Masaryk University
Faculty of Informatics

Optimization of JVM settings for application performance

Master's Thesis

Josef Pavelec

Brno, Spring 2017

Masaryk University
Faculty of Informatics

Optimization of JVM settings for application performance

Master's Thesis

Josef Pavelec

Brno, Spring 2017

Replace this page with a copy of the official signed thesis assignment and the copy of the Statement of an Author.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Josef Pavelec

Advisor: RNDr. Andriy Stetsko, PhD.

Acknowledgement

Will be written in the end.

Abstract

Will be written in the end.

Keywords

JVM settings, Optimization

Contents

1 Introduction

Will be written in the end.

2 Java Virtual Machine

Java Virtual Machine (JVM) is an abstract computing machine. JVM can not process any program written in Java language (stored in `java` source file) but it executes only the program called *bytecode* which is stored in a `class` file. A Java `class` file is produced from `java` source file by Java compiler (`javac program.java` command). JVM, like a real computing machine, has its own instruction set for processing *bytecode*. For running compiled Java program, it's necessary to have only an implementation of JVM for a given platform. Described approach offers the ability for applications to be developed in a platform-independent manner, and it can be shortened by Sun Microsystems slogan: "*Write once, run anywhere*". [9]

In the context of the JVM it should distinguish three terms:

- "**specification** is a document that formally describes what is required of a JVM implementation.
- **implementation** is a computer program that meets the requirements of the JVM specification.
- **instance** is an implementation running in a process that executes a computer program compiled into Java bytecode." [10]

For the correct implementation of Java Virtual Machine you only need to be able to read `class` file and perform the specified operations. The Java Virtual Machine specification doesn't contains any requirements or constraints for implementation of JVM. For this reason, the memory layout of run-time data areas, the garbage-collection algorithm used, any internal optimization of the JVM instruction (e.g. translating them into native code) and other matters connected with implementation of JVM are left to the implementor. Consequently, the JVM specification doesn't describe the possible ways how to adjust a JVM behaviour to achieve a better performance of a running application. [9]

There are many implementations of JVM[1]. This chapter of the thesis deals with the Java HotSpot Virtual Machine respective to Java

1. An extensive list of JVM implementation is accessible on `https://en.wikipedia.org/wiki/List_of_Java_virtual_machines`

Development Kit 8 (JDK 8) which is the primary reference JVM implementation. This version is the latest because JDK 9 release is scheduled for July 2017[2].

2.1 Java HotSpot VM

The Java 8 HotSpot Virtual Machine implementation is maintained by Oracle corporation. It implements JVM 8 specification and seeks to achieve best results for executing *bytecode* in areas such as automatic memory management or compilation to native code.

Since the first version of JVM was interpreted, the statement "*Java is slow*" has persisted notwithstanding it's not true in these days. More information regarding this topic will be provided in next sections.

As mentioned earlier, the main purpose of JVM is executing *bytecode* on a specific platform which means translating *bytecode* to native code of CPU. Since interpreting of *bytecode* had appeared inefficient, an approach called *Just-In-Time* compilation (JIT) was introduced in Java 1.2 [5]. A JVM implementation with JIT compiler translates a program into native code on the fly. Native code is cached and reused without recompiled. The extent of native code optimization is limited in this case because translation should be fast.

Java HotSpot VM is appropriately named after approach it takes toward compiling the code. A small part of the code is executed frequently in common programs. Frequent code sections represent approximately 20 %[3] which are in accordance with the Pareto principle. These sections are called *hot spots*. The more the section of code is executed, the hotter section is said to be. The faster will be *hot spots* executed, the higher performance of an application will be. [11]

Oracle's HotSpot VM combines interpretation and translation. First, it interprets all code and concurrently collects information how often code is executed and additional data. After that JVM uses collected information for compilation with a high level of optimization. The more information about code JVM has, the more level of optimization it can achieve but for the price of the slow interpretation in the beginning.

2. `https://blogs.oracle.com/java/proposed-schedule-change-for-java-9`
3. `http://artiomg.blogspot.cz/2011/10/just-in-time-compiler-jit-in-hotspot.html`

Actually, depending on the aggressiveness and level of optimization there are two different compilers in HotSpot JVM – **client** and **server** (see sections 2.1.1 and 2.1.2).

The choice of which compiler to use is often the only one decision that is done during the compiler tuning. Even, choosing of compiler must be considered before JVM is installed because different JVM binaries contain different compilers. [11]

Another important role of any JVM implementations is automatic memory management known as **Garbage collections (GC)**. There are four different algorithms (called as Garbage Collectors) that provide this task in HotSpot:

- **serial collector**
- **parallel (throughput) collector**
- **concurrent (CMS) collector**
- **G1 collector**

Every of garbage collectors has quite a different performance characteristics and is suitable for a different category of an application and environment. [11]

2.1.1 Client compiler

The HotSpot VM's client compiler is designed for faster application startup, quick compilation and smaller memory footprint. This type of compiler is typically suitable for GUI application because there is a responsiveness without jitter desired. [8]

The name *client* compiler comes from the command-line argument `-client` used to select the compiler. When 64-bit version of the JDK is used, this option is ignored and *server* compiler (see section 2.1.2) will be used. Sometimes client compiler is called *compiler 1* or shortly *C1*[4]. [11] [2]

4. The short notation is used in list of advanced options. See section 2.2

2.1.2 Server compiler

The second HotSpot VM's compiler targets high throughput and peak performance. The server compiler uses the most powerful optimizations it can. The disadvantage of this behaviour is a fact, that time for getting an essential information for top optimization can be long. Concurrently, the overall memory footprint of server compiler is larger due to more data to handle. Native code produced by server compiler will be definitely more effective. Therefore advantages of high optimization of code depend on a run time of an application and the server compiler is suitable for long running applications. [8]

The `-server` option serves fo use the server compiler. The server compiler is also marked as *compiler 2* or *C2*.

A technique called *tiered compilation* combines a client compiler fast startup and a server compiler high optimization. With the tiered compilation, the code is first compiled by the client compiler, as it becomes hot, it is recompiled by the server compiler. The tiered compilation is available only for server compiler[5]. The tiered compilation was introduced in the JDK 7 and it becomes as default in the JDK 8. [11]

2.1.3 Garbage collectors

2.2 Setting options

There are many options how to adjust behaviour JVM. For example, we may enforce to use parallel collector or set maximum heap[6] size. Options are divided into several categories:

- "**Standard options** – are guaranteed to be supported by all implementations of the Java Virtual Machine (JVM). They are used for common actions, such as checking the version of the JRE, setting the class path, enabling verbose output, and so on.
- **Non-standard options** – are general purpose options that are specific to the Java HotSpot Virtual Machine, so they are not

5. If command `java -client -XX:+TieredCompilation` *programName* is used, `-XX:+TieredCompilation` option is quietly turned off. [11]
6. Heap is a memory area that JVM uses for residing the Java objects.

guaranteed to be supported by all JVM implementations, and are subject to change. These options start with `-X`.

- **Advanced options** – are not recommended for casual use. These are developer options used for tuning specific areas of the Java HotSpot Virtual Machine operation that often have specific system requirements and may require privileged access to system configuration parameters. They are also not guaranteed to be supported by all JVM implementations, and are subject to change. Advanced options start with `-XX`." [2]

For listing all standard options is used the command `java -?` and non-standard options `java -X`. A situation for listing advanced options becomes little complicated because we have to make a decision which supersets of advanced option we want to list. On the other hand, diagnostic options superset is not the point of interest in this thesis because these options not meant for VM tuning or for product mode. Similarly commercial features. Therefore, for listing advanced options which are relevant for this thesis command `java -XX:+PrintFlagsFinal -XX:+UnlockExperimentalVMOptions` was used. List of advanced options contains option name, type (e.g. bool, (u)intx, ccstr) and default value (see section 2.3) and category (e.g. product, experimental).

Generally, there are two types of advanced options: boolean options, and options that require a parameter. The boolean options use the syntax: `-XX:+OptionName` enables the option, and `-XX:-OptionName` disables the option. Options with parameter use syntax: `-XX:OptionName=value`, meaning to set the value of `OptionName` to `value`. [11]

2.3 Java ergonomics

This term covers the process by which the JVM provides a platform-dependent default selections. It aims to reduce the number of user defined used options and mainly improve the performance of running application. In addition, behaviour-based tuning dynamically tunes the sizes of the heap to achieve a smaller memory footprint and to meet a specified behaviour of the application. Depending on the environment features where the JVM runs can be defined the *Server-class*

Platform	Operating System	Default	Server-Class
i586	Linux	Client	Server
i586	Windows	Client	Client
SPARC (64-bit)	Solaris	Server	Server
AMD (64-bit)	Linux	Server	Server
AMD (64-bit)	Windows	Server	Server

Table 2.1: Determination the runtime compiler for different platforms. Taken from [3].

machine. A machine is considered the *Server-class* when meets next requirements:

- 2 or more physical processors
- 2 or more GB of physical memory

The table 2.1 shows what a compiler will be used for given platform. If JVM runs on the *Server-class* machine, it uses the *server* compiler with the exception of 32-bit platforms running a version of the Windows operating system.

The Default values which are shown in the list produced by `java -XX:+PrintFlagsFinal -XX:+UnlockExperimentalVMOptions` command are based on the Java ergonomics. On the *Server-class* machine, the following are selected by default:

- Initial heap size of 1/64 of physical memory up to 1 GB
- Maximum heap size of 1/4 of physical memory up to 1 GB
- Parallel (throughput) garbage collector

Unless the initial and maximum heap sizes are specified on the command line, these values will be used. [3]

3 Methodology of choosing important options

This chapter of thesis deals with a restriction of huge option set and determination options which have a considerable performance impact.

The set of options contains 815 options[1]. As described in the section 2.2, there are several types of options in JVM[2]. If JVM options had only boolean type, JVM would have 2^{815} different settings. Testing all options is technically possible, but practically not.

The methodology of choosing important options is based on a study how the options influence the JVM behaviour. Firstly, a principles of execution *bytecode* by JVM has been studied together with options which can adjust the execution. Based on the level influencing the JVM behaviour and availability a documentation it's possible to categorize JVM options into following groups:

- **Big Impact**
- **Small Impact**
- **Not Relevant**
- **Not Documented**

Some options require a current use of another option – typically garbage collectors have own specific options. For example, setting of `-XX:ConcGCThreads` option has the meaning only if the CMS or the G1 collector is used. Therefore, the first two groups of enumeration above are distributable into next two groups:

- *primitive* – options from this set doesn't require enabling of "parent" option.
- *complex* – for use some option it's necessary to enable a certain "parent" option.

---

1. The sum of 21 standard, 26 non-standard and 768 advanced (`java -XX:+PrintFlagsFinal -XX:+UnlockExperimentalVMOptions` command) options for Java HotSpot™ 64-Bit Server VM (build 25.111-b14, mixed mode).
2. For example, advanced option set contains 383 boolean, 184 integer and 173 unsigned integer options. (Remaining option types are double, string (ccstr), etc.)

A big disadvantage is a fact, that JVM doesn't point out this incompatibility. Hence, when a command `java -XX:+UseSerialGC -XX:ConcGCThreads=4` *programName* is launched, JVM executes *programName* without taking `-XX:ConcGCThreads=4` option into account. There is least at least an option `-XX:IgnoreUnrecognizedVMOptions` (default `false`) in JVM and when it is disabled syntax and type errors are highlighted. For example, command `java -XX:+UseConcMarkSweepGC -XX:ConcGCThreads=-4` *programName* end with message: "Improperly specified VM option 'ConcGCThreads=-4'".

In picture 3.1 there is a way how all options of JVM was splitted.

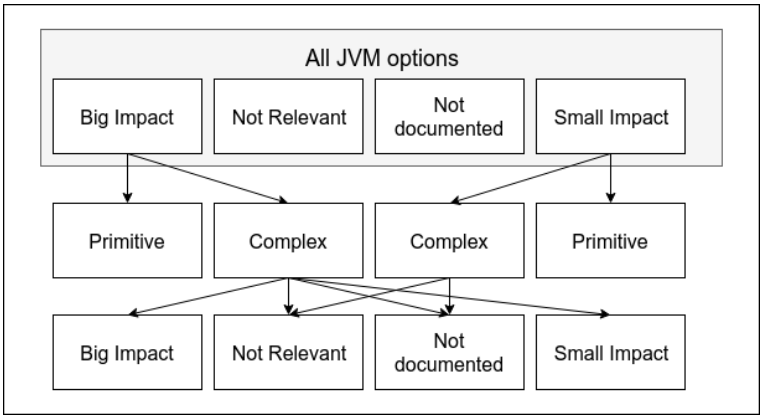

Figure 3.1: JVM options distribution into several categories with regard to performance impact and a documentation extent.

3.1 Big impact

These options were evaluated as most important for performance impact. The selection was based on the available Oracle's documentation ([2]) and a literature dealing with the Java performance ([11], [8]).

The big impact options, according to the previously mentioned distribution, are composed by two subsets – the *primitive* (see a complete list of *primitive* options in the appendix A.1) and the *complex* (see a complete list of *complex* options in the appendix A.2). These options are intended for further optimization (e.g. by using any tool described in section 4.2).

Among the most important *primitive* options are:

- `-client | -server` – choice of compiler. For more information about client compiler see section 2.1.1 and server compiler see section 2.1.2.

- `-d32 | -d64` – when a 32-bit operating system is used, then it's required using a 32-bit version of the JVM. When a 64-bit operating system is used, then it's possible to choose 32-bit or 64-bit version of the JVM. The 32-bit version will be faster and have a smaller footprint[3]. Because 32 bit memory references occupy smaller memory area and manipulating those references is less expensive. [11]

- `-Xmixed` – interprets all bytecode except for hot methods, which are compiled to native code. [2]

- `-Xcomp` – forces compilation to native code on the first invocation of method. [2]

- `-Xms<size>` – sets initial Java heap size. [2]

- `-Xmx<size>` – sets maximum Java heap size [2]. The JVM automatically sets the initial and maximum heap size (see section 2.3). Due to an application memory requirements, values initial and maximum heap size set by the JVM can be insufficient or uselessly great and their setting is a basic memory footprint tuning.

- `-XX:MaxGCPauseMillis` – sets a target for the maximum GC pause time in milliseconds. This value is not guaranteed (it's soft goal) but JVM makes its best effort to achieve it.

Several *complex* options with a big impact on the performance:

- `-XX:TieredCompilation` – is available only if `-server` option is enabled. It enables tiered compilation described in section 2.1.2.

3. For the heaps up to 3 GB. [11]

- `-XX:ParallelGCThreads` – the setting this option makes sense only if another GC than serial GC is used. By default, one thread for every CPU runs for GC, up to eight CPUs. For machines with more than eight CPUs `-XX:ParallelGCThreads` option equals $8 + ((N - 8) * 5/8)$, where N is a number of CPUs. When more JVM instances are running on the machine, it's a good idea to limit the total number of GC threads among all instances. GC threads are quite efficient and can consume 100 % of single CPU. [11]

- `-XX:ConcGCThreads` – if `-XX:UseConcMarkSweepGC` or `-XX:UseG1GC` option is enabled the use of `-XX:ConcGCThreads` is possible. This option sets the number of threads used for concurrent GC [2]. By default, value is based on the ParallelGCThreads option and is determined by equation: $ConcGCThreads = (3 + ParallelGCThreads)/4$. The calculation is using integer arithmetic. [11]

3.2 Small impact

The small impact set contains the JVM options that are important also. Although they don't influence a JVM behaviour in general, their use could have the big impact of performance in certain cases. For example, use `-XX:OptimizeStringConcat` option should be useful when an application merges strings extensively. In the other consideration, when an optimization aims to minimize the memory footprint, the `-XX:MaxHeapFreeRatio`[4] option significantly influences a success of optimization. Therefore, the several small impact options can be temporary moved to the big impact set to achieve better performance of given application and/or metric. Selected parameters which meets the definition of a small impact, for example:

- `-XX:AggressiveOpts` – enables the use of aggressive performance optimization features, notably for compilation. [2]. In Java 7, enabling this option means that different implementation of some

4. "Sets the maximum allowed percentage of free heap space (0 to 100) after a GC event. If free heap space expands above this value, then the heap will be shrunk." [2]

classes will be used[5]. Functionality these classes is the same but they have more efficient implementations. Since Java 8, there are not alternate implementations in the JVM[6].

When `-XX:AggressiveOpts` option is enabled default values of options `AutoBoxCacheMax` and `BiasedLockingStartupDelay` are changed also because of achievement better optimization.

- `-XX:GCTimeLimit` –

3.3 Not relevant

Essentially, there are options that don't have any influence on performance. They are particularly "printers" – options which only print some diagnostic information (e.g. `-XX:PrintGCDateStamps`) or commercial features.

3.4 Not documented

Represents the biggest set of options with no quality documentation. As described in the beginning of this chapter, all options testing is hardly feasible. This thesis doesn't deal with it.

5. For example, `java.math.BigDecimal`, `java.math.BigInteger`, `java.util.HashMap`, `java.util.TreeMap`.
6. Either more efficient implementations have been incorporated into the base JDK classes, or the base JDK classes have been improved in other ways. [11]

4 Tools

4.1 Analytic tools

4.1.1 JConsole

A JConsole is a graphical tools that allows monitor and manage Java applications on a local or remote machine. It is contained in JDK since Java 5 because in this version JMX[1] technology was introduced and JConsole is a JMX-compliant tool.[4]

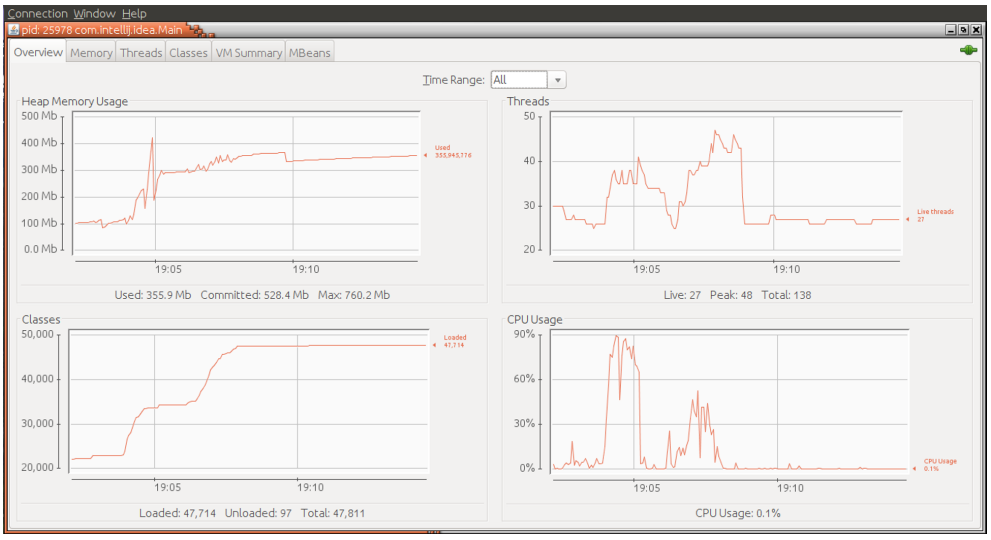

Figure 4.1: A JDK 8 JConsole tool main window. It shows use of resources, another tabs contain more detailed information and a facility to manage object called a MBeans.

1. Java Management Extensions, for more information visit `http://www.oracle.com/technetwork/articles/java/javamanagement-140525.html`

4.2 Automatic optimization tools

4.2.1 JATT

A JATT (JVM Auto Tuning Tool) is an open source software tool which was developed to optimize JVM. The JATT is based on the OpenTuner[2] framework and it supports only the Linux environment. Aforementioned tool offers the Console mode and the Graphical User Interface as well. Authors highly recommend to use the console mode for more advanced work. The JATT is designed to tune especially HotSpot JVM (namely, it was tested on OpenJDK 7 update 55) but it can be modified to auto tune a different JVM. [6] [1]

The *JATT* is an **offline** tuning tool – it means a tuning phase and a production phase are strictly separated. Firstly, an optimal parameter configuration that will produce best performance within specified deployment environment. Secondly, found optimal configuration is used to deploy the application. There is an opposite approach called **online** tuning which deals with finding an optimal configuration during the runtime of an application. The *JATT* does not offer an online auto tuning of JVM now but it is in development. An initial phase of development indicates use of a *jstat* utility[3]. [7]

Popsat JATT, applicare

2. "OpenTuner is a new framework for building domain-specific multi-objective program autotuners. OpenTuner supports fully customizable configuration representations, an extensible technique representation to allow for domain-specific techniques, and an easy to use interface for communicating with the tuned program." `http://opentuner.org/`

3. For more informations see sites `https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/tooldescr017.html` and `https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jstat.html` for use

Bibliography

[1] "Improving the performance of a real-time streaming solution by auto-tuning the JVM," https://dzone.com/articles/improving-performance-of-a-real-time-streaming-sol, [Online; visited 9-August-2017].

[2] "Java command documentation," http://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html, [Online; visited 7-December-2016].

[3] "Java ergonomics documentation," https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/ergonomics.html, [Online; visited 9-February-2017].

[4] "Monitoring and management using JMX technology," http://docs.oracle.com/javase/8/docs/technotes/guides/management/agent.html, [Online; visited 8-August-2017].

[5] "Performance of Java versus C++," http://scribblethink.org/Computer/javaCbenchmark.html, [Online; visited 08-February-2017].

[6] "JATT - java virtual machine auto tuning tool," https://sites.google.com/site/hotspotautotuner/home, [Online; visited 9-August-2017].

[7] W. Fernando, M. Kumara, J. Perera, and C. Philips, "Auto-tuning HotSpot JVM literature review," *University of Moratuwa*, pp. 1–2, 2012.

[8] C. Hunt and B. John, *Java™ Performance*, 1st ed. Addison-Wesley, 2012.

[9] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java® Virtual Machine Specification Java SE 8 Edition*, Oracle, 2 2015.

[10] A. Mehta, A. Saxena, and T. Bhawsar, "A brief study on jvm," *Asian Journal of Computer Science Engineering*, pp. 1–2, 2016.

[11] S. Oaks, *Java™ Performance: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2014.

A Big impact options

Primitive
-d32 | -d64
-server | -client
-Xmixed | -Xcomp | -Xint
-Xms<size>
-Xmx<size>
-Xss<size>
-XX:AggressiveOpts
-XX:BackgroundCompilation
-XX:CICompilerCount
-XX:CICompilerCountPerCPU
-XX:CompileThreshold
-XX:DoEscapeAnalysis
-XX:ErgoHeapSizeLimit
-XX:InitialTenuringThreshold
-XX:Inline
-XX:MaxMetaspaceSize
-XX:MaxNewSize
-XX:MaxTenuringThreshold
-XX:MetaspaceSize
-XX:NewRatio
-XX:NewSize
-XX:SoftRefLRUPolicyMSPerMB
-XX:SurvivorRatio
-XX:ThreadStackSize
-XX:UseBiasedLocking
-XX:UseGCOverheadLimit
-XX:UseParallelOldGC
-XX:UseSerialGC

Table A.1: The big impact primitive options. It's possible to use any option without enabling the other one. These options subject subsequent optimization.

Complex	
-server	-XX:TieredCompilation
-XX:UseParallelGC	-XX:InitialSurvivorRatio
	-XX:MinSurvivorRatio
	-XX:TargetSurvivorRatio
	-XX:GCTimeRatio
-XX:UseConcMarkSweepGC	-XX:CMSInitiatingOccupancyFraction
	-XX:UseCMSInitiatingOccupancyOnly
	-XX:ConcGCThreads
	-XX:CMSClassUnloadingEnabled
-XX:UseAdaptiveSizePolicy	-XX:UseAdaptiveSizePolicyFootprintGoal
	-XX:UseAdaptiveSizePolicyWithSystemGC
-XX:UseG1GC	-XX:ConcGCThreads
	-XX:UseStringDeduplication
	-XX:G1MixedGCLiveThresholdPercent
All GC except Serial GC	-XX:ParallelGCThreads

Table A.2: The big impact complex options. To use the option from the right column it's necessary to enable the option from the left column. These options subject subsequent optimization.