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 an ability for applications to be developed in a platform-independent manner and it can be shortened by Sun Microsystems slogan: "*Write once, run anywhere*". [18]

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." [19]

For the correct implementation of Java Virtual Machine, you only need to be able to read the `class` file and perform the specified operations. The Java Virtual Machine specification doesn't contain 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 the implementation of JVM are left to the implementor. Consequently, the JVM specification doesn't describe the possible ways how to adjust a JVM behavior to achieve a better performance of a running application. [18]

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 an official JDK 9 release is scheduled for September 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.

2.2 Compilation

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 [10]. A JVM implementation with JIT compiler translates a program into native code on the fly. Native code is cached and reused without recompiling. 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. [20]

Oracle's HotSpot VM combines interpretation and translation. First, it interprets all code and concurrently collects information how often the code is executed and additional data. After that JVM uses collected

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

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. Actually, depending on the aggressiveness and level of optimization there are two different compilers in HotSpot JVM – **client** and **server** (see sections 2.2.1 and 2.2.2).

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. [20]

2.2.1 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. [15]

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 2.2.2) will be used. Sometimes *client* compiler is called *compiler 1* or shortly *C1*[4]. [20] [5]

2.2.2 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 behavior 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 runtime of an application and the *server* compiler is suitable for long running applications. [15]

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

4. The short notation is used in a list of advanced options. See section 2.5

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. [20]

2.3 Memory management

The Java SE platform provides an automatic memory management. The JVM specification prescribes that any JVM implementation must ensure reclaiming unused memory – unreachable objects also called *garbage* or dead objects. A process of reclaiming unreachable objects is known as *garbage collection* (GC). A developer does not care about a complexity of memory allocation and *garbage collection*. [7]

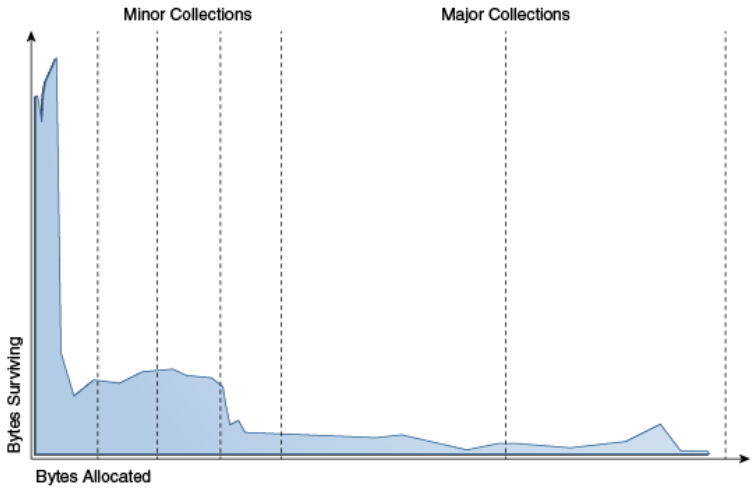

Figure 2.1: Typical distribution for lifetimes of objects. A majority of objects "die young". Taken from [7].

The *weak generational hypothesis* is generally true for Java applications which means: [15]

5. If command `java -client -XX:+TieredCompilation` *programName* is used, `-XX:+TieredCompilation` option is quietly turned off. [20]

- "Most allocated objects become unreachable quickly.
- Few references from older to younger objects exist."

Therefore the Java HotSpot VM splits the heap[6] into two areas (also called spaces), which are referred to as generations: [15]

- **The young generation** – the vast majority of new objects are allocated in this area. Most objects become into *garbage* quickly and a small part of it survives a young generation collection – *minor* garbage collection. Reclaiming objects from this area is usually effective because they occupy relatively a small space and mostly are dead.
- **The old generation** – long-live objects are moved (also called *promoted* or *tenured*) to the old generation. This area is also known as *tenured* space. The old generation is usually larger than the young generation and it grows slowly. *Garbage collection* in *tenured* space – *major* garbage collection is less frequent, but it can be quite lengthy.

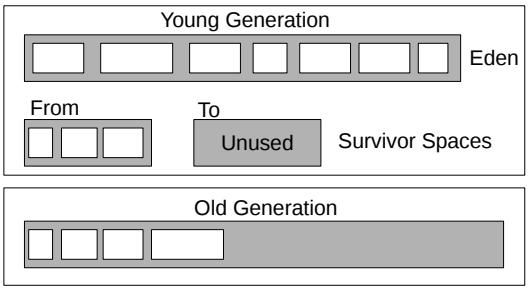

Figure 2.2: A Java HotSpot VM heap division. Adjusted from [15].

There is a more detailed heap division of JVM in figure 2.2. The *young generation* contains *Eden* space and two *Survivor* spaces. Almost all new objects are allocated in *Eden* space (large objects can be allocated directly in the old generation) and if an object is live after at least one *minor* garbage collection it moves to one of *Survivor* space. The object has to be reachable after several *minor* garbage collection to

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

promoted to the old generation. One of *Survivor* space remains unused. A *minor* garbage collection process is depicted in figure 2.3.

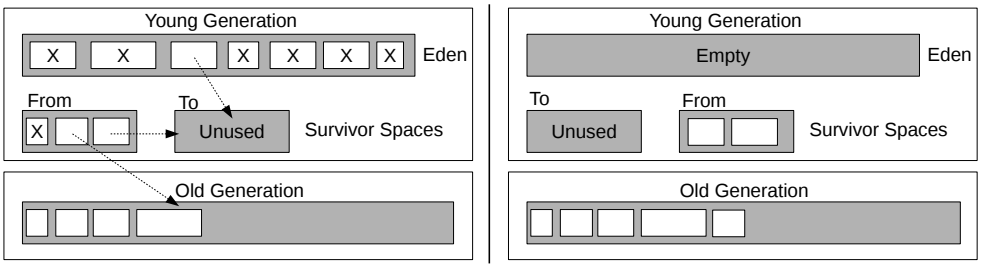

Figure 2.3: *Minor* garbage collection. Objects with a cross on the left side are unreachable. Adjusted from [15].

Compared to previous Java version, there is one essential change in a Java 8 memory management. There is not *Permanent* space anymore. Instead of this memory area, we can find a *Metaspace* in a current JVM. Both areas serve for storing class meta-data, method objects, interned strings and static variables. As seen in figure 2.4, *Permanent* generation is in the heap while *Metaspace* is not. It is part of native memory (process memory) which is only limited by the host operating system. [9]

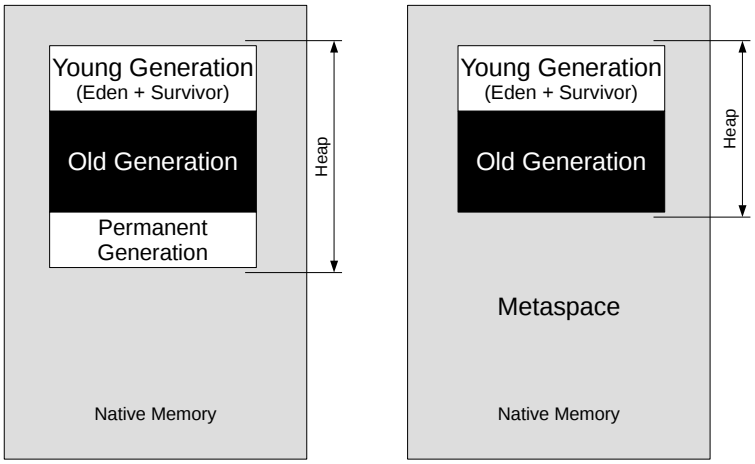

Figure 2.4: A permanent space (generation) has replaced *Metaspace*. Adjusted from [9].

2.4 Garbage collectors

At this place some, terms used in next sections will be described [20] [15]:

- **stop-the-world** – stopping of all application threads to perform *GC*. These pauses generally have the greatest impact on an application performance.
- **marking** – a phase refers to finding all reachable objects. Every object has an extra bit – the *mark bit* reserved for memory management.
- **sweeping** – deallocating *dead* objects
- **compaction** – defragmentation (usually the *old generation*) a memory space
- **bump-the-pointer** – it means that *garbage collector* remembers the start of a free memory space. It leads to faster objects allocation.

There are several algorithms which ensure *Garbage collection* and called *Garbage collectors*. Specifically, a HotSpot JVM contains four *Garbage collectors* with different impacts on performance. These techniques how to deallocate unused memory will be described in next sections.

2.4.1 Serial collector

The serial collector is the simplest one, designed for single-thread environments and small heaps (approximately up to 100 MB). Whenever it's working, the serial collector freezes all application threads and uses a single thread to perform garbage collection, which is relatively efficient (there is no communication overhead between threads). Therefore, this collector is best-suited to single processor machines. Over the *old generation*, it uses a *mark-compact* collection method with *bump-the-pointer* technique. That means, older objects are moved to the beginning of memory so objects which will be promoted from the *young generation* will be allocated faster into the continuous chunk of memory. [7] [15]

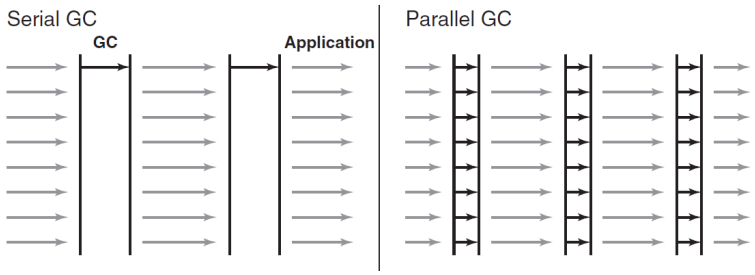

Figure 2.5: A comparison serial and parallel *garbage collectors*. The serial collector uses only one thread for *garbage collection* while the parallel collector can capitalize a multiprocessor environment. Taken from [15]

2.4.2 Parallel collector

The parallel collector (also known as *Throughput collector*) is suitable for multiprocessor machines where applications with medium-sized to large-sized data sets are running. A name of this collector is derived from the way how it processes a *garbage* – it uses multiple threads to perform *garbage collection*. Figure 2.5 describes the main difference between serial and parallel collectors. Both, *minor* and *major*[7] collections are performed in parallel, which can reduce *GC* overhead and *stop-the-world* delay. [7] [20] [15]

2.4.3 CMS collector

A *CMS* is an abbreviation for *Concurrent Mark-Sweep GC*. It is designed for reducing long *stop-the-world* pause. The *minor GC* is usually a short time operation and is done in a manner similar to the *Parallel collector*. On the other side, a *dead* objects removing from the *old generation* signifies by long pauses, especially when large Java heaps are involved. Therefore, *CMS* uses one or more background threads to scan through the *old generation* and remove *dead* objects concurrently. [20] [7]

Figure 2.6 illustrates partial phases of *CMS GC*. Firstly, a short pause, called *initial marking*, that targets to find a set of objects reachable from outside the *old generation*. Secondly, objects reachable from

7. Since JDK 7u4. The serial collector was used for *major collection* earlier. [20]

this set are marked during the *marking* phase. Because it's concurrent to the running application, at the end of this phase, not all live objects are guaranteed to be marked (application might be updating reference fields). Thirdly, to reduce the further amount of the work in next *remark* phase, the *pre-cleaning* was introduced. The use of *pre-cleaning* can reduce, sometimes dramatically, the number of objects that need to be visited during the *remark* pause, and, as a result, it is very effective in reducing the duration of the *remark* pause. Fourthly, a *remark* pause occurs to finalize the marking information by revising any object that was modified during the preceding two phases. The *remarking* is a more demanding task then *initial marking*, it is parallelized to increase its efficiency. After *remark* phase, all *live* objects are guaranteed marked. But it is not guaranteed that all *dead* objects will be identified as *garbage*. They will be processed during next *GC* cycle while occupying a memory space (Usually referred to as *floating garbage*). This is a typical trade-off *CMS garbage collector*. Fifthly, *dead* objects are deallocated during concurrent *sweeping* phase without relocating the *live* ones. This is the last phase of *CMS major GC* cycle. [20] [7] [15]

The *minor* and *major* collections occur independently. They do not overlap, but when one pause is quickly followed by another, it can appear to be a single long pause. To avoid this, *CMS GC* attempts schedule *remark* phase between two *minor* collections. Only for *remarking* because *initial marking* usually does not evince longer pause.[7]

This *garbage collector* algorithm has increased requirements on the Java heap because of several reasons. Concurrent *marking* phase takes longer than the *stop-the-world* pause and actually in the *sweeping* phase *garbage* objects are deallocated. In addition, a *floating garbage* can waste a memory space. As mentioned above, *sweeping* phase does not compact the memory space and cannot profit with a *bump-the-pointer* technique as well. It leads into fragmentation an *old generation* and *tenured* objects allocation becomes harder. When the heap is too fragmented to allocate a new object, *CMS* reverts to the behavior similar to *serial* or *parallel GCs* – it stops application threads to clean and compact the *old generation* resulting into expensive *stop-the-world* pause. [15]

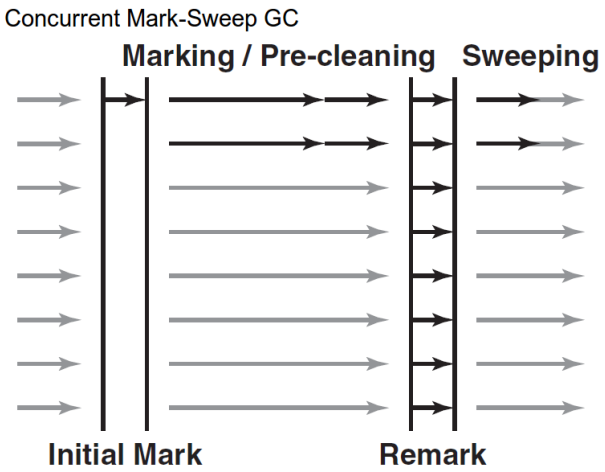

Figure 2.6: A *CMS GC* phases. Taken from [15]

2.4.4 G1 collector

Garbage first is another naming for this collector. The *garbage first GC* targets to reduce *stop-the-world* pause as well. In addition, it removes one of the biggest disadvantages of *CMS GC*, i.e. a fragmentation memory, *G1* is a *compacting GC*. [7]

The *G1* also differentiates between *young* and *tenured* objects but it divides a heap in a different way. Figure 2.7 illustrates the *G1* heap division. Actually, the heap is partitioned into fixed-sized pieces of contiguous memory range (also called *regions*). While an application is running *garbage first GC* concurrently performs global *marking* through the whole heap to determine *live* objects. Based on *marking*, *G1* can identify almost empty *regions*, i.e. the majority of objects are *dead*. When *garbage collection* will focus on these *regions*, more memory space will be released. And that is a basic principle of *G1 garbage collector* – therefore the *G1* is called *garbage first* also. After *marking* phase the *G1* copies *live* objects from one or more *regions* in a single *region* on the heap[8]. This process reclaims and continuously compacts memory – it's also called an *evacuation*. During one *evacuation* phase can be collected

8. In the same way like previous *GCs*, i.e. *G1* copies live objects from the *Eden* to the *Survivor*, from the *Survivor* to other *Survivor* or promotes to the *old generation*, from the *old generation* to the *old generation* space.

dead objects from the *young* and *old generation* as well. For this kind of collection, a *mixed collection* term is used. [7]

An *evacuation* is performed in the *stop-the-world*, parallel manner. The *G1 GC* determines the number of collected *regions* thereby *stop-the-world* pause as well. The user can specify *GC* pause time goal and *G1* adjusts the number of processed *regions* to effect *GC* pause. But it's only "soft" goal and it's not guaranteed it will be passed. A *pause prediction model* is building based on history of *stop-the-world* pauses and the JVM will make the best effort to achieve this goal. In addition, there is another *stop-the-world* pause in the *G1 GC* for *remarking* but no pause for *initial marking* which is part of an evacuation pause. [7]

When the size of an object is greater than 50 % of single *region* size then a specialized type of allocation occurs, called *humongous* allocation. *Humongous* objects are allocated directly within the *old generation* to prevent a potential copying overhead. A *humongous* object can occupy more *regions* which is shown in figure 2.7. [4]

The *G1* is a long-term replacement for *CMS collector*. It will become a default *GC* in Java 9[9]. For more information about this *garbage collector* see official Oracle's documentation [7]. Scott Oaks detailed describes *G1* in [20].

2.5 Setting options

There are many options how to adjust JVM behavior. For example, we may enforce to use parallel collector or set maximum heap 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 classpath, 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`.

9. `https://docs.oracle.com/javase/9/whatsnew/toc.htm`

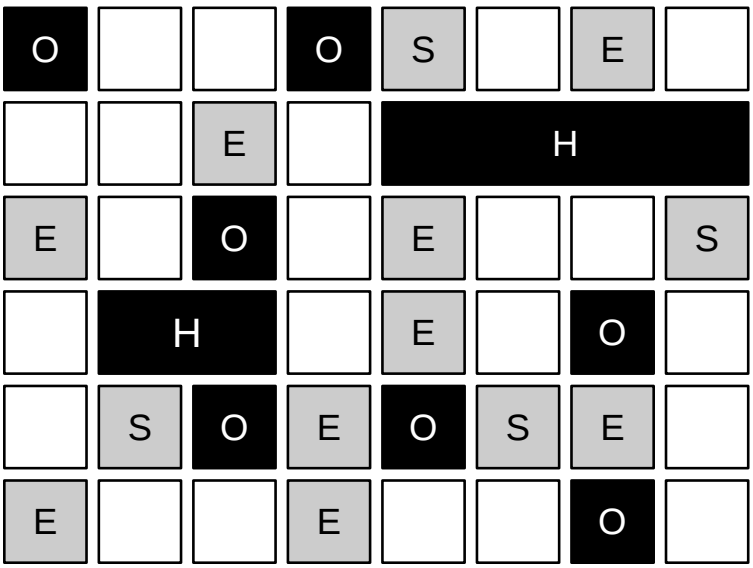

Figure 2.7: *G1 garbage collector* divides heap into fixed-sized regions. Every region belongs to the *young generation* (gray areas) or the *old generation* (O, black areas). White areas represent a unused memory space for a future allocating. There are still an *Eden* (E) and *Survivor* (S) spaces in the *young generation*. When an object is too large to allocate in a single region it takes over more regions direct in the *old generation* – it's called a *humongous* (H) object. Adjusted from [4] [7]

- **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 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 2.6) 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`. [20]

2.6 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* 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

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 [6].

- Parallel (throughput) garbage collector

Unless the initial and maximum heap sizes are specified on the command line, these values will be used. [6]

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 2.5, 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.

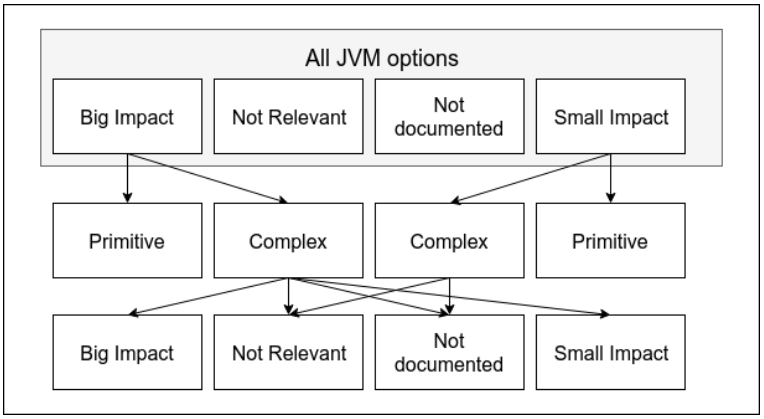

Figure 3.1: JVM options distribution into several categories with regard to performance impact and a documentation extent.

3.1 Big impact

These options were evaluated as most important for performance impact. The selection was based on the available Oracle's documentation ([5]) and a literature dealing with the Java performance ([20], [15]).

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 2.2.1 and server compiler see section 2.2.2.
- `-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. [20]
- `-Xmixed` – interprets all bytecode except for hot methods, which are compiled to native code. [5]
- `-Xcomp` – forces compilation to native code on the first invocation of method. [5]
- `-Xms<size>` – sets initial Java heap size. [5]
- `-Xmx<size>` – sets maximum Java heap size [5]. The JVM automatically sets the initial and maximum heap size (see section 2.6). 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 2.2.2.

3. For the heaps up to 3 GB. [20]

- `-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. [20]

- `-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 [5]. By default, value is based on the ParallelGCThreads option and is determined by equation: $ConcGCThreads = (3 + ParallelGCThreads)/4$. The calculation is using integer arithmetic. [20]

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. [5]. In Java 7, enabling this option means that different implementa-

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." [5]

tion of some 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.

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. [20]

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. [8]

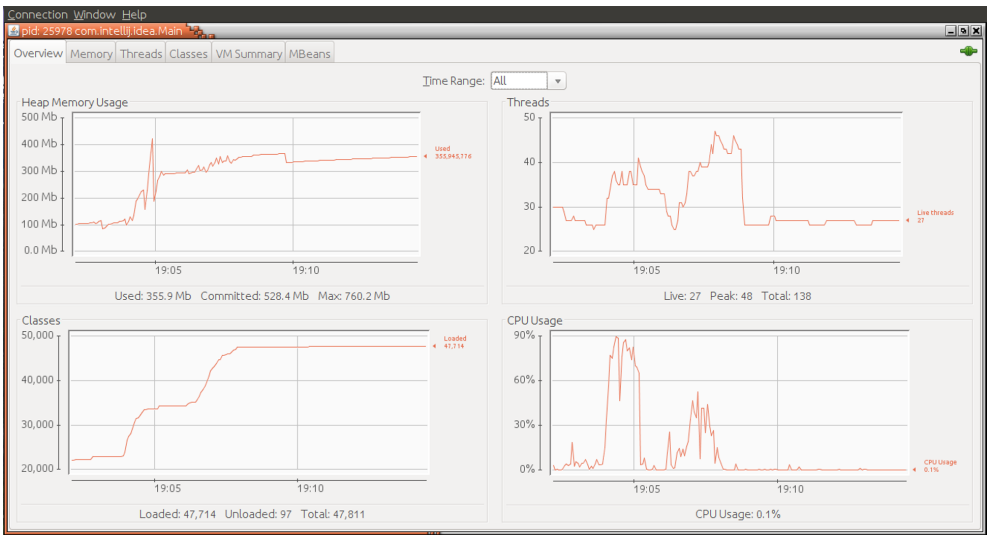

Figure 4.1: A JDK 8 JConsole tool main window. It shows use of resources, another tabs contain more detailed information and a facility to manage object called a MBeans.

4.1.2 GC Viewer

GC Viewer is a tool for analysis GC logs. Firstly, you have to run a JVM with options `-Xloggc:`*logFile* `-XX:+PrintGCDetails-XX:+PrintGCDa`

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

`teStamps`, where *logFile* is a file where GC events will be stored. Secondly, open the *logFile* in the GC Viewer to assessment of effectiveness GC operations, memory footprint, heap usage, throughput and so on. Therefore GC Viewer is an offline tool. This tool is completely written in Java language. Tagtraum industries[2] developed GC Viewer till 2008. Since this time it's open source project[3] under GNU Lesser General Public License.[11] The GC Viewer is being updated. All support JVMs are in table A.3.

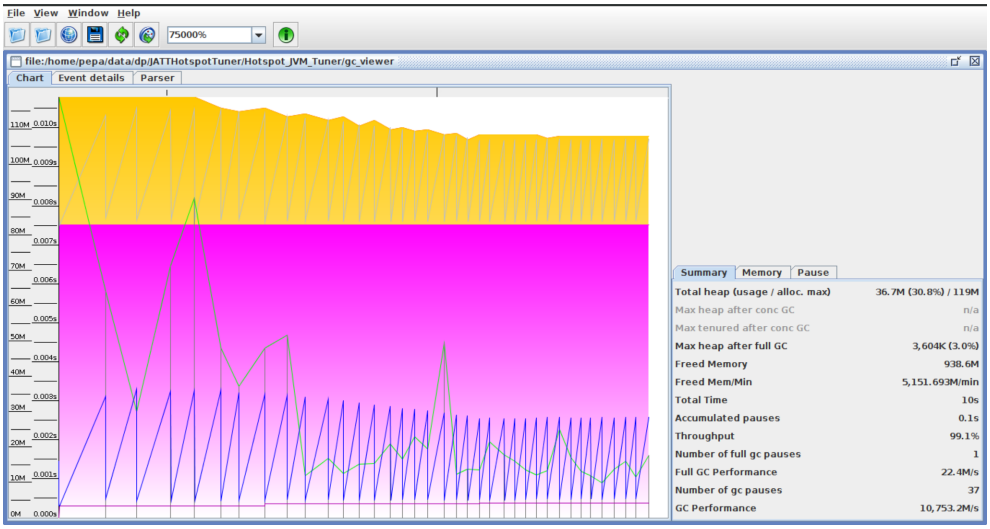

Figure 4.2: A GC Viewer GUI. This tool is can be used to detailed analysis of GC logs.

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[4]

2. http://www.tagtraum.com/gcviewer.html
3. Official repository is accessible at `https://github.com/chewiebug/GCViewer`
4. "OpenTuner is a new framework for building domain-specific multi-objective program autotuners. OpenTuner supports fully customizable configuration rep-

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. [12] [3]

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[5]. [14]

Already stated tool is written in Python programming language[6] and except for the OpenTuner it requires other packages. For a comprehensive information about requirements, installation and use see website [2]. There is shown a simple example of the Java application auto tuning and performance results are briefly discussed.

For logical division the JATT divides options into ten groups. We are able to auto tune JVM according to every group of our interest. Alongside this a searching space and a tuning time are reduced. We can easily use two or more groups at the same time or define own set of options to auto tune. All options are stored in `Flags` folder in `csv` files, including `Temporary` file which is predestined to input own list of options which the JATT will use to auto tune a JVM.

The JATT is student project which originated in 2012 at the University of Moratuwa, Sri Lanka. Project won the gold medal award at

---

resentations, 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/`

5. 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.

6. Official repository is located at `https://bitbucket.org/sapients/hotspottuner/`

Association for Computing Machinery (ACM) student research competition attached to 2015 International Symposium on Code Generation and Optimization. [12]

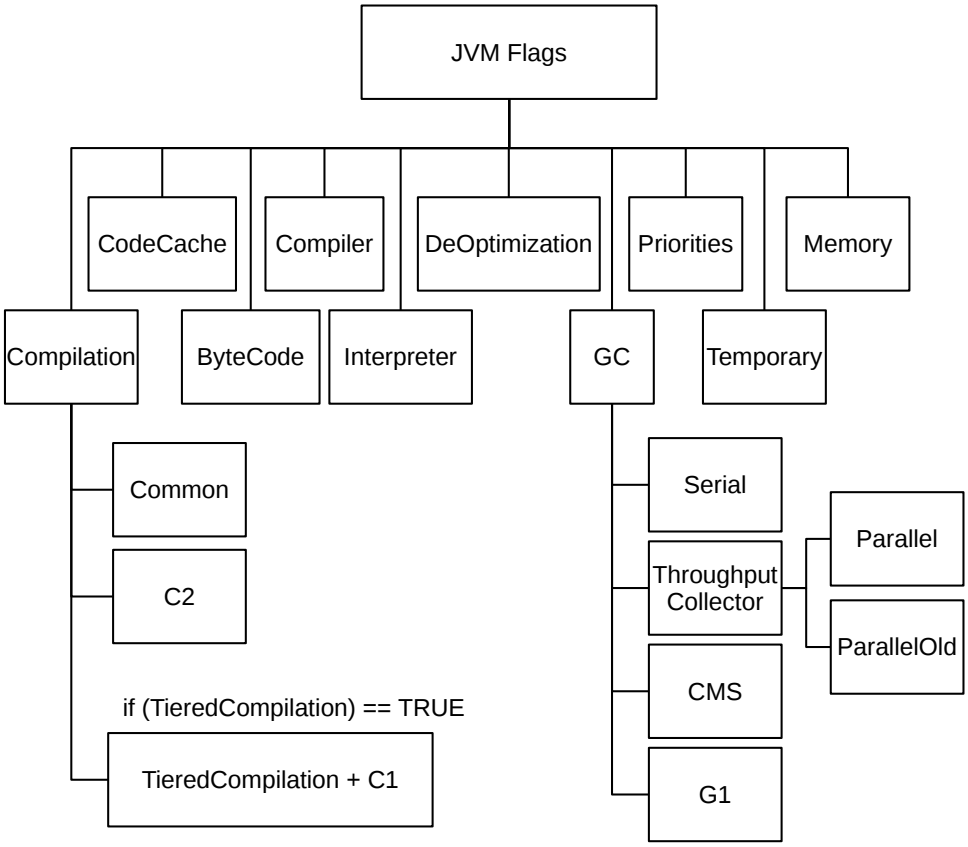

Figure 4.3: A JVM options hierarchy. The JATT divides JVM options into several groups. The division aims to reduce a searching space. But it's possible to use two or more groups together also. [21] [2]

To start auto tuning Java application, find `JATTHotspotTuner` folder and type command in terminal:
`python src/javaProgramTuner.py --source=`*application* `--iterations=`*numberOfIterations* `--flags=`*optionCategory* `--configfile=`*fileWithResults*

Parameters of command are:

- *application* – Java application to auto tune (a `*.class` file)
- *numberOfIterations* – measured runtime is average of *numberOfIterations* runs
- *optionCategory* – you can use one or more categories which are separated by commas (see picture 4.3)
- *fileWithResults* – a text file will be stored in `src/TunedConfiguration` folder with found configurations. If you more times run command with same *fileWithResults* data will not be overwritten while new records will be added. A result entry syntax is shown in figure 4.4

In figure 4.4 is an example of *fileWithResults* file. There is group(s) options configuration on the first line used to achieve given runtime. The `Improvement` represents ratio of improvements which is given by equation: $Improvement = Default\ metric/Runtime$ [3], where `Default metric` is the application runtime without changed JVM options. Result file ordinarily contains increasing entry sequence by improvement factor including a header with information about used option group(s) and tuning start time.

```
-XX:+Inline -XX:+ClipInlining -XX:+UseTypeProfile ... -XX:
AutoBoxCacheMax=120 -XX:EliminateAllocationArraySizeLimit=
56 -XX:ValueSearchLimit=1058 ...
Improvement: 1.05022358536
Runtime 0.114877080917
Default metric 0.120646619797
Configuration Found At: 2017-08-12 20:11:17.910694
```

Figure 4.4: An example of the JATT result entry. An option configurations of group(s) (shorted here) and improvement factor are the most important information.

4.2.2 ParamILS

Philipp Lengauer and Hanspeter Mössenböck deal with automatic parameter tuning for Java Garbage Collectors [17]. They introduced a tool

extensively based on ParamILS framework for JVM tuning, primarily for improving GC operations in 2014. The tool is practically short bash script which launch ParamILS with parameter model – options for tuning divided by Garbage collector algorithms in ParamILS-specic syntax. Own tool is under the GNU General Public License and the ParamILS software is owned by Meta-Algorithmic Technologies Inc. and permission is hereby granted for non-commercial use[7]. [17] [16] The ParamILS is a collection of easy readable Ruby scripts and both tools support Windows and Linux operating systems[8].

4.2.3 Commercial tools

Among important non-freely accessible tools belong:

- Arcturus Applicare[9] – comprehensive tool for automated performance tuning, intelligent monitoring, optimization and instant root cause analysis. Applicable for great systems such web applications.
- Dynatrace – this company is a leader for application performance monitoring and tuning for Java and .NET applications. Dynatrace collaborates with Philipp Lengauer and Hanspeter Mössenböck on JVM tuning. [1]
- Plumbr[10]

7. For commercial use contact Chris Fawcett, chris.fawcett@gmail.com
8. `http://www.cs.ubc.ca/labs/beta/Projects/ParamILS/ParamILS-Quickstart.pdf`
9. `http://www.arcturustech.com/tunewizard.html`
10. `https://plumbr.eu/`

5 Optimization JVM

5.1 Application

The DaCapo benchmark suite version 9.12 was chosen for the primal experimentation with JVM tuning. This suite includes these benchmarks: [13]

- **antlr** – A parser generator and translator generator
- **bloat** – A bytecode-level optimization and analysis tool for Java
- **chart** – A graph plotting toolkit and pdf renderer
- **eclipse** – An integrated development environment (IDE)
- **fop** – An output-independent print formatter
- **hsqldb** – An SQL relational database engine written in Java
- **jython** – A python interpreter written in Java
- **luindex** – A text indexing tool
- **lusearch** – A text search tool
- **pmd** – A source code analyzer for Java
- **xalan** – An XSLT processor for transforming XML documents

5.2 Measurement methodology

A machine with these parameters was used for measurement:

- CPU – Intel® Core™ i5-2450M, 2 cores, 4 threads
- Memory – 8 GB
- Operating system – Ubuntu 16.04, kernel Linux 4.10.0-33-generic (x86_64)

Firstly, 10 runs of *xalan* benchmark without warm-up period were realized. Measured values are in table B.1. Secondly, an auto tuning tool was used to gain JVM options. Thirdly, 10 runs of *xalan* benchmark with tuned options were realized again to observe performance changes.

5.3 JATT

The JATT provides an interface to auto tune DaCapo benchmark. A command `python src/dacapoTuner.py --source=xalan --iterations=1 --flags=gc --configfile=dacapo_results` starts finding better JVM options settings with respect to GC operations. Searching took approximately an hour and result configuration is in figure B. *Xalan* benchmark was performed with found JVM settings and measured values are in table B.2.

Average optimized runtime was 10415 milliseconds what in comparison to default value 11385 milliseconds is improvement by more than 8 %. Accumulated GC pause is shorter more than four times. To the contrary, there is huge increase in memory footprint – from original 128 MB to 742 MB. After detailed exploration logs in GC Viewer is apparent that only one GC event occurred.

A reduction runtime by 8 % is significant together without any code adjustment. But memory consumption could be problematic, especially when a Java application runs on a machine with limited memory.

With the goal of reducing memory footprint another JATT flag was used – namely *memory* flag. Memory footprint has been reduced from original 128 MB to 37 MB with runtime improvement by 6 %. In comparison to previous optimized JVM settings, there is runtime increase by approximately 2.5 %. Complete results are in table B.3 and found options are in figure B.

In this part of thesis will be *Big Impact* options (in table A.1) tune. As was described in section 4.2.1, the JATT tool allows tuning own set of JVM options. Chosen options are stored in `src/Flags/Temporary/temporary.csv` file and auto tuning initiates same command but with *temporary* flag. This file was filled with *Big Impact Primitive* options.

The best options setting what JATT has found is in figure B. *Xalan* benchmark achieved average values 12009 milliseconds for runtime and 46 MB for memory footprint. It means the application has run by 5 % longer than the application running in the JVM with the default settings and used about 82 MB less memory space. Measured values are in table B.4.

Finally, auto tuning by JATT with all option set was initiated. The best JVM settings was found approximately in two hours. Overall searching phase took four hours but JATT could not improve settings in the second half of auto tuning. Thanks to the using of all options runtime has been reduced by more than 20 % at the expense of increment memory footprint from original 128 MB to 231 MB. Complete results are in the table B.5.

Vysledne nastaveni tedy obsahuje pres 600 parametru, coz se do papirove prilohy uz nevyda. Lze je prilozit jako elektronickou prilohu do archivu prace.

5.4 ParamILS

This framework was used to optimize parameters stored in `gc_parameter_model/params_gc` file which led to improvement time and space complexity as well. Specifically, *Xalan* benchmark needed by 7 % runtime and almost 60 % memory space less to finish the test. Measured values are in the table B.6 and gained settings is in figure B.

5.5 Comparison

All results are enclosed in chapter B. In next two figures are graphically compared measured data. Apparently, setting which leads to the fastest application run is simultaneously the most memory consumption. Settings gained by JATT with *memory* flag and ParamILS with GC parameter model are better in both observed metrics.

When chosen *Big Impact* options was used to tune a JVM setting, it led to deterioration of runtime and memory footprint as well. It could be caused by overly restricted option set and/or incorrect combination of options which JATT cannot improve.

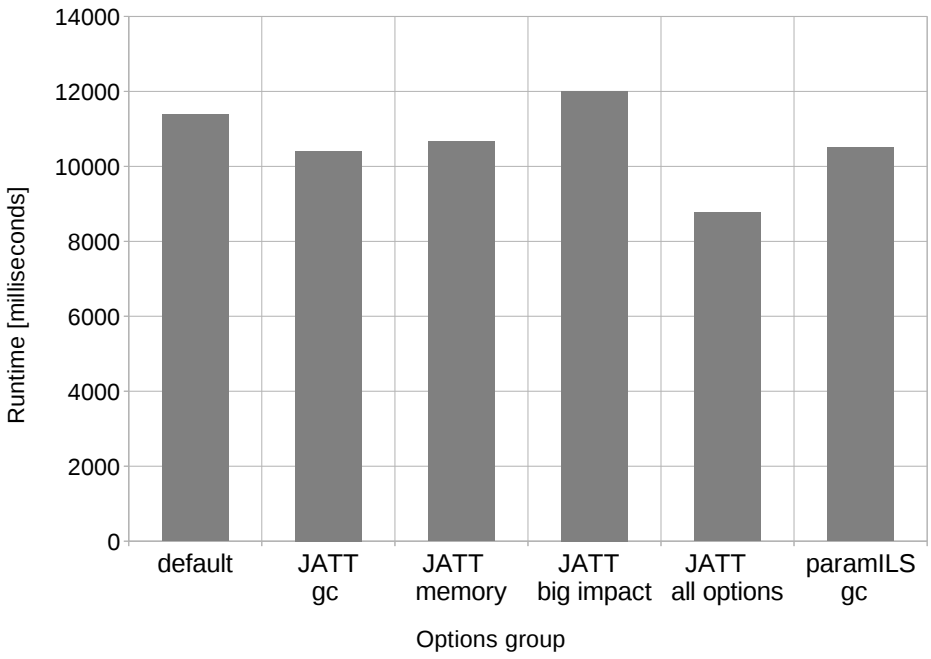

Figure 5.1: A comparison of an application runtime. Less is better.

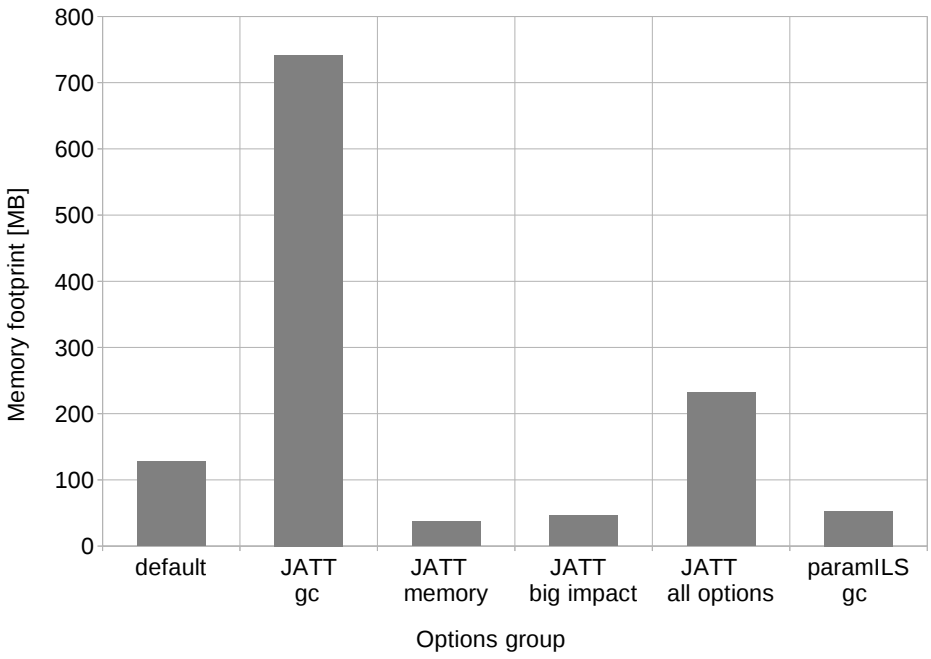

Figure 5.2: An overall application memory footprint. Less is better.

6 Conclusion

Will be written in the end.

Bibliography

[1] "Boost Java application performance (almost) automatically," https://www.dynatrace.com/blog/boost-java-performance-automatically/, [Online; visited 15-August-2017].

[2] "How to use JATT to tune a java program?" https://medium.com/@stsarut/how-to-use-jatt-to-tune-a-java-program-a1e4e86b28f4, [Online; visited 9-August-2017].

[3] "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].

[4] "Introduction to the G1 Garbage Collector," https://www.redhat.com/en/blog/part-1-introduction-g1-garbage-collector, [Online; visited 14-September-2017].

[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-2017].

[7] "Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide," http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/toc.html, [Online; visited 11-September-2017].

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

[9] "One important change in Memory Management in Java 8," http://karunsubramanian.com/websphere/one-important-change-in-memory-management-in-java-8, [Online; visited 11-September-2017].

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

[11] "GCViewer," https://github.com/chewiebug/GCViewer, [Online; visited 18-August-2017].

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

[13] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, Oct. 2006, pp. 169–190.

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

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

[16] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Paramils: An automatic algorithm configuration framework," *Journal of Artificial Intelligence Research*, vol. 36.

[17] P. Lengauer and H. Mössenböck, "The taming of the shrew: Increasing performance by automatic parameter tuning for java garbage collectors," *Johannes Kepler University Linz, Austria*, pp. 112–119.

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

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

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

[21] J. P. W. Fernando, M. Kumara and C. Philips, "Auto-tuning HotSpot JVM, project progress no. 2," University of Moratuwa, 2015.

A Big impact options

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

preliminary support for OpenJDK 9 Shenandoah algorithm in unified logging format `-Xlog:gc: -XX:+UseShenandoahGC`
Oracle JDK 1.8 `-Xloggc: [-XX:+PrintGCDetails] [-XX:+PrintGCDateStamps]`
Sun / Oracle JDK 1.7 with option `-Xloggc: [-XX:+PrintGCDetails] [-XX:+PrintGCDateStamps]`
Sun / Oracle JDK 1.6 with option `-Xloggc: [-XX:+PrintGCDetails] [-XX:+PrintGCDateStamps]`
Sun JDK 1.4/1.5 with the option `-Xloggc: [-XX:+PrintGCDetails]`
Sun JDK 1.2.2/1.3.1/1.4 with the option `-verbose:gc`
IBM JDK 1.3.1/1.3.0/1.2.2 with the option `-verbose:gc`
IBM iSeries Classic JVM 1.4.2 with option `-verbose:gc`
HP-UX JDK 1.2/1.3/1.4.x with the option `-Xverbosegc`
BEA JRockit 1.4.2/1.5/1.6 with the option `-verbose:memory [-Xverbose:gcpause,gcreport] [-Xverbosetimestamp]`

Table A.3: GC Viewer supports JVMs list. `https://github.com/chewiebug/GCViewer`

B Results

Run	Runtime [msec]	Memory Footprint [MB]	Accumul. Pauses [sec]	Full GC Perf. [MB/s]	GC Perf. [MB/s]
1	10716	119	0.08	28.3	13937.1
2	11100	119	0.1	15.2	11384.2
3	11882	135	0.1	16.4	10167.9
4	11268	119	0.09	24.3	12406.4
5	11513	119	0.11	23.3	8924.2
6	11539	162.5	0.09	15.3	11932.2
7	11163	119	0.09	23.5	11662.3
8	12043	120	0.12	21.8	8300.6
9	11325	119	0.08	24	12755
10	11300	147	0.08	12.9	13569.9
Average	**11385**	**127.85**	**0.094**	**20.5**	**11503.98**

Table B.1: *Xalan* benchmark runs in the default settings of the JVM.

Run	Runtime [msec]	Memory Footprint [MB]	Accumul. Pauses [sec]	Full GC Perf. [MB/s]	GC Perf. [MB/s]
1	10397	741.8	0.02	7857.2	130215.7
2	10633	741.8	0.02	5357.2	135230.5
3	10249	741.8	0.02	7933.9	141060.9
4	10521	741.8	0.02	6213	120291.4
5	10212	741.8	0.02	6718.6	88053.9
6	10448	741.8	0.02	5970.8	97636.3
7	10421	741.8	0.02	7614.7	131317.1
8	10385	741.8	0.02	6431.9	98775.7
9	10385	741.8	0.03	8095.7	39761.2
10	10497	741.8	0.02	5864.4	109633.8
Average	**10415**	**741.8**	**0.021**	**6805.74**	**109197.65**

Table B.2: *Xalan* benchmark runs with settings found by JATT with *gc* flag. Used settings is in a figure B

Run	Runtime [msec]	Memory Footprint [MB]	Accumul. Pauses [sec]	Full GC Perf. [MB/s]	GC Perf. [MB/s]
1	11203	36.7	0.09	16.7	12548
2	10571	36.6	0.09	17.8	12601.3
3	10627	36.8	0.09	12.6	13282.9
4	10379	36.7	0.09	13.2	12.682
5	10602	36.7	0.08	24.4	14580.6
6	10559	36.7	0.09	14.5	12622.3
7	10718	36.6	0.08	23.6	13130.1
8	11026	36.8	0.09	18.9	12101.7
9	10494	36.7	0.09	24.1	11375.6
10	10525	36.7	0.09	19.1	11919.8
Average	**10670**	**36.7**	**0.088**	**18.49**	**11417.4982**

Table B.3: *Xalan* benchmark runs with settings found by JATT with *memory* flag. Used settings is in a figure B

Run	Runtime [msec]	Memory Footprint [MB]	Accumul. Pauses [sec]	Full GC Perf. [MB/s]	GC Perf. [MB/s]
1	11934	46.1	0.08	22.1	13831.4
2	12034	45.9	0.06	27.6	19230.4
3	11950	45.8	0.06	22.3	18530
4	12034	46	0.07	23	16739.3
5	12083	46	0.07	37	14005.5
6	11984	46.1	0.07	30.8	15018.3
7	12152	46	0.06	26.8	18473.5
8	12035	46.1	0.07	24	16687.3
9	12000	46.1	0.06	33.3	19396.4
10	11880	46.1	0.06	25.1	17798
Average	**12009**	**46.02**	**0.066**	**27.2**	**16971.01**

Table B.4: *Xalan* benchmark runs with settings found by JATT with temporary – big impact options. Used settings is in a figure B

Run	Runtime [msec]	Memory Footprint [MB]	Accumul. Pauses [sec]	Full GC Perf. [MB/s]	GC Perf. [MB/s]
1	8707	231.3	0.03	879.9	67354.3
2	8949	231.4	0.03	1523.2	74420.9
3	8794	231.4	0.03	2100.2	71483.6
4	8731	231.3	0.03	1600.6	73481.3
5	8750	231.5	0.03	1874.4	72028.1
6	8740	231.5	0.03	1590.3	72196.1
7	8734	231.4	0.03	1557.1	60351.8
8	8742	231.5	0.04	1049.7	57957.4
9	8736	231.3	0.02	1962.3	73589.8
10	8791	231.5	0.05	1986.8	26118.7
Average	**8767**	**231.41**	**0.032**	**1612.45**	**64898.2**

Table B.5: *Xalan* benchmark runs with settings found by JATT for all options.

Run	Runtime [msec]	Memory Footprint [MB]	Accumul. Pauses [sec]	Full GC Perf. [MB/s]	GC Perf. [MB/s]
1	10562	52.7	0.07	n/a	13680.5
2	10595	52.9	0.06	n/a	16334.1
3	10197	52.7	0.06	n/a	16309.2
4	10202	52.9	0.06	n/a	16146.6
5	10663	52.7	0.07	n/a	14377.5
6	10660	52.9	0.07	n/a	13791.2
7	10430	52.9	0.08	n/a	12153.7
8	10683	52.9	0.07	n/a	12806.3
9	10457	53	0.06	n/a	16134.5
10	10523	52.9	0.06	n/a	15632.7
Average	**10497**	**52.85**	**0.066**	**n/a**	**14736.63**

Table B.6: *Xalan* benchmark runs with settings found by ParamILS with GC parameters model. Used settings is in a figure B

```
-XX:+UseParNewGC -XX:+ParGCUseLocalOverflow -XX:-ParGCTrimO
verflow -XX:-CMSParallelRemarkEnabled -XX:-CMSParallelSurv
ivorRemarkEnabled -XX:-CMSPLABRecordAlways -XX:+CMSConcurr
entMTEnabled -XX:ParGCArrayScanChunk=66 -XX:ParGCDesiredOb
jsFromOverflowList=25 -XX:CMSYoungGenPerWorker=83001713 -X
X:-UseConcMarkSweepGC -XX:+ResizePLAB -XX:+ResizeOldPLAB -
XX:+AlwaysPreTouch -XX:+ParallelRefProcEnabled -XX:+Parall
elRefProcBalancingEnabled -XX:+UseTLAB -XX:+ResizeTLAB -XX:
-ZeroTLAB -XX:-FastTLABRefill -XX:+NeverActAsServerClassMac
hine -XX:+AlwaysActAsServerClassMachine -XX:-UseAutoGCSele
ctPolicy -XX:-UseAdaptiveSizePolicy -XX:-UsePSAdaptiveSurv
ivorSizePolicy -XX:-UseAdaptiveGenerationSizePolicyAtMinor
Collection -XX:+UseAdaptiveGenerationSizePolicyAtMajorColl
ection -XX:+UseAdaptiveSizePolicyWithSystemGC -XX:-UseAdap
tiveGCBoundary -XX:-UseAdaptiveSizePolicyFootprintGoal -XX
:-UseAdaptiveSizeDecayMajorGCCost -XX:+UseGCOverheadLimit -
XX:-DisableExplicitGC -XX:+CollectGen0First -XX:-BindGCTas
kThreadsToCPUs -XX:-UseGCTaskAffinity -XX:YoungPLABSize=52
37 -XX:OldPLABSize=836 -XX:GCTaskTimeStampEntries=263 -XX:
TargetPLABWastePct=10 -XX:PLABWeight=37 -XX:OldPLABWeight=
50 -XX:MarkStackSize=4875124 -XX:MarkStackSizeMax=37052363
9 -XX:RefDiscoveryPolicy=0 -XX:InitiatingHeapOccupancyPerc
ent=27 -XX:MaxRAM=200187289457 -XX:ErgoHeapSizeLimit=0 -XX
:MaxRAMFraction=2 -XX:DefaultMaxRAMFraction=6 -XX:MinRAMFr
action=3 -XX:InitialRAMFraction=69 -XX:AutoGCSelectPauseMi
llis=7432 -XX:AdaptiveSizeThroughPutPolicy=0 -XX:AdaptiveS
izePausePolicy=0 -XX:AdaptiveSizePolicyInitializingSteps=1
8 -XX:AdaptiveSizePolicyOutputInterval=0 -XX:AdaptiveSizeP
olicyWeight=15 -XX:AdaptiveTimeWeight=25 -XX:PausePadding=
0 -XX:PromotedPadding=3 -XX:SurvivorPadding=1 -XX:Threshol
dTolerance=10 -XX:AdaptiveSizePolicyCollectionCostMargin=3
0 -XX:YoungGenerationSizeIncrement=28 -XX:YoungGenerationS
izeSupplement=79 -XX:YoungGenerationSizeSupplementDecay=6 -
XX:TenuredGenerationSizeIncrement=28 -XX:TenuredGeneration
SizeSupplement=43 -XX:TenuredGenerationSizeSupplementDecay
=3 -XX:MaxGCPauseMillis=16793486230599948288 -XX:GCPauseIn
tervalMillis=0 -XX:MaxGCMinorPauseMillis=12369914655517609
984 -XX:GCTimeRatio=145 -XX:AdaptiveSizeDecrementScaleFact
or=4 -XX:AdaptiveSizeMajorGCDecayTimeScale=14 -XX:MinSurvi
vorRatio=2 -XX:InitialSurvivorRatio=4 -XX:BaseFootPrintEst
imate=391267844 -XX:GCHeapFreeLimit=3 -XX:PrefetchCopyInte
rvalInBytes=766 -XX:PrefetchScanIntervalInBytes=534 -XX:Pr
efetchFieldsAhead=1 -XX:ProcessDistributionStride=5
```


Figure B.1: JVM options for *xalan* benchmark found by JATT with *gc* flag.

```
-XX:-UseSharedSpaces -XX:+RequireSharedSpaces -XX:-DumpShar
edSpaces -XX:+RelaxAccessControlCheck -XX:+UseVMInterrupti
bleIO -XX:ThreadSafetyMargin=72943233 -XX:StackYellowPages
=2 -XX:StackShadowPages=25 -XX:ThreadStackSize=1328 -XX:VM
ThreadStackSize=1304 -XX:CompilerThreadStackSize=0 -XX:Sha
redReadWriteSize=16288215 -XX:SharedReadOnlySize=10350874 -
XX:SharedMiscDataSize=5788599 -XX:SharedMiscCodeSize=77206
```

Figure B.2: JVM options for *xalan* benchmark found by JATT with *memory* flag.

```
-XX:+BackgroundCompilation -XX:+CICompilerCountPerCPU -XX:+
DoEscapeAnalysis -XX:-Inline -XX:-UseBiasedLocking -XX:-Us
eGCOverheadLimit -XX:-UseParallelOldGC -XX:-UseSerialGC -X
X:CICompilerCount=4 -XX:CompileThreshold=7316 -XX:ErgoHeap
SizeLimit=0 -XX:InitialTenuringThreshold=4 -XX:MaxGCPauseM
illis=20402228389095275883 -XX:MaxMetaspaceSize=2627439390
9091921920 -XX:MaxNewSize=718203203 -XX:MaxTenuringThresho
ld=13 -XX:MetaspaceSize=20627973 -XX:NewRatio=2 -XX:NewSiz
e=47816779 -XX:SoftRefLRUPolicyMSPerMB=1229 -XX:SurvivorRa
tio=9 -XX:ThreadStackSize=1399
```

Figure B.3: JVM options for *xalan* benchmark found by JATT with *temporary – big impact* options.

```
-XX:MaxHeapFreeRatio=90 -XX:AdaptiveSizeMajorGCDecayTimeSca
le=15 -XX:YoungGenerationSizeSupplement=90 -XX:AdaptiveSiz
eThroughPutPolicy=1 -XX:YoungGenerationSizeSupplementDecay
=2 -XX:-ResizePLAB -XX:+DisableExplicitGC -XX:TenuredGener
ationSizeSupplement=70 -XX:TenuredGenerationSizeSupplement
Decay=16 -XX:MinSurvivorRatio=2 -XX:+BindGCTaskThreadsToCP
Us -XX:MaxTenuringThreshold=4 -XX:NewRatio=1 -XX:-UseAdapt
iveSizePolicy -XX:MinHeapFreeRatio=10 -XX:YoungPLABSize=10
24 -XX:-UseAdaptiveSizeDecayMajorGCCost -XX:+CollectGen0Fi
rst -XX:AdaptiveSizePausePolicy=1 -XX:AdaptiveSizePolicyIn
itializingSteps=80 -XX:PLABWeight=80 -XX:YoungGenerationSi
zeIncrement=10 -XX:AdaptiveSizePolicyWeight=40 -XX:+UseAda
ptiveSizePolicyWithSystemGC -XX:AdaptiveSizeDecrementScale
Factor=2
```

Figure B.4: JVM options for *xalan* benchmark found by ParamILS with gc parameters model.