleanup
import java.lang.ref.PhantomReference;
import java.lang.ref.ReferenceQueue;
```

```
class ResourceCleaner {
    private static final ReferenceQueue<Object> QUEUE = new ReferenceQueue<>();

    static class CleanerThread extends Thread {
        @Override
        public void run() {
            while (true) {
                try {
                    PhantomResource res = (PhantomResource) QUEUE.remove();
                    res.cleanup();
                } catch (InterruptedException e) {
                    break;
                }
            }
        }
    }

    static class PhantomResource extends PhantomReference<Object> {
        private final Runnable cleanupAction;

        public PhantomResource(Object referent, Runnable cleanupAction) {
            super(referent, QUEUE);
            this.cleanupAction = cleanupAction;
        }

        public void cleanup() {
            cleanupAction.run();
        }
    }

    static {
        new CleanerThread().start();
    }

    public static void register(Object obj, Runnable cleanupAction) {
        new PhantomResource(obj, cleanupAction);
    }
}

// Use Case 2: Detecting memory leaks
class MemoryLeakDetector {
    private static final ReferenceQueue<Object> QUEUE = new ReferenceQueue<>();
    private static final Map<PhantomReference<Object>, String> REFERENCES = new
ConcurrentHashMap<>();

    public static void track(Object obj, String description) {
        PhantomReference<Object> ref = new PhantomReference<>(obj, QUEUE);
        REFERENCES.put(ref, description);
    }

    public static void checkForLeaks() {
        PhantomReference<Object> ref;
        while ((ref = (PhantomReference<Object>) QUEUE.poll()) != null) {
            String description = REFERENCES.remove(ref);
            System.out.println("Object collected: " + description);
        }
        if (!REFERENCES.isEmpty()) {
            System.out.println("Potential memory leaks:");
            for (String description : REFERENCES.values()) {
                System.out.println(description);
            }
        }
    }
}
```

Chapter 8: Native Memory Best Practices

Heap is the largest consumer of the application memory, but JVM will allocate and use a large amount of native memory. Non-heap memory is the heap memory. Native memory can also be allocated in applications (via JNI calls to malloc() and similar methods, or when using New I/O, or NIO).

Every time the JVM creates a thread, the OS allocates some native memory to hold that thread’ s stack, committing more memory to the process (until the thread exits, at least). Thread stacks, though, are fully allocated when they are created. From application memory allocation prospective both – heap and Native Memories collectively provide the memory footprint of the application.

Concept of revered memory Vs committed memory when heap is define as -Xms512m -Xmx2048m, where heap is started with 512 MB of memory, where OS committed to provide 2GB of memory (but not allocated upfront).

The flowing are the ways how one can minimized the memory footprint

- Heap: 50-60% of memory footprint is consumed by the heap. Controlling heap size will automatically control memory footprint.
- Thread Stack: another area where memory foot print can be limited is , thread stack – limiting the amount of memory consumed by the thread stack will resulted in controlling overall memory footprint.
- Code Cache – Code cache uses Native memory to hold compile code.
- Direct byte buffer – Developer directly or indirectly can allocated Native memory – the common example are used for filesystem and socket programming where buffer are used. The use of direct byte buffer should be avoided as much as possible. From tuning prospective - by setting the -XX:MaxDirectMemorySize=*N* flag, one can define how much max allocation of direct memory is allowed, since Java 7 onwards the default value is set to 0 (no limit).

Native Memory Tracking: Introduced in Java 8, NMT provides insights into how the JVM allocates native memory. It can be enabled with the option -XX:NativeMemoryTracking=summary/detail. The summary mode is typically sufficient for most analyses, allowing you to see how much memory the JVM has committed and its usage.

Use the command jcmd process_id VM.native_memory summary to get real-time native memory information. This helps in identifying potential performance issues

Total memory commitment: The total committed size of the process is the actual amount of physical memory that the process will consume. This is close to the RSS (or working set) of the application, but those OS-provided measurements do not include any memory that has been committed but paged out of the process.

Individual memory commitment: When it is time to tune maximum values—of the heap, the code cache, and the metaspace—it is helpful to know how much of that memory the JVM is using.

JVM Tuning for the Operating System:

Large Pages

Utilizing large pages can significantly enhance memory management efficiency in Java applications, especially in environments where large memory allocations are common. Proper configuration and understanding of the operating system's capabilities are essential for leveraging this feature effectively.

Linux kernels starting from version 2.6.32 support transparent huge pages, which can be enabled for Java applications. The configuration involves setting the kernel parameter to allow huge pages to be used without explicit advisories from the JVM. This can be done by modifying the /sys/kernel/mm/transparent_hugepage/enabled file.

To configure huge pages on Linux, the following steps are typically followed:

1. Determine the supported huge page sizes (commonly 2 MB).
2. Calculate the number of huge pages needed based on the JVM heap size.
3. Write the required number of huge pages to the operating system to apply the changes immediately.

   ```
   # echo 2200 > /proc/sys/vm/nr_hugepages
   ```
4. Save the configuration for persistence across reboots

   ```
   sys.nr_hugepages=2200
   ```

On many versions of Linux, the amount of huge page memory that a user can allocate is limited. Edit the /etc/security/limits.conf file and add memlock entries for the user running your JVMs

In Linux, the UseLargePages flag is not enabled by default, and the operating system must be configured to support large pages. If this flag is enabled on a system that does not support large pages, the JVM will use regular pages without warning.

Windows large pages can enhance memory management for Java applications, but they can only be enabled on server-based Windows versions. Note that if the operating system does not support large pages (like some home versions), the JVM will automatically set the UseLargePages flag to false without printing an error.

Compressed OOPs

In the context of a 64-bit Java Virtual Machine (JVM), Compressed Ordinary Object Pointers (Compressed OOPs) play a crucial role in optimizing memory usage and enhancing application performance. This feature addresses specific challenges inherent to 64-bit architectures and leverages various strategies to make Java applications more efficient.

In a 64-bit JVM, these pointers are naturally 64 bits (8 bytes) in size, allowing the JVM to address a vast memory space. Not all application need this much memory.

The following are the challenges faced by a 64-bit VMS (with 64-bit pointer).

- **Increased Memory Footprint:** Each object reference consumes 8 bytes, which can significantly increase the overall memory usage, especially in applications with millions of objects.
- **Cache Efficiency:** Larger pointers mean that fewer object references fit into CPU caches, potentially leading to increased cache misses and degraded performance.
- **Memory Bandwidth:** More memory consumption can lead to higher memory bandwidth usage, affecting the application's speed.

Compressed OOPs are a JVM optimization that reduces the size of object pointers from 64 bits to 32 bits. This compression is feasible under the assumption that most Java applications do not require addressing the entire 64-bit address space. Typically, the JVM limits the heap size to a subset of the 64-bit address range, allowing pointers to be represented as 32-bit offsets relative to a base address.

How Compress OOPs works

1. **Base Address:** The JVM maintains a base address for the heap. All object references are stored as 32-bit offsets from this base.
2. **Pointer Arithmetic:** When accessing an object, the JVM calculates the actual 64-bit address by adding the 32-bit offset to the base address.
3. **Alignment:** Objects are usually aligned on specific boundaries (e.g., 8-byte alignment), allowing the JVM to shift pointer values, effectively storing smaller offsets.

4. **Dynamic Adjustment:** The JVM can switch between compressed and uncompressed OOPs based on the heap size and other runtime factors.

By default, many modern JVM distributions enable Compressed OOPs when running in a 64-bit environment, especially if the heap size is below a certain threshold (commonly 32 GB). -XX:+UseCompressedOops to enable Compress OOPs, +XX:+UseCompressedOops disable Compress OOPs.

Trade off and Consideration while enabling Compress OOPS

Heap Size Limitations: Compressed OOPs are typically effective for heap sizes up to around 32 GB. Beyond this, the 32-bit offset may not suffice to address all memory, potentially necessitating larger object references.

Performance Overhead:

Pointer Decompression: Accessing objects requires additional calculations to reconstruct the full 64-bit address from the 32-bit offset, introducing minor computational overhead.

Alignment Constraints: Objects need to be aligned appropriately (e.g., 8-byte boundaries) to facilitate pointer compression, which might impose certain constraints on memory allocation.

Application utilizing large memory heap: Certain applications that already utilize very large heaps or have specific memory access patterns might not benefit as much from Compressed OOPs.

JNI uses malloc() in several scenarios:

1. **Temporary Buffer Allocation**:
   - When native code needs to process large amounts of data
   - For intermediate calculations before returning results to Java
   - When creating temporary storage for string manipulations
2. **Data Structure Creation**:
   - When implementing complex algorithms that require dynamic data structures
   - For creating linked lists, trees, or other custom data structures in native code
3. **Key Times malloc() is Used**:
   - During data conversion between Java and native formats
   - When implementing algorithms that require dynamic memory allocation
   - For caching or buffering operations in native code

Important Considerations:

1. **Memory Management**:
   - Always free allocated memory using free()
   - Memory leaks in native code won't be handled by Java's garbage collector
   - Use proper error handling for allocation failures
2. **JNI Best Practices**:
   - Minimize the time memory is held in native code
   - Use Java's memory management when possible
   - Always check for NULL after malloc()
   - Clean up resources in case of errors
3. **Common Pitfalls**:
   - Forgetting to free allocated memory
   - Not handling allocation failures
   - Memory leaks in error paths
   - Incorrect size calculations leading to buffer overflows

Chapter 9: Threading and Synchornization Performance

Since very stage of Java, Java been a muti-threaded application development language due to its ability to work on parrel tasks. The ability of multi-tasking largely depends on the number of CPU Core. Sine the CPU Cores are limited it's important to manage the threads that are getting executed on these CPU Cores. The following are the thread configuration parameters that Java uses to manages its parallel/multi-threaded workload.

- Setting of the Maximum number of threads in thread pool
    - The optimal number of threads depends on a) type of workload b) hardware type (number of CPU)
- Setting of the Minimum number of threads in thread pool
    - The minimum number of threads helps preventing creation of excessive threads.
    - If the system is not able to handle its workload due to thread constrained – then setting minimum number of threads is not helpful – better one should allow creating all the (required) threads all at once.
- Thread pool task size
    - Task of the thread poll held in a queue. When a thread is available, it pulls a task from the queue and completes it.
    - If the task queue depth is too large then tasks may have to wait much longer time to have thread allocated to perform the task. ThreadPoolExecutor – can be configure to manages it effectively.
- Sizing of the ThreadPoolExecutor
    - The theadPoolExecutor start with the minimum number of threads in the threadpool once they are exhausted then new threads are getting created.
    - The behavior of the threadPoolExecutor varies depending upon the queuType
        - **Synchronous Queue** – new task in the queue starts a new thread until all threads are exhausted; once the threads are exhausted new task are added to the queue. After the queue is full any new tasks are rejected. Once a thread completes its tasks , it pick up the next task from the queue.
        - **Unbounded Queue** – No task is rejected since the queue depth is unlimited – In this case max number of threads specified by the CPU, ignoring maximum thread pool size.
        - **Bounded** – Once a minimum of number threads are engaged by the threadPoolExecutor, any new task that comes gets added to the queue until queue is full, after the queue is full any new task that comes a new thread is created and first task for the queue is assigned to that new thread; the vacant space on the queue is then be allocated to the new task. This repeats until max thread pool count is reached, afterwards, any new tasks come it will be rejected.

ForkJoinPool It was introduce in Java 7 onwards , where threadPoolExecutor work on the complete task; whereas in care of ForkJoinPool a given task is broken down into multiple simpler task which then performed by the ForkJoinPool using divide & conquturned er logic. Once all the sub-part of the task is completed, they are joined and returned.

Automatic Parallelization

- ForkJoinPool Introduction: Java 7 introduced the ForkJoinPool class, which is designed for parallel processing using divide-and-conquer algorithms. It allows tasks to be broken down into smaller subsets that can be processed concurrently.
- Common Pool in Java 8: Java 8 enhances the ForkJoinPool by introducing a common pool that can be utilized by any ForkJoinTask not specifically submitted to a designated pool. This common pool is automatically sized based on the number of processors available on the machine.
- Parallel Class Loading: In complex environments, parallelizing class loading can enhance application startup times. However, this can be hampered by disk I/O bottlenecks when multiple classloaders access the same disk simultaneously. 13
- Limitations of Parallelization: While parallelization can speed up processing, it is important to note that Java does not automatically parallelize all code. Developers need to structure tasks appropriately to take advantage of parallel processing capabilities.

Thread Synchornization

In case of the synchronization the speedup is calculated by the following Amdahl's law

$$Speedup = \frac{1}{(1 - P) + \frac{P}{N}}$$

Where P is the amount of the code that run in parallel, N is the number of threads. If P reduces, even with the increasing value of N the speedup will negatively impacted- hence we need to reduce the code within synchronization block.

False Sharing

False sharing is a performance issue that occurs in multi-threaded applications when different CPU cores write to variables that are located in the same CPU cache line, even though these variables are logically independent. Modern CPUs read and write memory in fixed-size blocks called cache lines (typically 64 bytes). When a CPU core needs to modify a variable, it loads the entire cache line containing that variable

When two threads on different CPU cores update different variables that happen to be in the same cache line. Each core must invalidate the other core's cache line copy, forcing a reload. This creates unnecessary cache coherence traffic and performance degradation

Solution to avoid/detect false sharing.

- Java 8 introduced @sun.misc.Contended annotation, to detect the false sharing.
- Java 15+ has improved object layout algorithms to help prevent false sharing.
- Using proper padding as shown in the example.
- Using data structures designed to avoid false sharing.
- Keep related data together
- Use appropriate padding when necessary
- Consider using thread-local storage where appropriate
- Use concurrent collections designed to minimize false sharing
- Profile your application to identify potential false sharing hotspots

JVM Thread Tuning

Thread Stack Sizes: The JVM allows adjustment of thread stack sizes to optimize memory usage, especially in environments where memory is limited. Each thread has a native stack for storing call stack information.

Biased Locking: By default, the JVM uses biased locking, which gives priority to the thread that most recently accessed a lock. This can improve performance by increasing cache hits, but it may also introduce overhead due to the required bookkeeping. Disabling biased locking can benefit applications that use thread pools.

Lock Spinning: The JVM provides options for handling synchronized locks that are contended. A blocked thread can either enter a busy loop (lock spinning) or be placed in a queue. Lock spinning can lead to performance issues if not managed properly.

Thread Priorities: Java threads can be assigned priorities, which serve as hints to the operating system about their importance. However, the actual scheduling of threads is influenced by various factors, including how long a thread has been idle. This ensures that no thread is starved of CPU time, regardless of its priority.

Java Thread Monitoring

Thread State Monitoring: Tools like jstack and jcmd provide insights into the state of each thread in the JVM, indicating whether threads are running, waiting for a lock, or waiting for I/O operations. This information is crucial for diagnosing performance issues in applications.

Detecting Blocked Threads: Successive thread dumps can reveal significant contention on locks if many threads are blocked. If threads are blocked waiting for I/O, it may indicate that the I/O operations themselves need tuning, such as optimizing SQL queries or database performance.

Java Flight Recorder (JFR): JFR is a powerful tool for monitoring thread activity at a low level. It captures events related to thread blocking, such as when threads are waiting to acquire a monitor or perform I/O operations. These events can be visualized in Java Mission Control, allowing for effective analysis of thread performance.

Real-Time Monitoring: Interactive tools like jconsole allow for real-time observation of thread activity. Users can see how the number of threads fluctuates during program execution, providing immediate feedback on thread usage and performance.

Diagnosing Performance Issues: Diagnosing blocked threads can be challenging but is essential, especially in multi-CPU systems. Profilers can help visualize thread execution timelines, making it easier to identify when threads are blocked