CLASSIFICATION OF JAVA PERFORMANCE TUNING METHODS

Manzar Choudhary
Principal Software Engineer
EMC, India Center of Excellence (iCOE)
manzar.choudhary@emc.com

Narendra Sharma
Engineering Manager
EMC, India Center of Excellence (iCOE)
narendra.sharma@emc.com

Mudit Verma
Software Engineering
EMC, India Center of Excellence (iCOE)
mudit.verma@emc.com
Table of Contents
Overview................................................................................................................................. 3
Characteristics of the Tuning Techniques ............................................................................... 3
Application Tuning Techniques ............................................................................................... 4
Application Level Techniques ................................................................................................. 7
Third Party Component Level ............................................................................................... 10
Conclusion ............................................................................................................................... 11
References ............................................................................................................................... 11

Disclaimer: The views, processes or methodologies published in this article are those of the authors. They do not necessarily reflect EMC Corporation’s views, processes or methodologies.
Overview

Achieving a target performance sometimes becomes a big challenge for an application. In general, once the application coding is done and desired functionality is achieved, the performance aspect comes into the picture. After achieving the desired functionality, people start targeting for performance numbers. At that point the need for application tuning arises.

Tuning an application requires high expertise. Sometimes by just looking at the application design or Java Virtual Machine (JVM) parameter, an expert can identify the bottlenecks in the application and can give tuning advice, but sometimes detailed and extensive diagnosis of the application call flow needs to be done in order to issue any tuning advice. Different applications have different types of resource usage, e.g. some applications rely heavily on database. On the other hand, web service applications may be making tons of network calls. Hence, each application requires different tuning techniques.

There are lots of well-known techniques for tuning a Java application. Some of the well-known techniques are tuning the Garbage Collector (GC), starting you application with correct and appropriate JVM setting, and analysis and optimization of application codebase. However finding out the suitable technique for the application is not easy. Diagnosis of a bottleneck in the application requires different approaches. In complex or multi-tier applications, a proper attack strategy needs to be developed in order to identify performance bottlenecks. For example sometimes tuning JDBC configuration improves the performance. However, in order to identify JDBC bottleneck, we need to have the correct analysis approach. This article lists and explains various techniques that are used in the course of application tuning. An attempt is made to identify the factors, which can help in selecting a suitable performance tuning technique for an application. Further, an effort is made in this article to categorize all the tuning techniques based on different classification parameters and characteristics. Different classification parameters are used in this article.

Characteristics of the Tuning Techniques

Nature of technique – Tuning techniques can be classified into two major groups – Invasive techniques and Non-Invasive techniques. Invasive techniques require getting down at the lower level in the technology stack. Invasive techniques require more time and efforts as it involves getting down to finer and intrinsic details of the application. Further tuning exercise will be more efficient and time saving if developer and architects of the application are involved, while applying invasive techniques for application tuning. Examples of invasive techniques are code level optimization, design level changes, JDBC tuning, SQL Queries optimizations, etc.

Non-Invasive techniques are applied more in terms of configurations without getting into the application level. The Tuning Engineer need not know about the application level details, Instead he can focus more on application environment details. Examples of Non-Invasive techniques are GC tuning, configuring JVM settings, tuning application container in which application is running, etc.

Complexity of technique – This parameter measures the ease of implementation of tuning technique - how easily can a technique be applied. For example, configuring heap memory for
JVM is easy, however, tuning GC for JVM can be quite a complex task, as full analysis of GC logs is required for tuning GC. Implementation of design level changes for the application can again be a complex task, as it involves lots of effort. Low complexity techniques can be applied by amateurs or novice developers, and does not requires experts for tuning.

**Ease of Identification** – This parameter measures the ease of investigation and identification of bottlenecks. For instance, if heap memory is causing a problem it can easily identified by looking into error stack. However, if logging statement are taking time, investigation efforts are more to conclude that. In tuning exercise, most time is spent identifying the bottlenecks in the application. Once bottlenecks are identified, there are well-known tried and tested approaches to solve the bottleneck.

**Area of Improvement** - In general, we do application performance tuning for two things - high responsiveness (response time), or high performance (scalability). Sometimes as per the type of bottleneck, we can identify the tuning technique. For instance, if Optimizing the SQL Query results in faster application response, a result set will be fetched faster. However, if we can increase the heap size, we can accommodate more serve request s, thus scaling the application horizontally.

**Return Of investment (ROI) of Technique** – This parameter tells what will be the benefits of technique, i.e. how much performance can be increased by applying the techniques. Although it is hard to quantify the ROI of the techniques, sometimes we know that by using some particular technique we can’t leverage more than certain amount of benefits. For instance, if there is some design level bottlenecks, tuning the GC cannot leverage the performance beyond a certain level.

**Application Tuning Techniques**

All performance tuning techniques can broadly categorized into 3 categories.

1. Java Level
2. Application Level
3. Third Party Component Level

**Java Level Techniques**

**Characteristics**

These techniques are applied at JVM Level. In general, they deal in configuring various parameters that are exposed by JVM implementation. These techniques are non - invasive in nature. Some of them are pretty trivial to use and do not require any special expertise hence having low complexity and ease of identifications. By looking at the initial JVM configuration (startup options) and nature of performance bottleneck, these configuration parameters can be set appropriately. Both response time and scalability requirements can be achieved by tweaking these parameters. Java level Techniques should be the first choice if some application needs to be tweaked for performance.
Tuning Techniques

Choosing Correct VM
Java comes in different flavors with lots of variations in Virtual Machines. Some Virtual Machines are good for development, but running enterprise level applications on them might result in performance loss. JVMs can be a single use JVM or continuous JVMs. Single Use JVMs are not suitable for production environment, as a new JVM is initialized for each java program invocation. In general, continuous JVMs are better in term of CPU usage and throughput. However, if our Java program is written keeping Single Use JVM in mind, then we should use the same.

Tuning JVM OPTIONS – Ergonomic settings

Famous –Xms and –Xmx
JVM provides two options to set initial and maximum heap space. These options are hard to predict at the start, requires trial and error techniques and some time looking at GC output can help in taking decisions, but easy to do.

When to use
On Getting “java.lang.OutOfMemoryError”: Memory option configuration need to be considered when we get “java.lang.OutOfMemoryError” in the application logs. Sometimes it is hard to predict memory requirement of the application initially, and we configure less limit for maximum heap memory (Xmx) size. In such scenarios, we get Out Of Memory Exception. We need to keep on increasing the Xmx value until we stop getting the error. However sometimes, this error may also be the result of Memory leaks in the application.

Application is taking longer than expected: There can be many reasons due to which application can be taking too long, however one reason can be wrong configuration of Xms. For instance, if Xms value is too high, close to physical memory (RAM) than pagination of memory will start happening, resulting in slowness of application

When GC pauses are frequent: From Java Garbage Collector (GC) logs we can figure out if there are lots of GC pauses. It means that there is not enough free heap space and JVM is starting GC too frequently. When heap space is small, it will fill up fast, JVM kicks the GC, and hence there will be a pause. Heap size needs to be increased so that GC kicks in after some time.

When GC pauses are long: Again this can be seen in GC logs. This happens if heap space is too much, it takes longer to fill up the heap space hence GC will kick in after a long time, but it will take longer for GC to complete the cleaning as heap space is too much. Therefore, we should use optimum heap space size.

When applications fail to start: This might be because application is not having enough memory. We need to increase the memory.
How to Use

Run JVM process with -verbose:gc option in order to get the GC logs, and see how much memory is being used over the period of time. Then it’s just a matter of setting correct value Xmx and Xms parameter. In general, unless you have problems, try giving as much as possible memory to the JVM.

Garbage Collection Tuning

These options must be tweaked only after seeing the Garbage Collection reports .GC logging should be enabled and GC reports should also be analyzed. J2SE 1.4 provides choice of four different Garbage collector options, and if nothing is specified Serial Garbage Collector is chosen. From Version 5, depending upon the class of machine, GC is chosen.

When to Use

Most of the time, GC is not required: For smaller and simple applications choice of GC does not matter, as there won’t be a noticeable performance difference. For big multi-threaded systems which run on lots of CPUs and require a lot of memory, it makes more sense to consider tuning Garbage collector policy.

However, there are some other clear indicators too. If you see that throughput of your application decreases drastically when GC is running (which happens in general but not drastically), it means something can be done with GC parameters to increase the throughput. In general, Garbage collector takes around 5-20% of total execution time, which means that average time between GC runs should be five to six times of the average garbage collection run time. Memory leaks also contribute to Garbage collection bottle necks.

“XX:+PrintTenuringDistribution” can give the detailed stats which will help in deciding the size of various generations.

How to Use

GC Run frequency can be controlled by setting the sizes of different generations appropriately. Parameters like “NewSize”, “MaxNewSize” and “NewRatio” are used to control sizes of the generations. If application involves lots of short lived object creation, young generation size should be more, so that GC doesn’t have to run too frequently to clean up the young generation size. In contrast, if we have more long lived objects in application, then doing the opposite makes more sense, so that tenured generation can hold more. Sooner the memory is copied to older generation, the better the GC performance is. It is recommended to first decide how much heap memory you are going to assign to JVM, then sub-divide it into various generations accordingly.

Nature of GC Run can also help in determining the Size of generations. If Full GC is running too frequently, it means young generation size is more as compared to old generation or old generation is filling too fast. Tweaking “NewRatio” JVM parameter helps in reducing the Full GC cycles. In Java 5 two new parameter are introduced - “MaxGCPauseMillis” and “GCTimeRatio” which can control the frequency and GC runtime behavior. “GCTimeRatio” parameter control
the ratio of Garbage collection time to application time, while “MaxGCPauseMillis” hints the
garbage collector of - maximum time that Garbage collector can run.

Choosing a Garbage Collector Algorithm

Java provides different types of GC algorithm implementations. It also chooses to GC
automatically for us depending upon the machine type. For Server class machine (i.e. at least 2
CPU Cores and 2 GB of memory) parallel GC is used. For Non server class, machine serial GC
is being used. Serial Garbage collectors are designed for applications which have small data
sets and default options work well with it. Throughput garbage collectors works well with
medium to large data sets. The standard and most common collector is Serial collector.

When to use

Well changing garbage collector algorithms make more sense when all the above mentioned
techniques were used, and still garbage collection bottlenecks are being faced. In general, GC
algorithms are changed for applications which are big in nature; involve a lot of heap memory,
and runs in multi-threaded, multi-processor environment. Sometimes having the correct
Garbage collector helps in reducing the Garbage Collector pause time – which becomes major
requirement for mission critical applications.

How to use

For small size applications, Serial Garbage collector suffices. It can be set using
XX:UseSerialGC. If there is multi-processor environment and throughput of the application is
critical – espically during peak load, it is recommended to use parallel collectors. Parallel
collectors are also known as throughput collectors and runs minor collection in parallel which
gives significant boost to the overall GC run time. It can be set using the following parameters
XX:+UseParallelGC. The number of Garbage collector threads can be controlled by using
additional parameter -XX:ParallelGCThreads. Since minor collections are running in parallel,
frequency of major collections decreases and overall GC pauses are reduced. For applications
where response time is critical, concurrent garbage collectors should be used. Concurrent
collectors are famous for low GC pauses. The concurrent collector is enabled using following
option XX: +UseConcMarkSweepGC.

Application Level Techniques

Characteristics

Application level techniques are invasive in nature and have higher complexity in terms of
identification and implementation. It involves scanning and analyzing the existing application
design and code base. However, ROI for these techniques are high. Modifying the application
design in order to gain performance gives tremendous flexibility. There are very few limitations
with this type of technique. However, since effort requirement are high, this approach should be
the last one to consider.
Tuning Techniques

Code Level Optimization

This section describes the code level changes that can be made to increase performance. Optimizing code for performance can contribute significantly in the performance of the application. Small code changes at different places throughout the application code base can aggregate to significant performance boost. Actually Java does not restrict us from using non-efficient coding practices. Different Coders have different coding style, same routine can be written in different ways, yet producing the correct result. However, choice of basic constructs – like looping, collections, object creation, string usage etc. in the routine matters.

When to use

In general, coding best practices should be adopted in the initial phase of the application development. It is much easier to incorporate the coding practices while developing the application. Further, it becomes more difficult and cumbersome for the person to tune the application at code level if he has not written the code base himself.

Memory leak is the major driver for code level analysis and optimizations. If there are memory leaks in the application, Out of Memory error will occur and cannot be fixed with other tuning techniques.

When other tuning techniques fail to achieve the desired throughput or response time for the application, we need to consider analyzing the code base and try optimizing it.

How to do

There is a very big list of code level optimization techniques available today. However, we will consider only major ones here.

Avoid unnecessary object creation: Creating an object requires CPU resources and time, further garbage collecting will involve more CPU cycles. If something can be achieved without the use of object creation, go for it. Creation pool of frequently used objects will help them recycle, and saves unnecessary overhead of object creation. If possible try to use Singleton design pattern, so that object creation can be avoided. Using Static methods wherever required can help in performance boost. Using the final modifiers also helps. Creating temporary object in frequently called code should be avoided. Sometimes early initialization of the object helps, creating the long lived object during the bootstrapping phase early in the application helps increase performance.

Strings: Lots of optimization can be done around String, as this object is most used object in almost all applications. String pool provided by JVM can be used judiciously. However, there are some concerns. Java 6 used permgen to store String pool, while Java 7 onwards is stored in heap and subject to garbage collected. Best practices like use of String Buffer in place of String concatenation recommended in Java documentation should be honored.
Other Best practices like Loop optimization, Use Of Primitives in place of objects, Auto boxing avoidance, Use of native methods like System.arraycopy(), and choosing a correct collection from Java collection framework can be adopted in Java coding.

**Design Level Tuning**

Apart from coding constructs, design approach plays a significant role in application performance. Application Design is something which is very hard to change, once application is done. Good architects take care of performance from the beginning, and they leave the scope plugging in the better implementation if targeted performance numbers are not met.

**When to use**

When performance numbers are way below the expected, the application design review can be done. Since doing changes in the application design requires significant effort, this exercise should be done only if returns are high. Further, sometimes it’s not possible to make any design changes because of the way in which application code are written. It is suggested that first other tuning techniques should be applied and if still there is a need of increasing the performance numbers, tuning should be done.

**How to do**

In order to do design level changes, first of all design review needs to be done. The task becomes more difficult if the original architect of the application is not involved. All aspects of the design should be minutely analyzed and then different approaches should be considered. Some general principles are listed below

Build Cache – Where possible, try to use cache. As we know memory is cheap nowadays - we can allocate more memory to our application, hence caches should be used generously. For example, if application configuration parameters are being used quite often, it makes more sense to fetch it from database and store it in cache, and whenever any changes happen in the config, cache should be refreshed.

Introduce parallelism. If design permits, multi-threading should be introduced as a design optimization technique. With hyper threading and multi-core processors, multi-threaded applications run with high performance.

Reduce number of Layers. In a multi-tier environment we should re-evaluate the need of each layer. Sometime architects tend to introduce a layer for the sake of future extensibility, but it might significantly reduce performance. The idea is to bring data as close as possible to the application layer, so that commute time between the layers can be avoided.

Avoid Synchronization – Unnecessary synchronization across the layers affect the performance. Further, sometimes locking on the shared resources decreases the performance.

Extensible implementations – Interfaces should be designed in such a way that in future if faster and more efficient implementation of some component comes, it can be seamlessly integrated in our application.
Choice of protocols: Different protocols have different advantages and overheads. Sometimes we may face issues with protocols that are being used by the application. We might consider changing to more efficient and faster protocols.

Third Party Component Level

Characteristics

These are more or less non-invasive in nature, In general their scope is very limited and bottleneck identification is not very difficult. Application and third party component integration points are identified and analysis is restricted to that scope. This technique becomes invasive in nature if integration with third party component is very tightly coupled and we need to change the third party component. ROI can be very high if third party components are heavily used across the application.

Majority of applications these days uses lot of third party frameworks and libraries. With so many frameworks available today, choosing the right one is often confusing and difficult task. Choice of framework often depends upon popularity and ease of integration with the application and performance plays fewer roles. Some importance should be given to the performance part of the third party frameworks. Popular examples are Log4J, Hibernate, Spring, JDBC etc.

When to use

It requires detailed analysis to identify the bottlenecks with third part components. During the performance evaluation phase if we observe that some particular module or third party component is acting as a bottleneck, we need to need to shift our focus on integration aspect of our application with the third party framework. For instance, if we see that database fetches or writes are slow we need to consider the Object Relation Mapping (ORM) framework that is being used or JDBC drivers that are being used. Key idea is to identify the external integration points, and if that is acting as a bottleneck, we should consider tuning it.

How to Use

In general, all third party frameworks come with some form of documentation. It is recommended to get the full knowledge of third party framework. Identify and explore all the different integration options and configuration parameters that third party frameworks come with. Use a Proof of Concept (POC) with that framework, trying out all the different configurations and identify the best set of tuning configurations for your needs. For instance, DB vendor can provide different type of drivers depending upon the operating environment and needs. It is recommended that all drivers’ details should be carefully investigated so that best can be picked out of it.
Conclusion

All Java performance tuning techniques require some level of expertise; depending on the experience level these techniques can used. For example, configuring the heap memory size is easier but choosing the correct strategy for Garbage collection requires some level of expertise. There is well defined principle or strategy involved in tuning any application for performance. It is to identify the most obvious cause, choose the quickest and easiest approach to fix it, and then test the performance level and if level is not achieved, start again with the next obvious case. The above strategy goes well for Java also. Mostly performance application tuning techniques is mix of experience and intuitions. This article gives the basic idea of Java Application performance tuning to the developer and guides them on how to approach the tuning issue.

References


EMC believes the information in this publication is accurate as of its publication date. The information is subject to change without notice.

THE INFORMATION IN THIS PUBLICATION IS PROVIDED “AS IS.” EMC CORPORATION MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WITH RESPECT TO THE INFORMATION IN THIS PUBLICATION, AND SPECIFICALLY DISCLAIMS IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Use, copying, and distribution of any EMC software described in this publication requires an applicable software license.