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

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

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 program called *bytecode* which is stored in `class` file. A Java `class` file is produced from `java` source file by Java compiler. JVM, like a real computing machine, has 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*". [1]

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." [2]

"To implement the Java Virtual Machine (JVM) correctly, you need only be able to read the `class` file format and correctly perform the specified operations. Implementation details that are not part of the Java Virtual Machine's specification would unnecessarily constrain the creativity of implementors. For example, the memory layout of run-time data areas, the garbage-collection algorithm used, and any internal optimization of the JVM instructions (for example, translating them into machine code) are left to the discretion of the implementor." [1]

There are many implementations of JVM[1]. This chapter of thesis deals with the Java HotSpot Virtual Machine respective to Java Development Kit 8 (JDK 8) which is primary reference JVM implementation. This version is latest because JDK 9 release is scheduled for July 2017[2].

---

1. Extensive list of JVM implementation is accessible on `https://en.wikipedia.org/wiki/List_of_Java_virtual_machines`
2. `https://blogs.oracle.com/java/proposed-schedule-change-for-java-9`

2.1 Java HotSpot VM

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

First version of JVM was interpreted. Since statement "*Java is slow*" endures notwithstanding it's not true in these days. There will be described why it's not true in next sections.

As mentioned earlier, main purpose of JVM is executing *bytecode* on specific platform which means translating *bytecode* to native code of CPU. Because interpreting of *bytecode* had appeared like inefficient there was introduced approach called *Just-In-Time* compilation (JIT) in Java 1.2 [3]. A JVM implementation with JIT compiler translates program into native code on the fly. Native code is cached and reused without recompiled. 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 code is executed frequently in a common programs. Frequent code sections represent approximately 20 %[3] which is 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. [4]

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 achieved but for the price of the slow interpretation in the beginning. Actually, depending on the level of optimization there are two different compilers in HotSpot JVM – **client** and **server**.

Another important role of any JVM implementations is automatic memory management knowns as **Garbage collections (GC)**. There are

3. `http://artiomg.blogspot.cz/2011/10/just-in-time-compiler-jit-in-hotspot.html`

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 different performance characteristics and is suitable for different category of an application and environment. [4]

Je nutné v následujících podkapitolách podrobněji rozebrat jednotlivé překladače a kolektory? Přece jen se práce zaměřuje na ladění výkonu. Na druhou stranu bez uvedení specifik pro jednotlivé kolektory nebude možné se odkazovat proč jsme například použili CMS místo parallel a naopak. Přikláněl bych se je uvést protože např. parallel GC ma za cíl co nejvyšší propustnost a CMS GC zase minimalizuje délku *stop-the-world* pauzy. Pokud bychom měli metriku s nejvýznamnější váhou právě na minimalizaci *stop-the-world* (předpokládám, že tato situace může nastat, protože cílová aplikace – terminál – obsluhuje požadavky od uživatele) bude možné říct, PROČ je CMS GC vhodnější oproti jiným kolektorům.

2.1.1 Client compiler

2.1.2 Server compiler

2.1.3 Garbage collectors

2.2 Setting options

There are many options how to adjust behavior JVM. For example, we may enforce to use parallel collector or set maximum heap[4] size. Options are divided into several categories:

4. Heap is a memory area that JVM uses for residing the Java objects.

- "**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 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`." [5]

For listing all standard options is used 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 point of interest in this thesis because these options not meant for VM tunning 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. Boolean options use 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`. [4]

2.3 Java ergonomics

"Ergonomics is the process by which the Java Virtual Machine (JVM) and garbage collection tuning, such as behavior-based tuning, improve

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. Server-class machine has been defined as machine with two or more physical processors and two or more GB of physical memory. Taken from [6].

application performance. The JVM provides platform-dependent default selections for the garbage collector, heap size, and runtime compiler. These selections match the needs of different types of applications while requiring less command-line tuning. In addition, behavior-based tuning dynamically tunes the sizes of the heap to meet a specified behavior of the application." [6]

Default values which are shown in the list produced by `java -XX:+PrintFlagsFinal -XX:+UnlockExperimentalVMOptions` command are based on Java ergonomics. E.g. for server-class machine is default value for initial heap size of 1/64 of physical memory up to 1 GB and maximum heap size of 1/4 of physical memory up to 1 GB and parallel collector is set as default. [6]

3 Methodology of choosing important options

First step to tunning JVM for performance optimization is determination options which has considerable performance impact. The option set contains 815 options[1]. In addition, it can be assumed that options without documentation, on Oracle's official sites ([5]) and options which are not mentioned in a literature deals with Java performance ([4], [7]), are not so important.

It's possible to categorize JVM options into following groups:

- **Big Impact** – these options were evaluated as most important for performance impact. Some options require enabling another option and are called *complex* (e.g. for use `-XX:TieredCompilation` option is necessary using `-server` option). Another options doesn't require enabling "parent" option and are called *primitive*. Lists of *complex* (see 5.2) and *primitive* (see 5.1) options are stored in the appendix of this thesis.
- **Small Impact** – options in this group are considered having not too much big impact for performance but in the certain situations can significantly influence JVM behavior (e.g. `-XX:GCHeapFreeLimit` option).
- **Not Relevant** – essentially, there are options that don't have any influence for performance. They are particularly "printers" – options which only print some diagnostic information (e.g. `-XX:PrintGCDateStamps`) or commercial features.
- **Not Documented** – represents the biggest set of options with no documentation (e.g. `-XX:CompactFields`)

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

---

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).

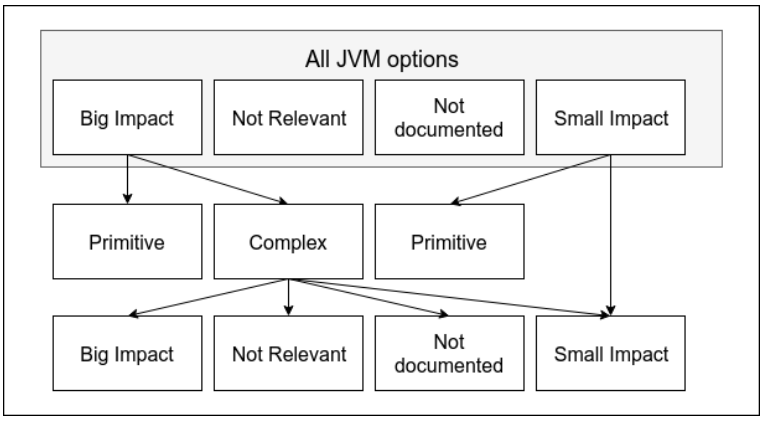

Figure 3.1: JVM options distribution into several categories with regard to performance impact and a documentation extent.

4 Tools

4.1 Analytic tools

4.2 Automatic optimization tools

Complex - C1/C2

JVM je zasobnikovy pocitac Ergonomics - Automatic Selections and behavior tuning

Java Flight Recorder Moznosti analyzy vykonu - vmstat jconsole, prepinace -XX:PrintGCDetails/-XX:PrintGCTimeStamps

Bibliography

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

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

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

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

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

[6] "Java ergonomics documentation," https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/ergonomics.html, [Online; visited 9-February-2016].

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

5 Appendix

Primitive
-d32 | -d64
-server | -client
-Xmixed | -Xcomp | -Xint
-Xms<size>
-Xmx<size>
-Xss<size>
-XX:AggressiveOpts
-XX:AlwaysTenure | -XX:NeverTenure
-XX:AutoBoxCacheMax
-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:ParallelGCThreads
-XX:SoftRefLRUPolicyMSPerMB
-XX:SurvivorRatio
-XX:ThreadStackSize
-XX:UseBiasedLocking
-XX:UseGCOverheadLimit
-XX:UseParallelOldGC
-XX:UseSerialGC

Table 5.1: Big impact primitive option. It's possible to use any option without enabling other one.

Complex	
-server	-XX:TieredCompilation
-XX:UseParallelGC	-XX:InitialSurvivorRatio
	-XX:MinSurvivorRatio
	-XX:TargetSurvivorRatio
	-XX:MaxGCPauseMillis
	-XX:GCTimeRatio
-XX:UseConcMarkSweepGC	-XX:CMSInitiatingOccupancyFraction
	-XX:UseCMSInitiatingOccupancyOnly
	-XX:ConcGCThreads
	-XX:CMSClassUnloadingEnabled
-XX:UseAdaptiveSizePolicy	-XX:UseAdaptiveSizePolicyFootprintGoal
	-XX:UseAdaptiveSizePolicyWithSystemGC
-XX:UseG1GC	-XX:MaxGCPauseMillis
	-XX:ConcGCThreads
	-XX:UseStringDeduplication
	-XX:G1MixedGCLiveThresholdPercent

Table 5.2: Big impact complex options. To use option from right column it's necessary to enable option from left column.