Masaryk University
Faculty of Informatics

Optimization of JVM settings for application performance

Master's Thesis

Josef Pavelec

Brno, Fall 2017

Masaryk University
Faculty of Informatics

Optimization of JVM settings for application performance

Master's Thesis

Josef Pavelec

Brno, Fall 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.

Acknowledgements

I would like to thank my thesis supervisor, RNDr. Andriy Stetsko, PhD., for the guidance, advice and remarks throughout the whole process of working on this diploma thesis. A big thanks go also to Ing. Leoš Kunc for technical consultation and sharing experiences in given problematics. I thank them both for their time and patience.

I also thank my family, girlfriend and my friends for their strong support during my labor on this thesis and throughout my studies.

Abstract

The Java Virtual Machine contains many options for adjustment of settings. The options are selected based on performance impact. The autotuner applications are used to find better setting than the default one. New settings are being measured and evaluated in comparison with original ones. All the results are presented graphically. Further continuations are discussed in the conclusion of the thesis.

Keywords

Java, JVM settings, automatic optimization, garbage collector, paramILS, Java HotSpot Autotuning Tool

Contents

1 Introduction **1**

2 Java Virtual Machine **2**
2.1 *Java HotSpot VM* 3
2.2 *Compilation* 3
2.2.1 Client compiler 4
2.2.2 Server compiler 4
2.3 *Memory management* 5
2.4 *Garbage collectors* 7
2.4.1 Serial collector 8
2.4.2 Parallel collector 9
2.4.3 CMS collector 9
2.4.4 G1 collector 11
2.5 *Setting options* 12
2.6 *Java ergonomics* 14

3 Categorization of JVM options **16**
3.1 *Impacting under all circumstances* 17
3.2 *Impacting under some circumstances* 20
3.3 *Not relevant* 22
3.4 *Not documented* 22

4 Tools **23**
4.1 *Analytic tools* 23
4.1.1 JConsole 23
4.1.2 GC Viewer 24
4.1.3 top 26
4.1.4 iotop 26
4.1.5 ProcessData 26
4.2 *Automatic optimization tools* 27
4.2.1 JATT 27
4.2.2 OpenTuner 30
4.2.3 ParamILS 32
4.2.4 Commercial tools 33

5 JVM settings optimization **35**

5.1 *Application* 35
5.2 *Metrics* 35
5.3 *Measurement methodology* 36
5.4 *JATT* 37
5.5 *ParamILS* 37
5.6 *Results* 38

6 Conclusion **41**

A Electronic appendix information **45**

B Additional information **46**

C Charts **49**

1 Introduction

The *Java* programming language has still the biggest community of developers nowadays. [13] Applications that use the *Java Virtual Machine* (JVM) environment can be written even in different programming languages. As an example the programming language Kotlin can be mentioned. In May 2017, the Google company has officially claimed its support of the Kotlin programming language for Android application development. [3]

The JVM itself contains hundreds of options which allow modification of its working. Though it includes process for setting default values of options for the given environment where it runs, such setting does not always have to be optimal. This thesis is focused on the JVM setting optimization for increasing performance.

The official release of version 9 is available currently, improving performance as any new version of the software. This thesis deals with Java 8 version (software version 9 was not available yet during elaboration phase of the thesis).

The first part of the thesis deals with two main tasks that are being performed by the JVM. It is a compilation and automatic memory management, including descriptions of particular algorithms and comparison with the previous version. In the next part the JVM options with a given impact on performance are overviewed and discussed. The selection was performed based on the official Oracle sources [7] and available bibliography [24] [19]. Following part describes tools, that were used for analysis and optimization itself, including usage description, programming language, licence agreement, pricing policy (if applicable) and supported platforms. The last but one chapter focuses on optimization. The chosen applications are narrated, metrics defined and the achieved results presented and discussed.

2 Java Virtual Machine

Java Virtual Machine is an abstract computing machine. JVM cannot 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 is 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*". [22]

In the context of the JVM three terms should be distinguished:

- "**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." [23]

For the correct implementation of Java Virtual Machine, it is only required 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. [22]

There are many implementations of JVM[1]. This chapter of the thesis deals with the Java 8 HotSpot Virtual Machine which is the

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

primary reference JVM implementation. This version was the latest during writting thesis because an official version of Java 9 was not available.

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, opinion that "*Java is slow*" has persisted notwithstanding it is no longer true. 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* was proved inefficient, an approach called *Just-In-Time* compilation (JIT) was introduced in Java 1.2 [12]. 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 according to approach it takes towards compiling the code. A small part of the code is executed frequently in common programs. Frequent code sections represent approximately 20 %[2] which is in accordance with the Pareto principle. These sections are called *hot spots*. The more often 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. [24]

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 information for compilation with a high level of optimization. The

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

more information about code JVM has, the higher 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 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. [24]

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. [19]

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*[3]. [24] [7]

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. [19]

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

Some JVM implementations don't contain both compilers[4].

3. The short notation is used in a list of advanced options. See section 2.5
4. E.g. Java HotSpot 64-bit for a Linux platform

A technique called *tiered compilation* combines a *client* compiler fast startup and a *server* compiler high optimization. With the *tiered compilation*, the code is compiled first 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 default in the JDK 8. [24]

2.3 Memory management

The Java SE platform provides an automatic memory management. The JVM specification defines 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*. [9]

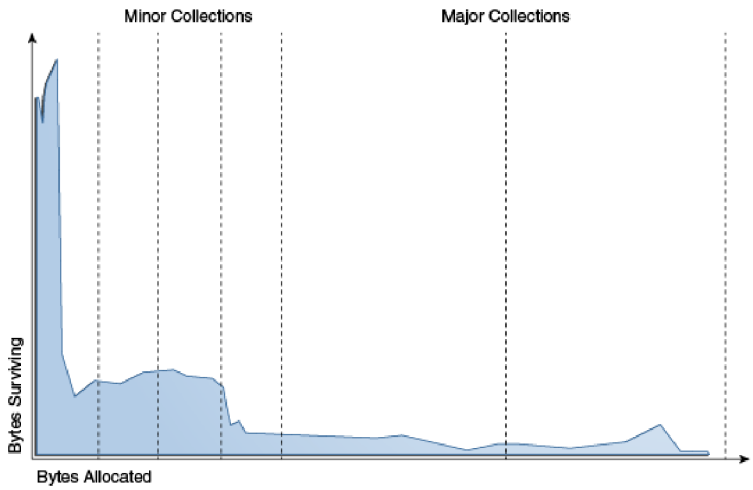


Figure 2.1: Typical distribution for lifetimes of objects. A majority of objects "die young". Taken from [9].

The *weak generational hypothesis* is generally true for Java applications which means: [19]

5. If command `java -client -XX:+TieredCompilation` *programName* is used, `-XX:+TieredCompilation` option is quietly turned off. [24]

- "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: [19]

- **The young generation** – the vast majority of new objects are allocated in this area. Most objects become *garbage* quickly and a small part of it survive 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 long-lasting.

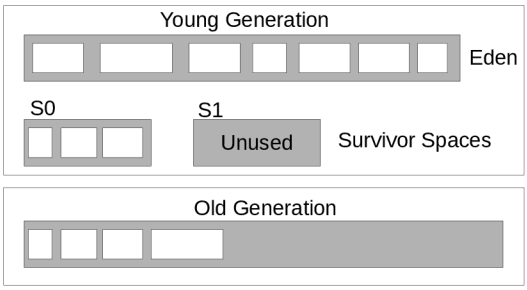


Figure 2.2: A Java HotSpot VM heap division. Adjusted from [19].

There is a more detailed heap division of JVM in figure 2.2. The *young generation* contains *Eden* space and two *Survivor* spaces (*S0* and *S1*). Almost all new objects are allocated in *Eden* space (large objects can be allocated directly in the old generation) and if an object is alive after at least one *minor* garbage collection it moves to one of *Survivor* spaces. The object has to be reachable after several *minor* garbage

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

collections to *promoted* to the old generation. One of *Survivor* spaces 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 [19].

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 the *Metaspace* is not. It is part of native memory (process memory) which is only limited by the host operating system. [11]

2.4 Garbage collectors

At this place some terms used in next sections will be described [24] [19]:

- **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.

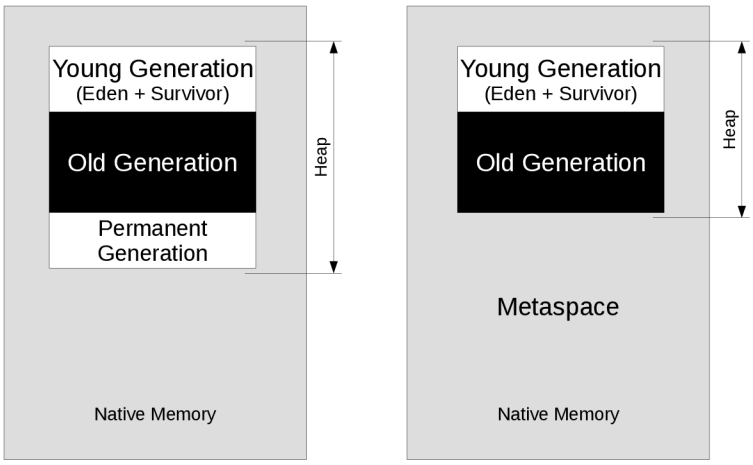


Figure 2.4: A permanent space (generation) has been replaced by a *Metaspace*. Adjusted from [11].

- **compaction** – defragmentation (usually the *old generation*) of 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 is 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. [9] [19]

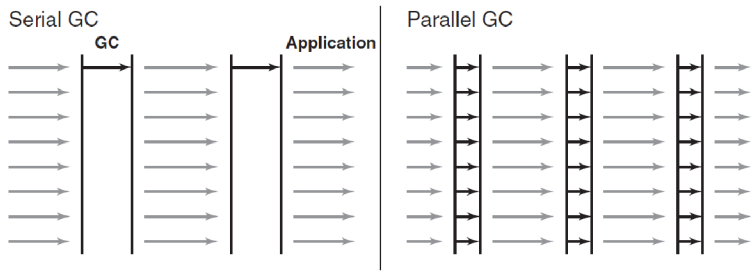


Figure 2.5: A comparison of a serial and parallel *garbage collectors*. The serial collector uses only one thread for *garbage collection* while the parallel collector can capitalize a multiprocessor environment. Taken from [19]

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. [9] [24] [19]

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 *young generation garbage collection* is usually a short time operation and it 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

7. Since JDK 7u4. The serial collector was used for *major collection* earlier. [24]

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. [24] [9]

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 this set are marked during the *marking* phase. Because it is 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. [24] [9] [19]

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.[9]

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 of 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. [19]

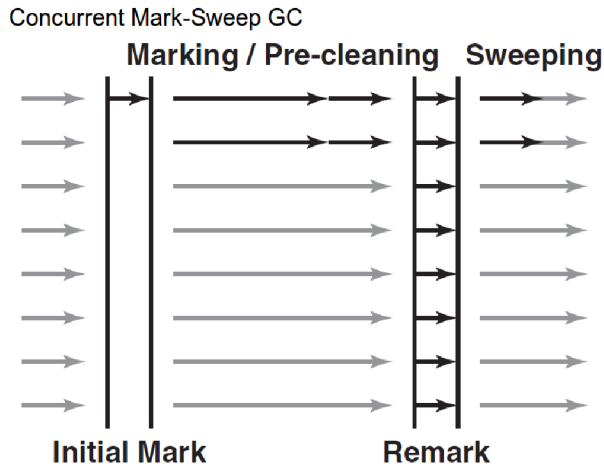


Figure 2.6: A *CMS GC* phases. Taken from [19]

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*. [9]

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* too. 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 is also called an *evacuation*. During one *evacuation* phase *dead* objects can be collected from the *young* and *old generation* as well. For this kind of collection, a *mixed collection* term is used. [9]

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. User can specify *GC* pause time goal and *G1* adjusts the number of processed *regions* to effect *GC* pause. But it is only a "soft" goal and it is not guaranteed it will be pass. 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. [9]

When a 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. [6]

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 [9]. Scott Oaks detailed describes *G1* in [24].

2.5 Setting options

There are many options how to adjust JVM behavior, compiling and memory management. 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.

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

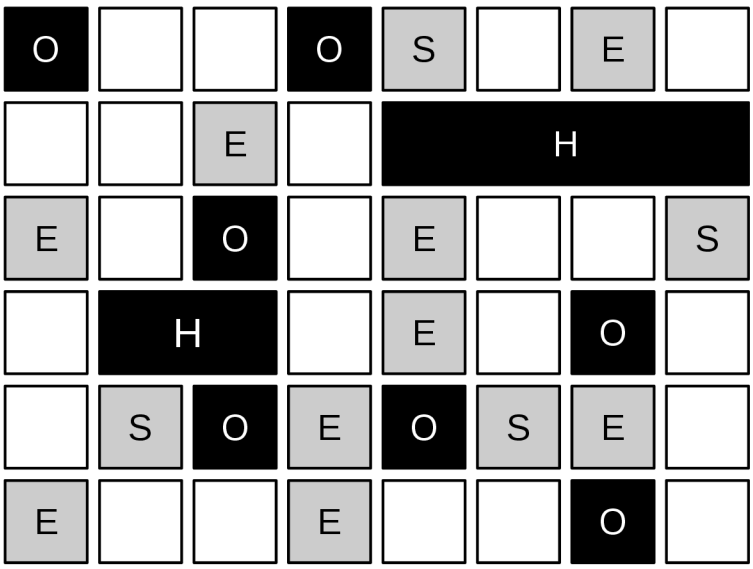


Figure 2.7: *G1 garbage collector* divides heap into fixed-sized regions. Every region belongs to the *young generation* (gray areas) or the *old generation* (O, black areas). White areas represent an unused memory space for a future allocating. There are still an *Eden* (E) and *Survivor* (S) spaces in the *young generation*. When an object is too large to be allocated in a single region it takes over more regions direct in the *old generation* – it is called a *humongous* (H) object. Adjusted from [6] [9]

- **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`." [7]

The command `java -?` is used for listing all standard options and `java -X` for non-standard options . 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 are 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`. [24]

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 options and mainly improve the performance of a running application. In addition, behavior-based tuning dynamically tunes the sizes of the heap to achieve a smaller memory footprint and to meet a specified behavior of the application. Depending on the environment features where the JVM runs the *Server-class* machine can be defined. 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 which 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:

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 [8].

- 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. [8]

3 Categorization of JVM 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 the 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 behavior. Principles of execution *bytecode* by JVM have been studied together with options which can adjust the execution. Based on the level influencing the JVM behavior and availability of a documentation it is possible to categorize JVM options into following groups:

- **Impacting under all circumstances**
- **Impacting under some circumstances**
- **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 a sense 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 don't require enabling of "parent" option.
- *complex* – for usage of a specific option it is 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 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'".

3.1 Impacting under all circumstances

These options were evaluated as those with the biggest impact on performance. The selection was based on the available Oracle's documentation ([7]) and a literature dealing with the Java performance ([24], [19]).

The impacting under all circumstances options, according to the previously mentioned distribution, are composed by two subsets – the *primitive* and the *complex*. Most of them will be used to for further optimization (e.g. by using any tool described in section 4.2). Several others will be used to meet any performance goal such as limit heap size (`-XX:MaxHeapSize`).

Among the most important *primitive* options are:

- `-XX:InitialHeapSize` – sets initial Java heap size. This option is equivalent to `-Xms<size>`. [7]

- `-XX:MaxHeapSize` – sets maximum Java heap size [7]. 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 unnecessarily large and their setting is a basic memory footprint tuning. The `-XX:MaxHeapSize` option is equivalent to -Xmx<size>.

- `-XX:NewSize` – sets the initial size (in bytes) of the heap for the young generation. [7]

- `-XX:NewMaxSize` – sets the maximum size (in bytes) of the heap for the young generation. [7]. It is possible to use `-Xmn<size>` to set both initial and maximal size of the heap for the young generation.

- `-XX:UseAdaptiveSizePolicy` – controls how the JVM alters the ratio of young generation to old generation within the heap. To see how the JVM is resizing the spaces in an application, set the `-XX:+PrintAdaptiveSizePolicy`. By default, this value is true. [7] [24]

- `-XX:NewRatio` – sets the ratio of the young generation to the old generation. By default, this value is 2. [7][24]

- `-XX:ThreadStackSize` – sets the thread stack size. This option is equivalent to `-Xss`. The default value depends on virtual memory. [7] It is desirable to increase this value for applications with deep resursive calls.

- `-XX:ReservedCodeCacheSize` – sets the maximum code cache size for JIT-compiled code. By default, when the *tiered compilation* is disabled (`-XX:-TieredCompilation`), then this value is 48 MB and 240 MB otherwise. This option has a limit 2 GB. It is possible to set initial code cache size (`-XX:InitialCodeCacheSize`) also. Default initial value of code cache size is based on the *Java ergonomics* process (for more information see section 2.6), e.g. on Intel machines, the server compiler starts with a 2,496 KB cache. Resizing the cache happens in the background and doesn't really affect performance. [7][24]

- `-XX:SurvivorRatio` – it means the ratio between *eden* and *survivor* space (for more information about these memory areas see section 2.3). By default, it is set to 8. [7]

- `-XX:Inline` – getter and setter methods represent relative big amount of code. When this option is enabled (it is by default), attributes are accessible directly in compiled code. Disabling *Inline* processing leads into a performance degradation. [24]

- `-XX:CICompilerCount` – number of threads destined to compilation. [7] By default, it should be 2 for *server* machine by official documentation but for machine used in this thesis is default value 3 (for machine details se section 5.3).

- GC selection – it is one of the basic *JVM* tunning. The section 2.4 deeply describes each *garbage collector*. The option naming corresponds with *GC* to use. All of these options are *boolean* options:

  - `-XX:UseSerialGC`
  - `-XX:UseParallelGC`
  - `-XX:UseParallelOldGC`
  - `-XX:UseConcMarkSweepGC`
  - `-XX:UseG1GC`
  - `-XX:UseParNewGC` – using parallel GC in the young generation. By default, it is disabled. However, enabling *CMS GC* automatically enables this option. The configuration `-XX:+UseParNewGC -XX:-UseConcMarkSweepGC` became deprecated in JDK 8. [7]

Several *complex* options with a impact on the performance under all circumstances:

- `-XX:TieredCompilation` – is available only if `-server` option is enabled. It enables tiered compilation described in section 2.2.2.

- `-XX:ParallelGCThreads` – the setting of this option makes sense only if another GC apart from *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 is 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. [24]

- `-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 [7]. By default, value is based on the ParallelGCThreads option and is determined by equation: $ConcGCThreads = (3 + ParallelGCThreads)/4$. The calculation is using integer arithmetic. [24]

- `-XX:CMSInitiatingOccupancyFraction` – when textitCMS GC is used. Set the percentage of the old generation occupancy to start the concurrent cycle. If the concurrent cycle starts when 60 % of the old generation is filled, it is better chance to finish while the cycle starts when 70 % of the old generation is filled. Using the `-XX:+UseCMSInitiatingOccupancyOnly` option guarantees that *CMS GC* starts the background thread exactly when the old generation is filled according the specified value. It uses complex algorithm otherwise (`-XX:UseCMSInitiatingOccupancyOnly` is false by default). [24]

- `-XX:CompileThreshold` – adjusts method invoke number before compilation to native code. It is 1000 for *client* and 10000 for *server* by default. This option is ignored when *tiered compilation* is enabled (`-XX:+TieredCompilation`). [24] For more information see section 2.2

3.2 Impacting under some circumstances

The impacting under some circumstances set contains the JVM options that are also important. Although they don't influence a JVM behavior in general, their use could have a big impact on performance in certain cases. For example, `-XX:OptimizeStringConcat` option should be used for applications merging strings extensively. In the other consideration, when an optimization aims to minimize the memory footprint, the `-XX:MaxHeapFreeRatio`[3] option significantly influences a success of optimization.

3. "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." [7]

Therefore, several options can be temporarily moved to the option set intended to optimization to achieve better performance of given application and/or metric. Selected options which meet the definition of the impacting under some circumstances, for example:

- `-client | -server` – choice of compiler. For more information about client compiler see section 2.2.1 and server compiler see section 2.2.2. These options are categorized within this category because some JVM implementations don't contain both compilers.
- `-d32 | -d64` – when a 32-bit operating system is used, then using of a 32-bit version of the JVM is required. When a 64-bit operating system is used, then it is possible to choose 32-bit or 64-bit version of the JVM. The 32-bit version will be faster and have a smaller footprint[4], because 32 bit memory references occupy smaller memory area and manipulating those references is less expensive. [24]
- `-Xmixed` – interprets all bytecode except for hot methods, which are compiled to native code. [7]
- `-Xcomp` – forces compilation to native code on the first invocation of a method. [7]
- `-XX:MinHeapFreeRatio` – when the percentage of free heap space after a GC event falls below this value, JVM will expand the heap to keep set ratio. By default, this value is set to 40 %.[7]
- `-XX:UseGCOverheadLimit` – limits the proportion of time spent by the JVM on GC before an *OutOfMemoryError* exception is thrown. By default, this option is true. The *JVM* throws *OutOfMemoryError* if more than 98 % is spent on garbage collection and less than 2 % of the heap is recoverd. [7]
- `-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.

4. For the heaps up to 3 GB. [24]

- `-XX:AggressiveOpts` – enables the use of aggressive performance optimization features, notably for compilation. [7]. In Java 7, enabling this option means that different implementation of some classes will be used[5]. Functionality of these classes is the same but they have more efficient implementations. Since Java 8, there are not alternate implementations in the JVM[6].

Complex impacting under some circumstances options are following:

- `-XX:G1HeapRegionSize` – defines the size of *G1 GC* area. For more information see section. 2.7
- `-XX:UseCondCardMark` – enables checking of whether the card is already marked before updating the card table. This option is disabled by default and should only be used on machines with multiple sockets. Only when *server* compiler is used. [7]

Selected options from both described groups are in the table B.2. This option group is intended for further optimization and it is called *temporary* or shortly *temp* (especially in charts).

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. [24]

4 Tools

There are introduced tools which were used for an analysis and autotuning in next sections. An usage, license, pricing and programming language are described for every tool. This chapter is divided into two main sections. Firstly, monitoring and analytic tools are discussed. Secondly, *autotuners* are introduced.

4.1 Analytic tools

With the exception of the first tool all tools in this subsection have been used for results processing and analyzing.

4.1.1 JConsole

JConsole is a graphical tool that allows monitoring and managing Java applications on a local or remote machine. It is contained in JDK since Java 5 because in this version Java Management Extension (JMX) technology was introduced and JConsole is a JMX-compliant tool. [10]

Using the JConsole we can monitoring memory areas usage, number of threads and loaded classes of running java application. It supports managing as well. This is achieved by managing objects called *MBeans*. It is possible to read from or write to objects through JMX[1].

JConsole is a tool providing GUI for monitoring and managing running Java applications only. It can be extended using plugins e.g. for watching/checking Hibernate[2] etc., but it does not offer any interface for console or recording data to files in various formats, which would be useful for later statistic evaluation of application performance. For these reasons JConsole is not suitable for exact application performance measurement and it is mentioned here for complementary monitoring.

---

1. For more information about JMX visit `http://www.oracle.com/technetwork/articles/java/javamanagement-140525.html` and how to manage *MBeans* by JConsole visit `https://docs.oracle.com/javase/8/docs/technotes/guides/management/jconsole.html`
2. The Hibernate is a framework for object-relational mapping for the Java applications.

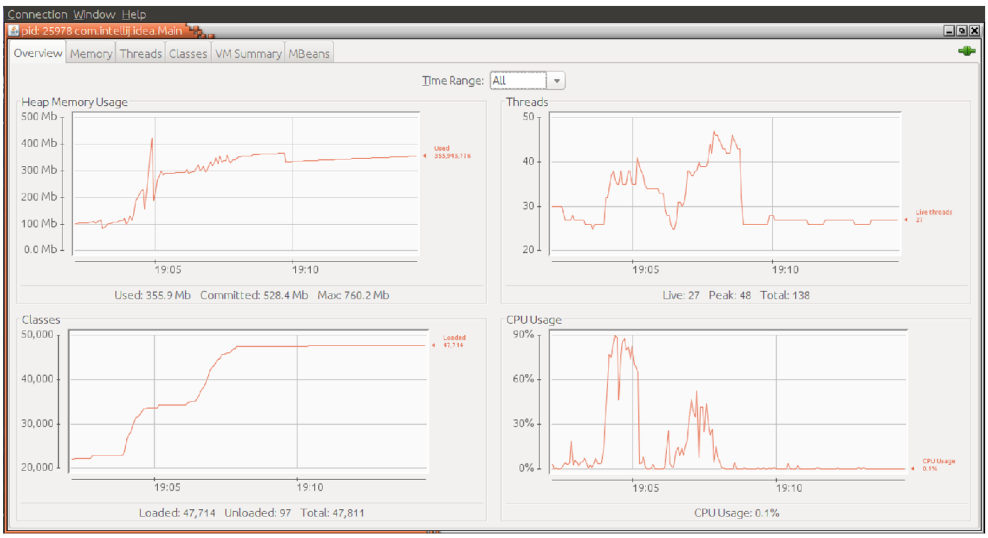


Figure 4.1: A JDK 8 JConsole tool main window. It shows usage of resources, other tabs contain more detailed information and a facility to manage object called *MBeans*.

4.1.2 GC Viewer

GC Viewer is a tool for analysing GC logs. Firstly, it is necessary to run a JVM with options `-Xloggc:`*logFile* `-XX:+PrintGCDetails-XX:+PrintGCDateStamps`, where *logFile* is a file where every GC event logs detailed information about just finished garbage collection. Secondly, open the *logFile* in the GC Viewer to assess effectiveness of 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[3] were developing GC Viewer till 2008. Since this time it is open source project[4] under GNU Lesser General Public License. The GC Viewer is being updated. [15] All support JVMs are in table B.1.

There is every GC event information stored in GC log file. The format of data depends on which *garbage collector* was used. However, we can always see time of start GC event, total heap usage before and after GC event as well as generation size and *metaspace* (usually when

3. http://www.tagtraum.com/gcviewer.html
4. Official repository is accessible at `https://github.com/chewiebug/GCViewer`

major GC was performed). There is an example of *minor* GC event log in figure 4.2. Separated parts have a following meaning:

- `2017-11-11T01:14:09.696+0100:` – *GC* evenet start timestamp.
- `67.671:` – *JVM* runtime in seconds.
- `[GC (Allocation Failure)` – cause of the *garbage collection*.
- `[PSYoungGen:` – name of the used *garbage collector*. In this case *Parallel GC* was used.
- `1424K->464K(1536K)]` – a real usage of the *young generation*. before and after a GC event. A value in parenthesis means allocated memory.
- `4910K->4171K(5632K)` – a real usage of the whole *heap*. A value in parenthesis means allocated memory again.
- `0.0006171 secs]` – a *GC* pause in seconds.

```
2017-11-11T01:14:09.696+0100:67.671:[GC(AllocationFailure)
[PSYoungGen:1424K->464K(1536K)]4910K->4171K(5632K),0.00061
71secs][Times:user=0.00sys=0.00,real=0.00secs]
```

Figure 4.2: A *minor* GC event log. The *garbage collection* occurred because there wasn't any sufficient piece of memory to allocate a new object into the *eden* area.

Although *GC Viewer* provides export, no format is suitable for further statistical processing. One of them contains only average values, others contain timestamps with a heap usage but without a generation distinction or without a *metaspace* usage. Not only for these reasons a tool for processing all measured data has been developed. The tool is called *ProcessData* and it is introduced in section 4.1.5. There are other analytic tools described in next two sections.

4.1.3 top

The *top* tool provides real-time monitoring of a running system. It can display summary information and/or a list of processes or threads being managed by operating system. CPU usage, amount of allocated memory or swap can be measured. By default, this tool displays information in terminal every second. There are options which can adjust refresh time (`-d` option) or starts *top* in *batch* mode (`-b` option) which is useful for sending output to a file. Thanks to this we can collect CPU usage of running *JVM* as well as system services and measurement tools. The command `top -b -d 0.2 » logFile.log` ensures logging all necessary data for further processing into the *logFile.log* file. [14]

4.1.4 iotop

The output of this tool is similar to the one provided by the previous tool. However the *iotop* displays IOPS[5] and waiting on I/O for every process or thread. Analogously, delay between two samples can be set (`-d` option again). It is possible to reduce output to processes or threads that are performing I/O (`-o` option) and add timestamps to output (`-t` option). By default, *iotop* measures IOPS per threads but we switched it to collect information about running process (`-P` option). For logging into *logFile.log* the command `iotop -P -d 0.2 -o -t » logFile.log` was used.

4.1.5 ProcessData

When we log running JVM performance characteristics a large amount of data is created. However, form of these data isn't suitable for direct evaluation. The *ProcessData* tool has been developed for this purpose. It supports a processing of all log files into form that is usable for a statistical evaluation (tool provides output into the `*.csv` file). The *ProcessData* is written in *Java* programming language and offers console interface with using one of the next arguments:

- g – for processing GC log files. In comparison to available *GC Viewer* exports a memory usage is processed in a more detailed way.

5. Input/output Operations Per Second

- `t` – outputs the CPU usage of running JVM, tools and system (each of them to it´s own file) with timestamps.
- `i` – similar to previous option with the difference to process IOPS.
- `o` – processes output of *Java* application (it's useful when application outputs some data suitable for evaluation, e.g runtime).
- `a` – processes all kinds of logs together.

The second argument is a list of folders (separated by space) to process. Folders should be located in `logs/` folder (a relative path) and contain logs. The *ProcessData* will output data into `data/subFolder`, where subFolder names are same as in the input list.

For example, the command `javaProcessDatagfolder01folder02` processes GC log files which are stored in `logs/folder01` and `logs/folder02` folders and the output with the heap usage information saves into `data/folder01` and `data/folder02` folders.

The source code of *ProcessData* (`ProcessData.java`) application is a part of an electronic appendix in the `tools` folder.

4.2 Automatic optimization tools

This section of thesis deals with tools which are usable for automatic tuning of JVM options.

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* framework and only the Linux environment is supported (for more information see next section 4.2.2). The aforementioned tool offers the console mode and the Graphical User Interface as well. Authors highly recommend using 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. [16] [5]

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 the 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. [18]

The *JATT* does not offer an online auto-tuning of JVM yet, but it should be in development according to information on official website. However, the website has not been updated for longer time. Author of this thesis has asked *JATT* developers for more information about current phase of the project but with no response[6].

The *JATT* tool is written in Python programming language[7] and except for the OpenTuner, it requires other packages. For a comprehensive information about requirements, installation and usage see website [4]. There is 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 our own set of options to auto-tune. All options are stored in `Flags` folder in `csv` files, including a `Temporary` file which is predestined to input our own list of options which the JATT will use to auto-tune a JVM.

Because the available *JATT* version aims to Java 7 it was necessary to update options in every group. When we compare JVM 7 update 80 with JVM 8 update 144 there are 104 new options in JVM 8 and 23 options became unsupported. One of the reasons is an adjusted memory management that section 2.3 describes.

The *JATT* is a student project which originated in 2012 at the University of Moratuwa, Sri Lanka. The project won the gold medal award at Association for Computing Machinery (ACM) student research competition attached to 2015 International Symposium on Code Generation and Optimization. [16]

---

6. E-mails with question have been sent to Saman Amarasinghe, one of project supervisors and Milinda Fernando, the team member.
7. Official repository is located at `https://bitbucket.org/sapients/hotspottuner/`

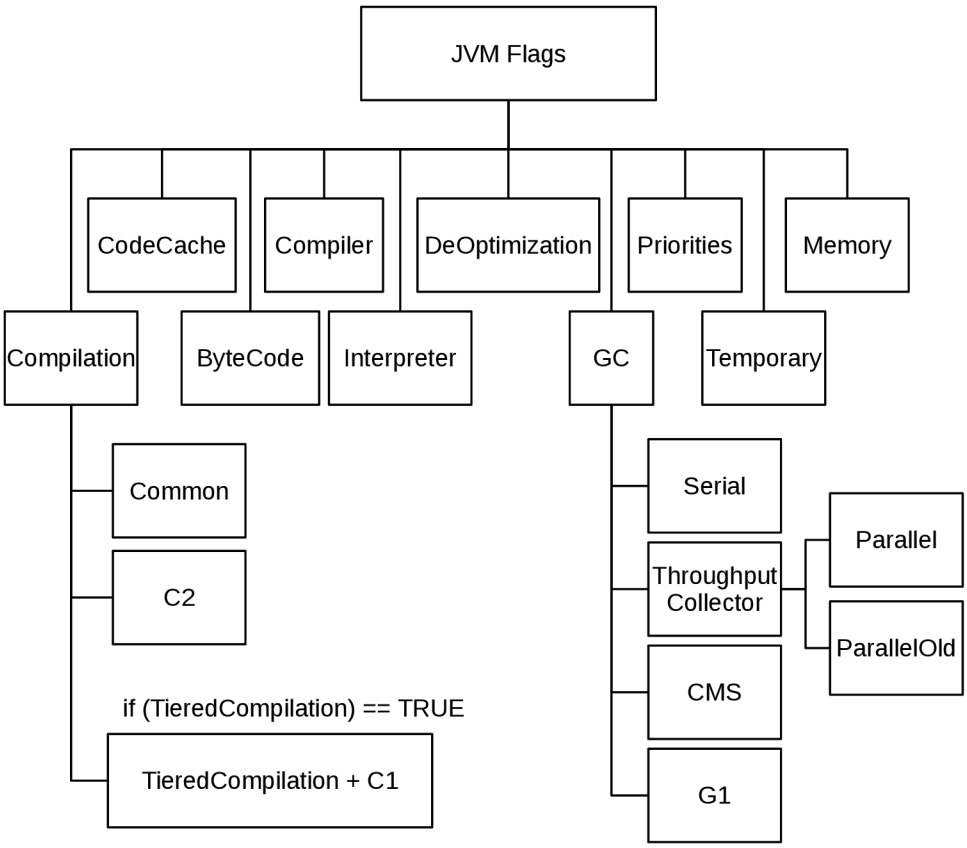


Figure 4.3: A JVM options hierarchy. The JATT divides JVM options into several groups. The division aims to reduce a searching space. But it is possible to use two or more groups together as well. [25] [4]

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* – it is possible usage of 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 a command with same *fileWithResults* repeatedly runs, 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 there is an example of the *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: $Improvement = Default\ metric/Runtime$ [5], 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: 1.10242196922
Runtime 90381
Default metric 99638.0
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) (shorten here) and improvement factor are the most important information.

4.2.2 OpenTuner

The *OpenTuner* is a framework for building domain-specific multi-objective program auto-tuners. *OpenTuner* supports an extensible technique representation to allow domain-specific techniques, fully-customizable configuration representations and an interface for communicating with the program to be autotuned. [17]

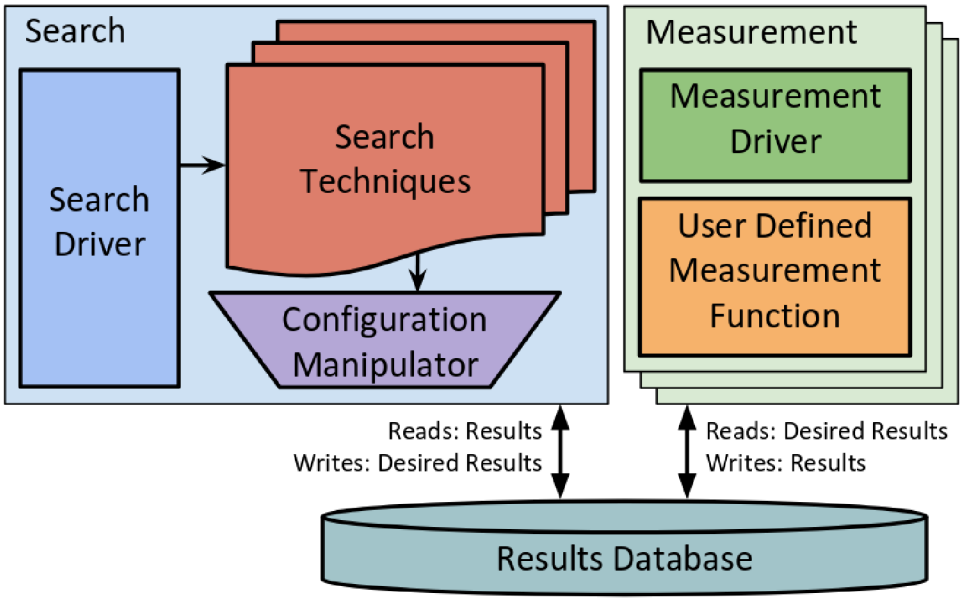


Figure 4.5: The *OpenTuner* framework components and a result storing into a database for a recurrent usage. Taken from [17].

One of the big challenges for *OpenTuner* is dealing with the landscape of a configuration space. That can be monotonic function, discontinuous, haphazard or high dimensionality. Parameters can be strongly coupled or independent from each other. Therefore the *OpenTuner* combines several techniques for reaching the best result. Moreover, when one of the techniques is more successful for a given application, the *OpenTuner* dynamically dedicates more resources for this technique, what should result into faster searching for optimal configuration. The *OpenTuner* uses next techniques: [17]

- *AUC Bandit Meta Technique (multi-armed bandit with sliding window, area under the curve credit assignment)* – it is *OpenTuner's* core technique
- *Nelder-Mead search*
- *Round Robin*
- *Torczon hillclimbers*

- number of evolutionary mutation
- pattern search, etc.

Additionally, already mentioned techniques share configurations in a database (which is apparent from the picture 4.5). In this picture there are two major *OpenTuner* components which communicate with each other through the database. This concept allows parallelism between multiple search measurement processes, possibly across different machines. The *configuration manipulator* provides a layer of abstraction between the search techniques and the raw configuration structure – the JVM options in our case. [17]

4.2.3 ParamILS

Philipp Lengauer and Hanspeter Mössenböck deal with automatic parameter tuning for Java Garbage Collectors in [21]. They introduced a tool extensively based on *ParamILS* framework for JVM tuning in 2014, primarily for improving GC operations. The tool is practically short bash script which launches *ParamILS* with parameter model – options for tuning divided by *garbage collector* algorithms in ParamILS-specific syntax. The tool itself 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. There is an annual charge $5000 for commercial purposes per company[8]. [21] [20] The *ParamILS* is a collection of easy readable Ruby scripts and both tools support Windows and Linux operating systems[9].

The *ParamILS* is an abbreviation for *Parameter Iterated Local Search*. It is based on a configuration model and a local optima searching. It always considers changing only one parameter at a time. This approach is very similar to *hill climbing* technique. Additionally, it consists methods for a *solution perturbation* to escape from local optima, because when *ParamILS* reaches the local optima it may not be a global maximum. [20] There is a piece of a configuration model in figure 4.6.

8. This information has been gained from an e-mail communication with Chris Fawcett.
9. `http://www.cs.ubc.ca/labs/beta/Projects/ParamILS/ParamILS-Quickstart.pdf`

```
...
XX:MaxHeapFreeRatio 60, 70, 80, 90 [70]
XX:MinHeapFreeRatio 10, 20, 40, 50 [40]
XX:MinSurvivorRatio 1, 2, 3, 6 [3]
...
```

Figure 4.6: A pattern of the *ParamILS* configuration model. The value in a pair of square brackets is a default value and the framework searches for a better setting which will arise from a possible values in curly brackets.

The model splits JVM options into several groups based on *GC* algorithms. Lengauer and Mössenböck focused their work on *garbage collection* tuning Java 7. The configuration model became outdated and unsatisfactory. For purposes of this thesis it had to be updated.

There are the *ParamILS* tool as well as the bash script for a JVM autotuning in an electronic appendix of this thesis in a `tools/paramILS` folder. For starting the autotuning process type command: `bash run.sh` "*name*" "*timeout*" "*java*" "*fixedOptions*" "*model*" "*command*" where[10]:

- *name* – a folder name where results will be stored.
- *timeout* – a timeout for the optimization process in seconds.
- *java* – a path to Java binary folder which will be used. Specify "java" or "/usr/bin/java" for default Java installation.
- *fixedOptions* – these options will not be optimized, e.g, "-XX:MaxHeapSize=10m" to limit heap size.
- *model* – configuration model file.
- *command* – the command to run your application without "java" call.

4.2.4 Commercial tools

Among important non-freely accessible tools belong:

10. Adjusted from a `README` file.

- Arcturus Applicare Tune Wizard[11] – comprehensive tool for automated performance tuning, intelligent monitoring, optimization and instant root cause analysis. Applicable for great systems such web applications. It costs $2000 per year and one JVM[12].

- Dynatrace – this company is a leader in application performance monitoring and tuning for Java and .NET applications. The price per license depends on amount of memory. For example, the license for machine with 16 GB memory costs $0.35 per hour[13]. Dynatrace collaborates with Philipp Lengauer and Hanspeter Mössenböck on JVM tuning. [1]

- Plumbr[14] – it costs $83 per month and one JVM[15].

11. `http://www.arcturustech.com/tunewizard.html`
12. The information was obtained by mail communication with Ridhi Khurana, Sales Director, Arcturus Technologies, Inc.
13. `https://www.dynatrace.com/pricing/#price-self-service`
14. `https://plumbr.io/`
15. https://plumbr.io/pricing

5 JVM settings optimization

Next chapter deals with optimization of JVM setting. Java applications which were tuned by *JATT* and *ParamILS* tools are introduced. There are metrics and measurement methodology defined. Finally, results are presented and discussed.

5.1 Application

A *xalan* benchmark has been chosen for optimization of JVM settings for application performance. For additional optimization, *batik* and *h2* benchmarks were used. All the applications are parts of well known *DaCapo* benchmark suite[1] (9.12 version) which is intended to be a tool for Java benchmarking. One of aspects for choosing *xalan* was high CPU usage. There is an average CPU usage in the picture **??**. These applications process data in the following domains: [2]

- **xalan** – transforms *XML* documents into *HTML* (17 files).
- **batik** – is based on *Apache™ Batik SVG Toolkit*[2] and produces a set of Scalable Vector Graphics images.
- **h2** – performs 4000 transactions to an in-memory DB. It simulates a banking system.

Each of the *DaCapo* benchmark is executable in more iterations and just the last iteration is a benchmark score. By default, score is a runtime of the iteration but it can be adjusted to the different metric by creating its own *Callback*. Previous phase is called *warm up* and it serves mainly for code optimizing (described in section 2.2) and classes loading. However, every iteration prints a benchmark score to standard output.

5.2 Metrics

The *xalan* benchmark has chosen a total runtime of 10 iterations for a main metric in the primal phase. However, this metric became in-

1. `http://www.dacapobench.org/`
2. `https://xmlgraphics.apache.org/batik/`

Benchmark	Heap size	Iterations	Runtime
xalan	5 MB	20	total
xalan	5 MB	20	last iteration
xalan	10 MB	20	total
xalan	10 MB	20	last iteration
xalan	15 MB	20	total
xalan	15 MB	20	last iteration
xalan	20 MB	20	total
xalan	20 MB	20	last iteration
batik	80 MB	20	total
batik	80 MB	20	last iteration
h2	300 MB	10	total
h2	300 MB	10	last iteration

Table 5.1: Metrics which were used to performance optimization and performance improvement determination.

applicable because of two reasons. First, the amount of iterations was not sufficient for measurement. Second, the heap size has grown to large size. For these reasons all of metrics limit the *heap* size (e.g., `-XX:MaxHeapSize=5m` for 5 MB). The runtime is examined in two points of view:

- **Total runtime** – the runtime of whole application is determinative and every iteration counts into result.
- **Last iteration runtime** – long running application can afford a slower initial phase (*warm up* phase in *DaCapo* context) which will lead into the faster runtime of next phases.

The table 5.1 shows runtime metrics.

Other used metric is *accumulated GC pause*. It represents the part of runtime which *JVM* spends by performing *minor* or *major GCs*. This value has not been tuned directly, but it was observed in dependency to previous metrics tuning.

5.3 Measurement methodology

A machine with following parameters was used for a measurement:

- CPU – Intel® Core™ i5-2450M @ 2.5 GHz
- Memory – 8 GB
- Operating system – Ubuntu 16.04, kernel Linux 4.10.0-33-generic (x86_64), console mode

There were 100 runs carried out for every selected *DaCapo* benchmark with a default JVM settings and the heap size restriction. After that one of the tools was used for optimized JVM settings acquirement (each of autotune processes took 8 hours). Finally, new JVM settings were used for 100 *DaCapo* benchmark runs. For minimization of ambient influences the console mode of operating system was used.

5.4 JATT

The JATT provides an interface to autotune *DaCapo* benchmarks (`src/dacapoTuner.py`). But it takes the last iteration runtime for metric only (it processes standard error output which is printed by *DaCapo* benchmark). Therefore this interface was adjusted for our metrics (e.g, `src/dacapoTunerRuntime20it20m.py` for 20 iterations with 20 MB heap size limit). The rest of parameters is the same as we described in section 4.2.1. The `-iterations` parameter has a different meaning for *JATT* tool and *DaCapo* benchmark suite. Thanks to the interface we tuned options from *gc*, *compiler* and *temporary* groups.

5.5 ParamILS

While the *JATT* is reading the *DaCapo* standard error output the *ParamILS* gets results by *callbacks*. The *Callback* is a *Java* class which extends *DaCapo callback* and defines its own metric – score. Because the original *ParamILS* focuses on a *garbage collection* tunning the *callback* returns sum of *GC* pauses. In accordance to our main metrics the *DaCapoRuntimeCallback* has been defined (a source code is in figure B.2).

Figure 5.1: Total runtime has been optimized for *xalan* benchmark with 20 MB heap size. Optimized settings found by *JATT* led to shorter runtime by 10 seconds (it is approximately 25 %).

5.6 Results

This section is destined for measurement results presentation. Mainly observed metrics are evaluated in *barplots* combined with *boxplots*. How to read these charts is described in figure B.1. Other performance characteristics are presented too (CPU usage, IOPS or heap occupation). The most interesting results are presented and discussed below. The rest is located in appendix in the chapter C.

For all tuned applications, the heap size was reduced. Consequently, the *JVM* has been forced to perform *GCs* more frequently. However, it is possible to reduce *GC* overhead by optimization *JVM* settings. The figure 5.4 shows reducing time which is necessary to *garbage collection* for *h2* benchmark. It has been reduced four times.

In opposite, setting of *compilation* options tuning were not to successful. Mostly, it led to improvement by several percent. In some cases it led to performance degradation.

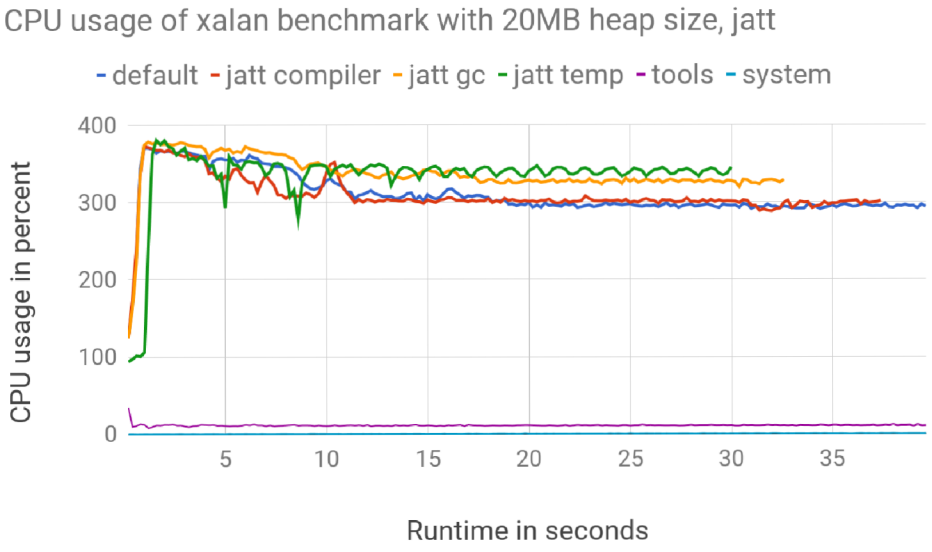


Figure 5.2: CPU usage has increased for tuned configuration.

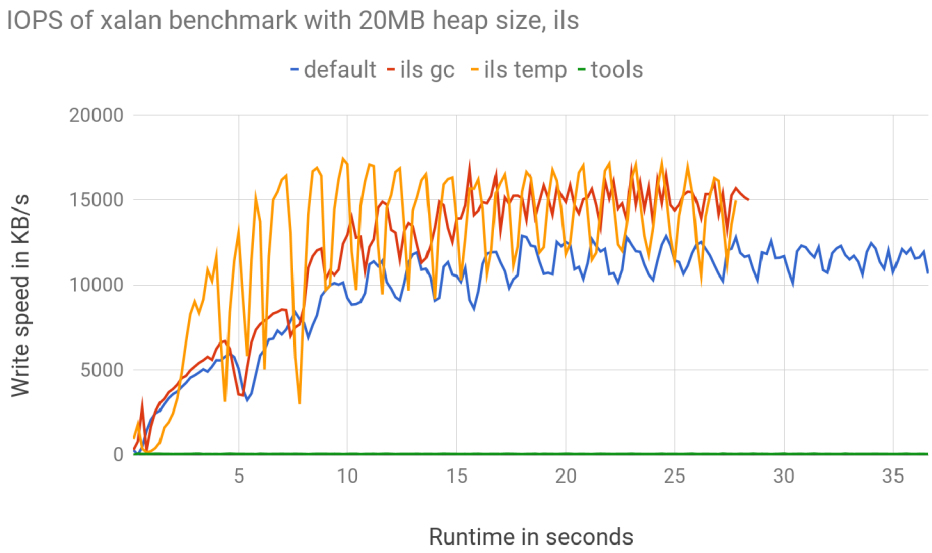


Figure 5.3: Xalan benchmark transforms many files. The *IOPS* improvement has been expected.

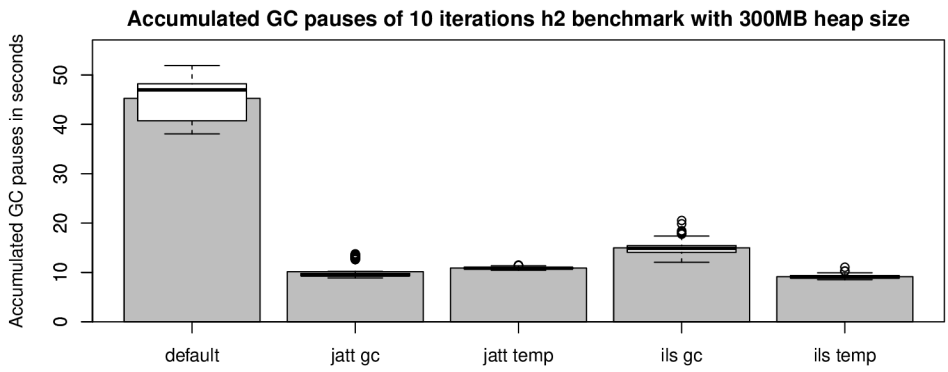


Figure 5.4: GC pauses has been dramatically reduced for *h2* benchmark for all tuned configurations.

6 Conclusion

This thesis deals with optimization of JVM settings. There is described important *JVM* techniques in the beginning. The options were selected and using autotuning tool the optimization is performed.

Many measurements as been obtained which are presented in last chapter. All the data is attached in the electronic appendix. There are many charts in the appendix of this thesis.

We focused on small heap sizes. For this reason mainly a *garbage collection* improvement was reached. It led to reduction in runtime of application. In the best case by 25 percent.

The further research could be focused on application with big heap size. Due to settings optimization of advanced *garbage collectors* for performance (e.g., web server). The other possible continuation is a comparison with the latest version of Java (recently released version 9) that should have a better self-tuning and optimization.

Bibliography

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

[2] "The DaCapo benchmark suite," http://www.dacapobench.org/, [Online; visited 20-October-2017].

[3] "Google is adding Kotlin as an official programming language for Android development," https://www.theverge.com/2017/5/17/15654988/google-jet-brains-kotlin-programming-language-android-development-io-2017, [Online; visited 11-December-2017].

[4] "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].

[5] "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].

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

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

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

[9] "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].

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

[11] "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].

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

[13] "TIOBE Index," https://www.tiobe.com/tiobe-index/, [Online; visited 11-December-2017].

[14] "top - Linux manual information."

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

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

[17] J. Ansel, S. Kamil, and K. Veeramachaneni, "Opentuner: An extensible framework for program autotuning," *Massachusetts Institute of Technology*, pp. 3–10, 2014.

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

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

[20] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Paramils: An automatic algorithm configuration framework," *Journal of Artificial Intelligence Research*, vol. 36, pp. 267–306, 2009.

[21] 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.

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

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

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

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

A Electronic appendix information

There are several folders in an electronic appendix:

- `application` – *DaCapo* suite JAR
- `results` – all measured data processed by `ProcessData` application. There is a `flags.txt` file in every folder with optimized options.
- `tools`
  - `jatt` – *JATT* tool
  - `paramILS` – *ParamIlS* tool
  - `ProcessData` – *ProcessData* tool

B Additional information

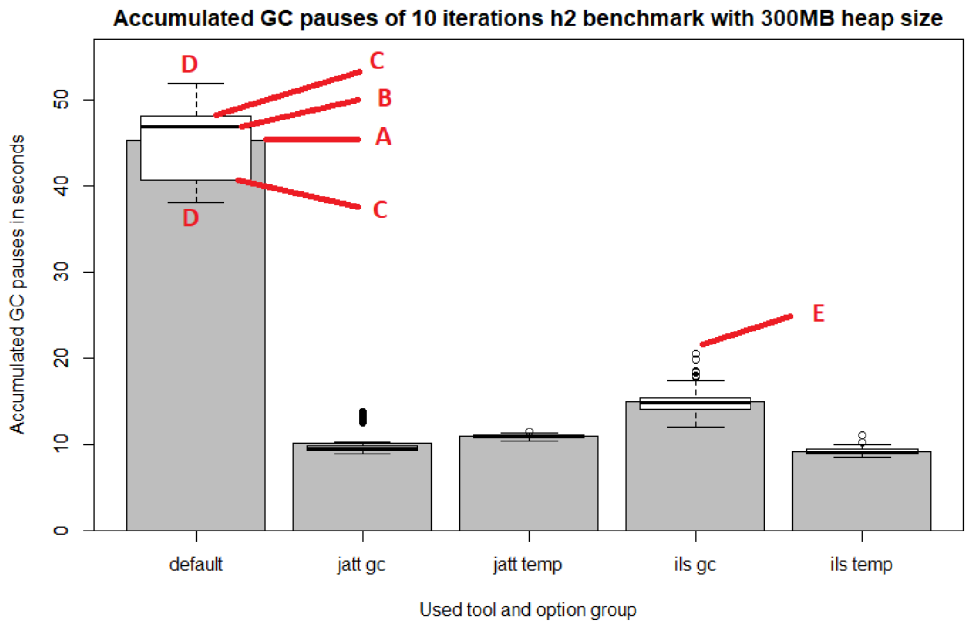


Figure B.1: Results are presented in combination *barplot* with *boxplot*. Important are following sections: A) mean of result set, B) median of result set, C) first and third quartile, D) minimum and maximum, E) circles are values out of this range.

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 B.1: A list of JMVs that the GC Viewer supports. `https://github.com/chewiebug/GCViewer`

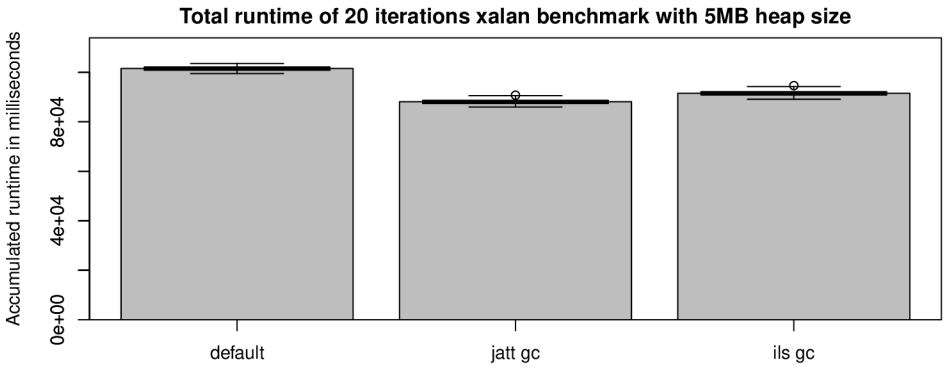
-XX:Inline	-XX:UseParallelGC
-XX:NewRatio	-XX:UseG1GC
-XX:BackgroundCompilation	-XX:UseConcMarkSweepGC
-XX:CICompilerCount	-XX:BiasedLockingStartupDelay
-XX:SurvivorRatio	-XX:GCHeapFreeLimit
-XX:CompileThreshold	-XX:GCTimeLimit
-XX:DoEscapeAnalysis	-XX:GCTimeRatio
-XX:UseParallelOldGC	-XX:InitialSurvivorRatio
-XX:UseSerialGC	-XX:MaxHeapFreeRatio
-XX:AllocatePrefetchDistance	-XX:MinHeapFreeRatio
-XX:MinSurvivorRatio	-XX:ObjectAlignmentInBytes
-XX:ParallelGCThreads	-XX:OptimizeStringConcat
-XX:TieredCompilation	-XX:UseBiasedLocking
-XX:UseAdaptiveSizePolicy	

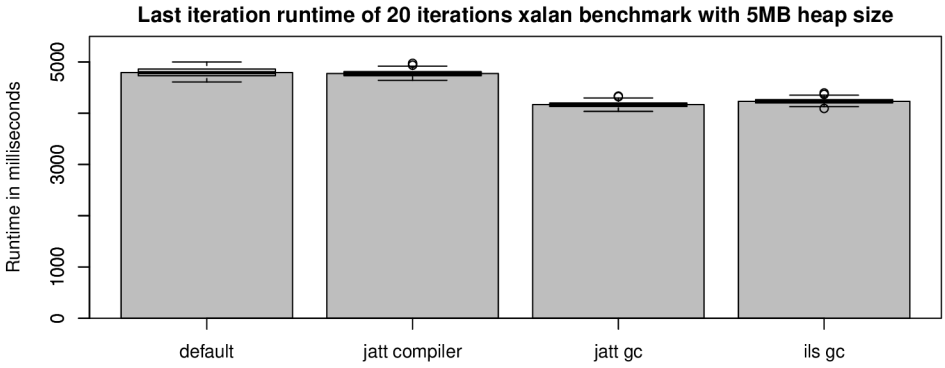
Table B.2: These options represent own set (*temporary*) and have been chosen for optimization.

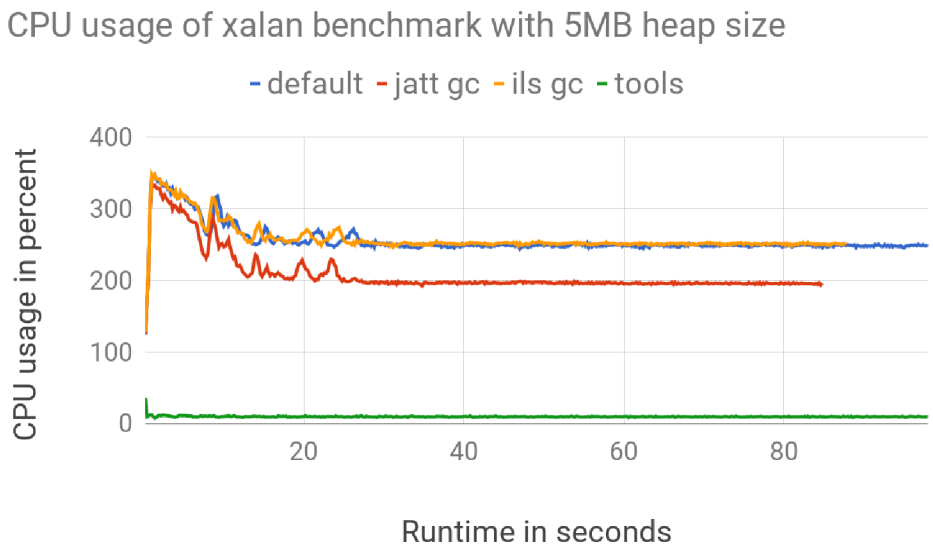
```
import org.dacapo.harness.Callback;
import org.dacapo.harness.CommandLineArgs;

public class DaCapoRuntimeCallback extends Callback {

    private long start;
    private long score = 0;

    public DaCapoRuntimeCallback(CommandLineArgs args){
        super(args);
    }

    @Override
    public void start(String benchmark) {
          start = System.currentTimeMillis();
          super.start(benchmark);

    }

    @Override
    public void stop() {
          super.stop();
          score += System.currentTimeMillis() - start;
          if (!isWarmup()) {
               System.err.println("##### score=" + score);
          }
    }
}
```

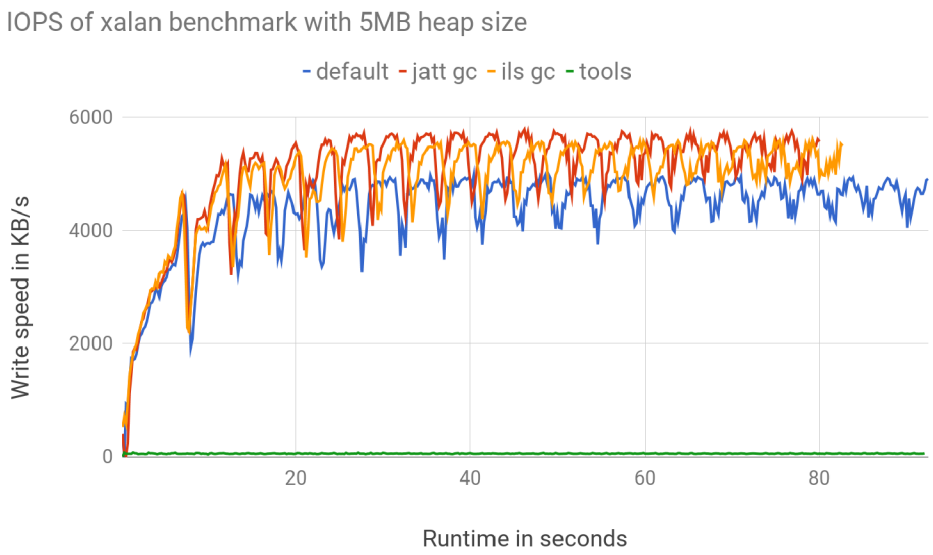
Figure B.2: The `DaCapoRuntimeCallback` measures total runtime of *DaCapo* benchmarks.

C Charts


Total runtime of 20 iterations xalan benchmark with 5MB heap size
Accumulated runtime in milliseconds
0e+00
4e+04
8e+04
default
jatt gc
ils gc

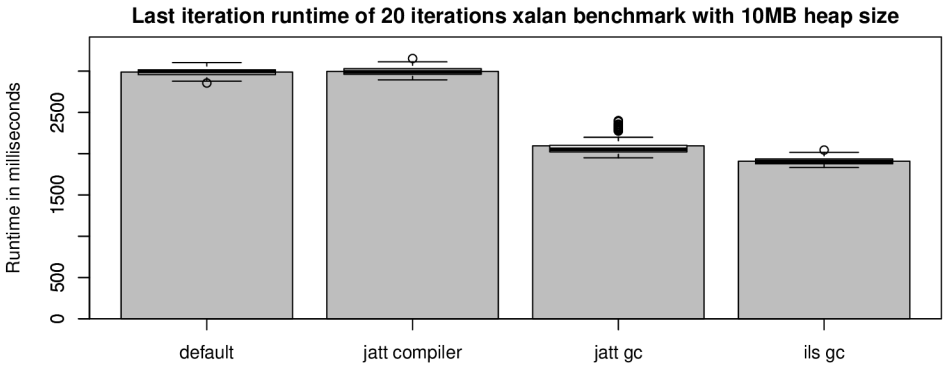

Last iteration runtime of 20 iterations xalan benchmark with 5MB heap size
Runtime in milliseconds
0
1000
3000
5000
default
jatt compiler
jatt gc
ils gc

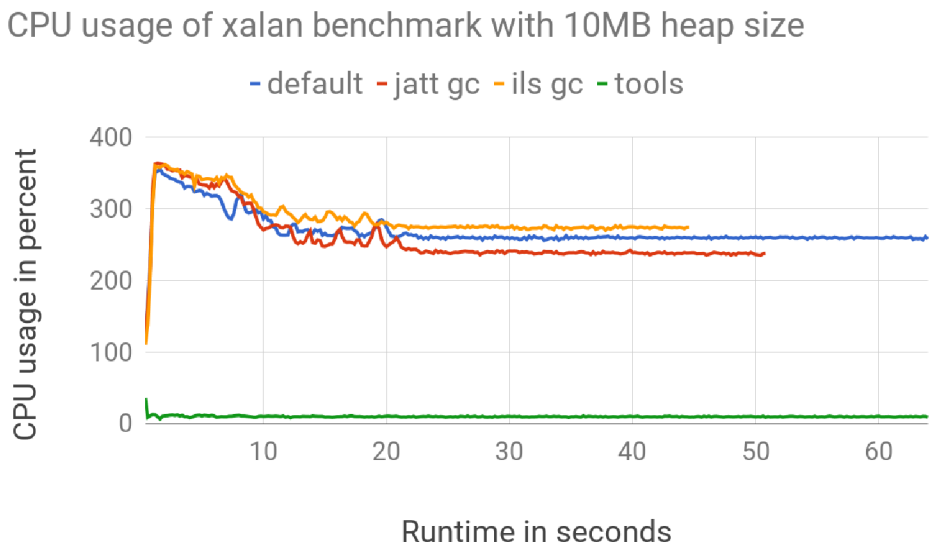

CPU usage of xalan benchmark with 5MB heap size
default
jatt gc
ils gc
tools
CPU usage in percent
400
300
200
100
0
20
40
60
80
Runtime in seconds

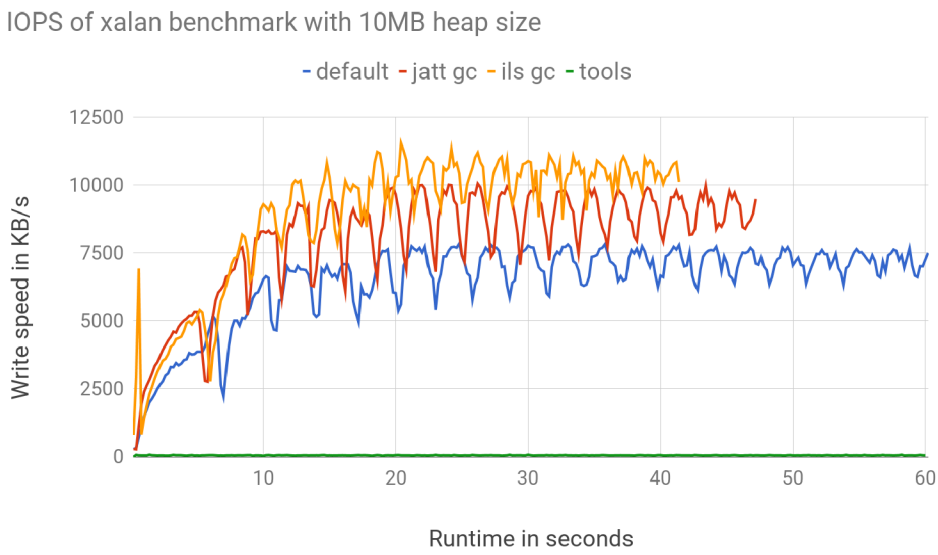

IOPS of xalan benchmark with 5MB heap size
default
jatt gc
ils gc
tools
Write speed in KB/s
6000
4000
2000
0
20
40
60
80
Runtime in seconds

Accumulated GC pauses of 20 iterations xalan benchmark with 5MB heap size
Accumulated GC pauses in seconds
0
10
20
30
40
50
60
70
default
jatt gc
ils gc

Total runtime of 20 iterations xalan benchmark with 10MB heap size
Accumulated runtime in milliseconds
0
20000
40000
60000
default
jatt gc
ils gc

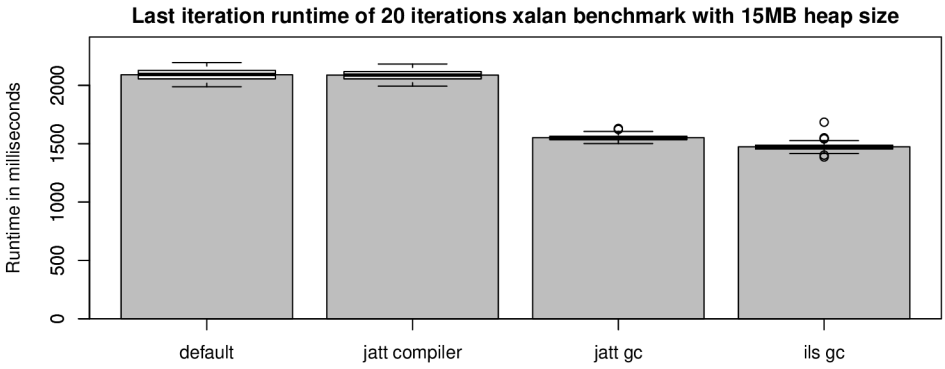

Last iteration runtime of 20 iterations xalan benchmark with 10MB heap size
Runtime in milliseconds
0
500
1500
2500
default
jatt compiler
jatt gc
ils gc

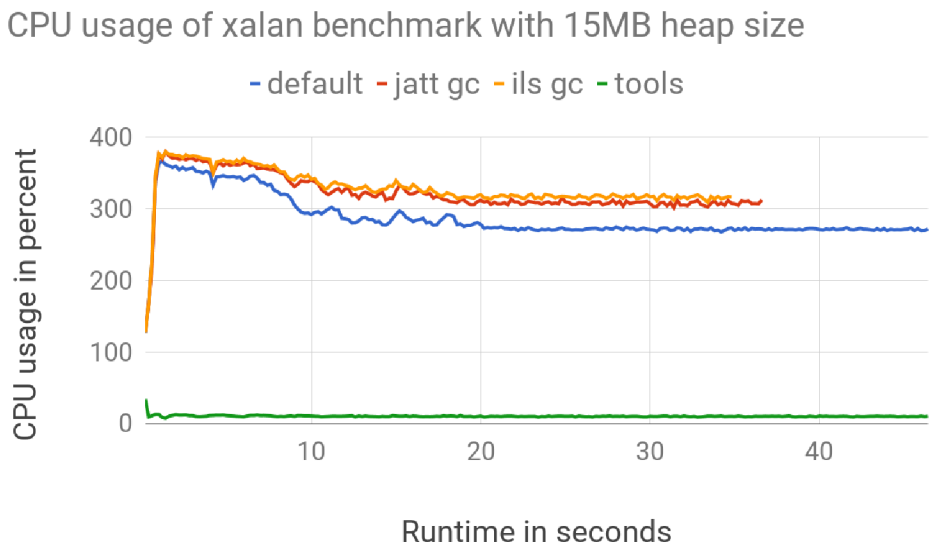

CPU usage of xalan benchmark with 10MB heap size
default
jatt gc
ils gc
tools
CPU usage in percent
400
300
200
100
0
10
20
30
40
50
60
Runtime in seconds

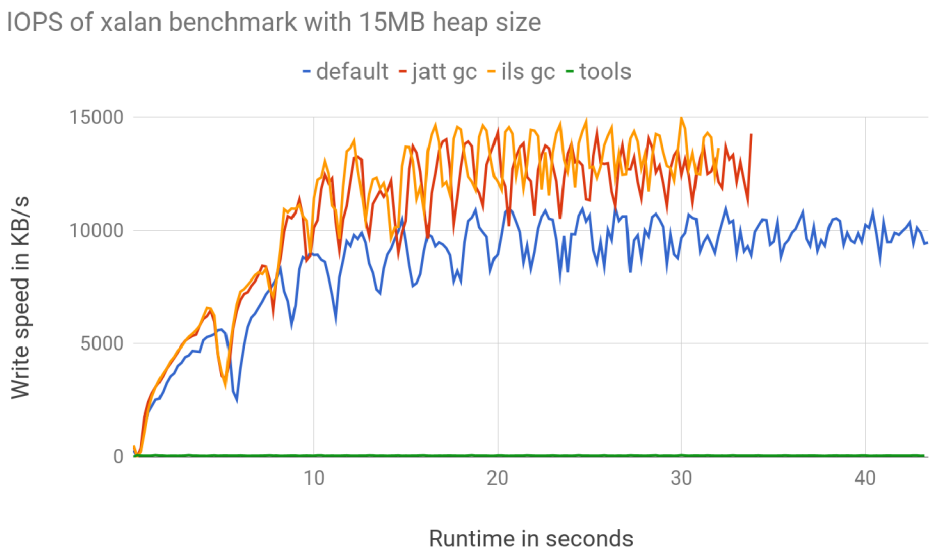

IOPS of xalan benchmark with 10MB heap size
default
jatt gc
ils gc
tools
Write speed in KB/s
12500
10000
7500
5000
2500
0
10
20
30
40
50
60
Runtime in seconds

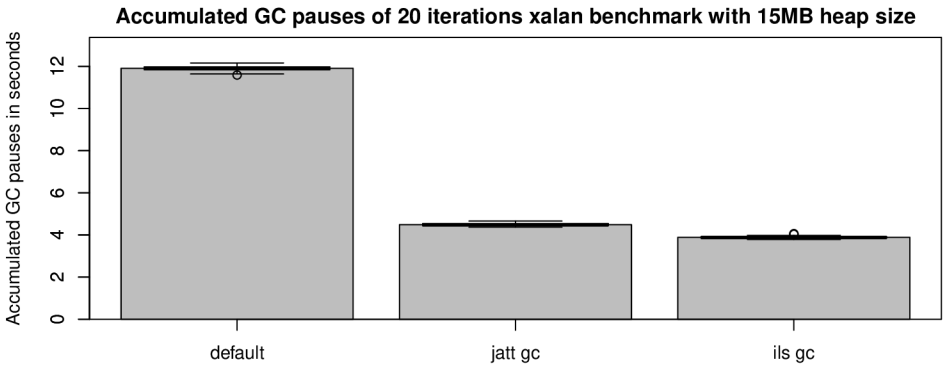
Accumulated GC pauses of 20 iterations xalan benchmark with 10MB heap size
Accumulated GC pauses in seconds
0
5
10
15
20
25
30
default
jatt gc
ils gc

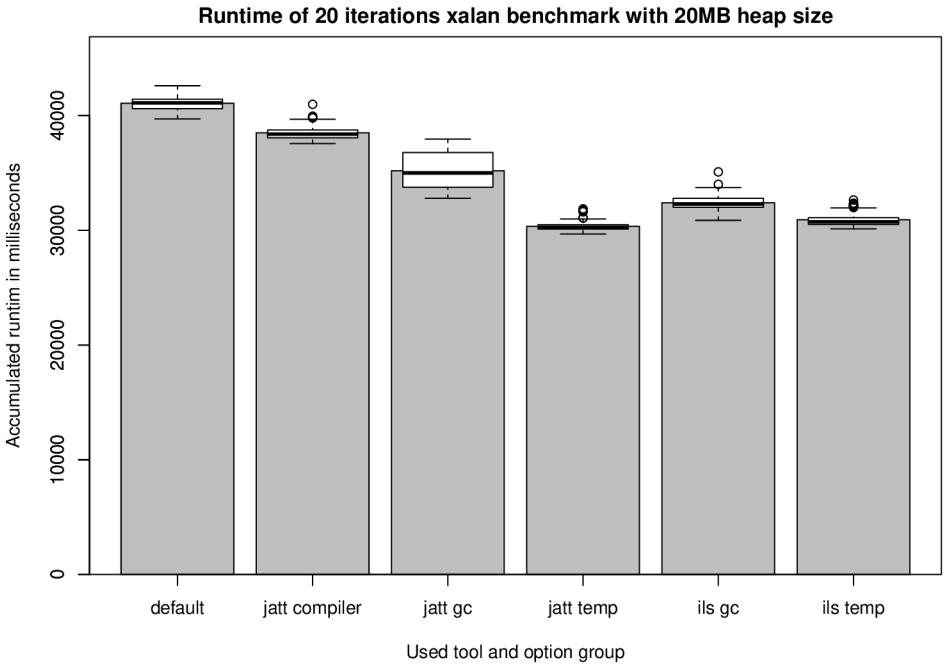
Total runtime of 20 iterations xalan benchmark with 15MB heap size
Accumulated runtime in milliseconds
0
10000
30000
50000
default
jatt gc
ils gc

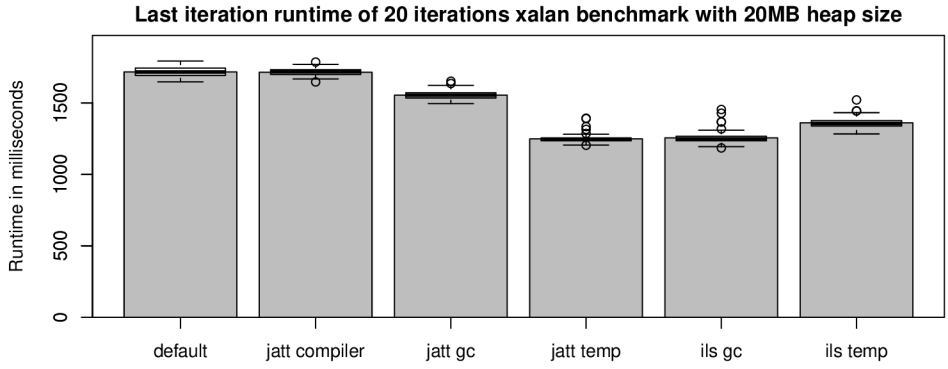

Last iteration runtime of 20 iterations xalan benchmark with 15MB heap size
Runtime in milliseconds
0
500
1000
1500
2000
default
jatt compiler
jatt gc
ils gc

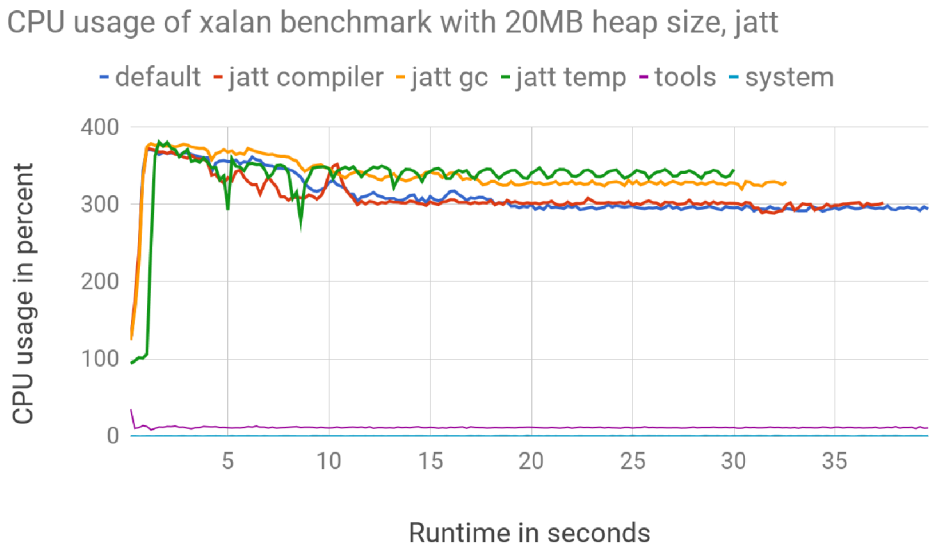

CPU usage of xalan benchmark with 15MB heap size
default
jatt gc
ils gc
tools
CPU usage in percent
400
300
200
100
0
10
20
30
40
Runtime in seconds


IOPS of xalan benchmark with 15MB heap size
default
jatt gc
ils gc
tools
Write speed in KB/s
15000
10000
5000
0
10
20
30
40
Runtime in seconds

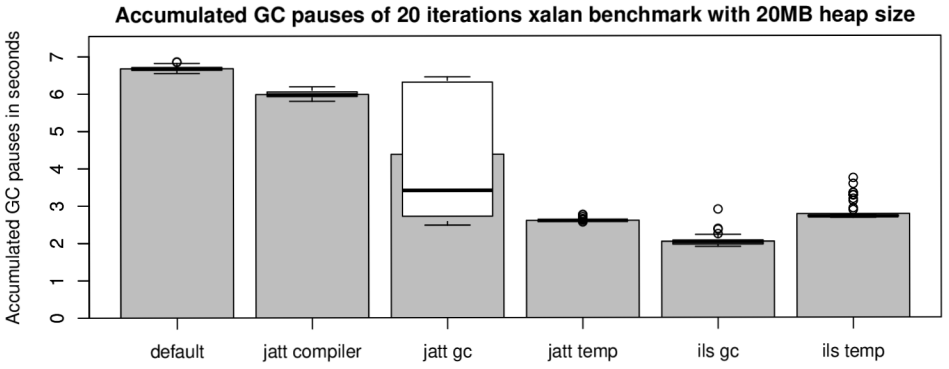

Accumulated GC pauses of 20 iterations xalan benchmark with 15MB heap size
Accumulated GC pauses in seconds
0
2
4
6
8
10
12
default
jatt gc
ils gc

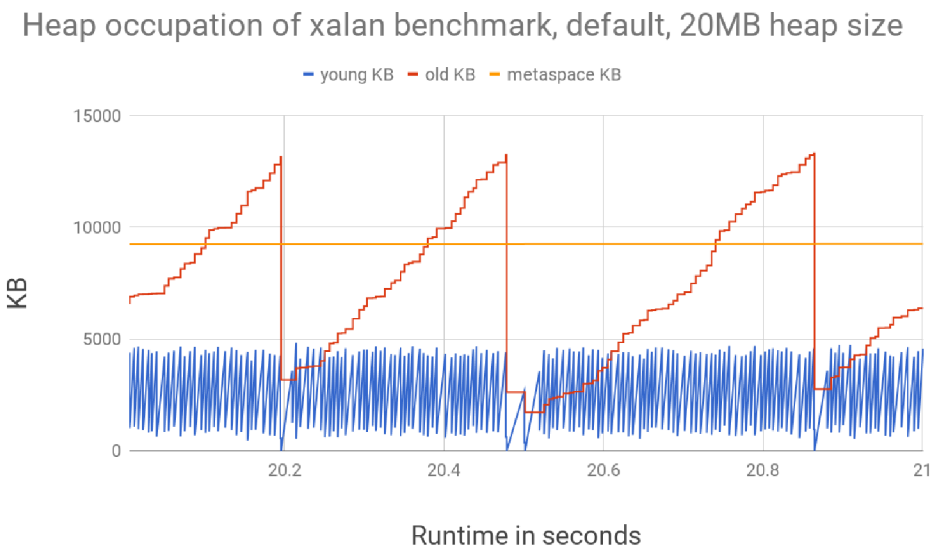

Runtime of 20 iterations xalan benchmark with 20MB heap size
Accumulated runtim in milliseconds
0
10000
20000
30000
40000
default
jatt compiler
jatt gc
jatt temp
ils gc
ils temp
Used tool and option group

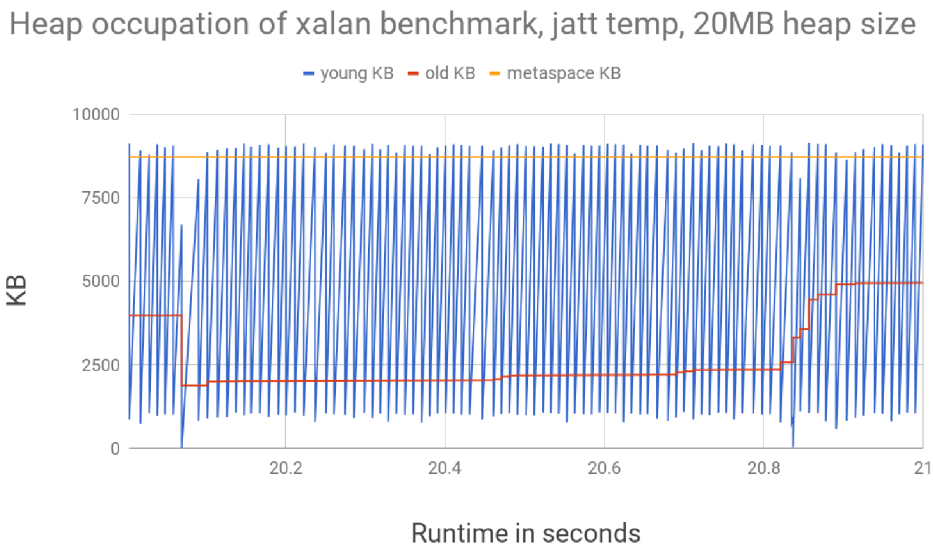

Last iteration runtime of 20 iterations xalan benchmark with 20MB heap size
Runtime in milliseconds
0
500
1000
1500
default
jatt compiler
jatt gc
jatt temp
ils gc
ils temp

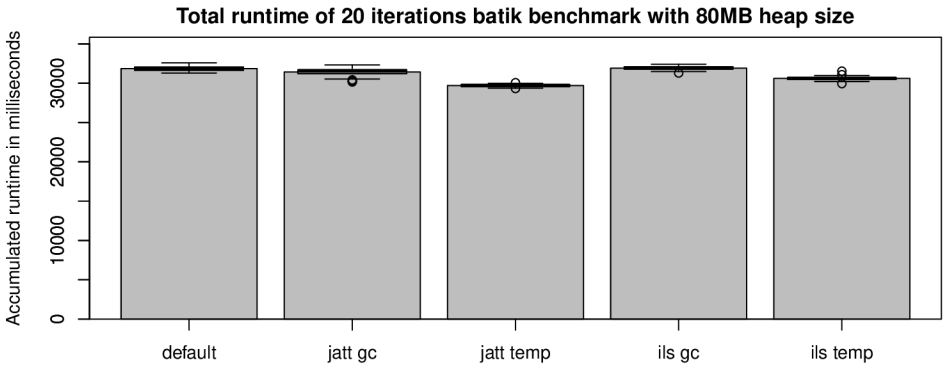

CPU usage of xalan benchmark with 20MB heap size, jatt
default
jatt compiler
jatt gc
jatt temp
tools
system
CPU usage in percent
400
300
200
100
0
5
10
15
20
25
30
35
Runtime in seconds

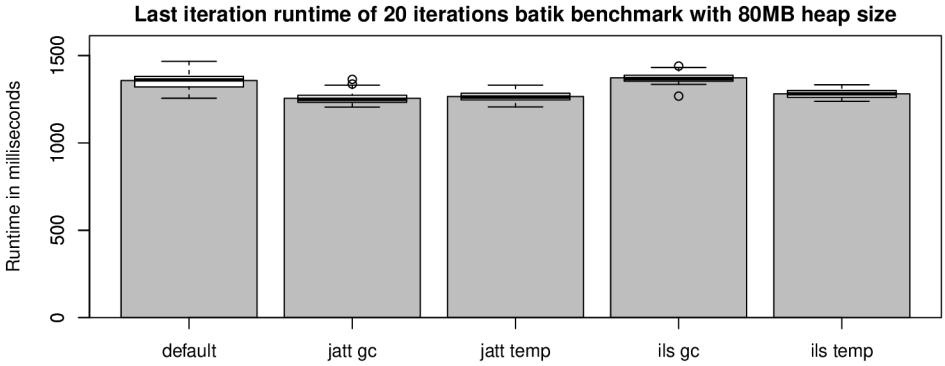
IOPS of xalan benchmark with 20MB heap size, jatt
default
jatt compiler
jatt gc
jatt temp
tools
Write speed in KB/s
20000
15000
10000
5000
0
5
10
15
20
25
30
35
Runtime in seconds

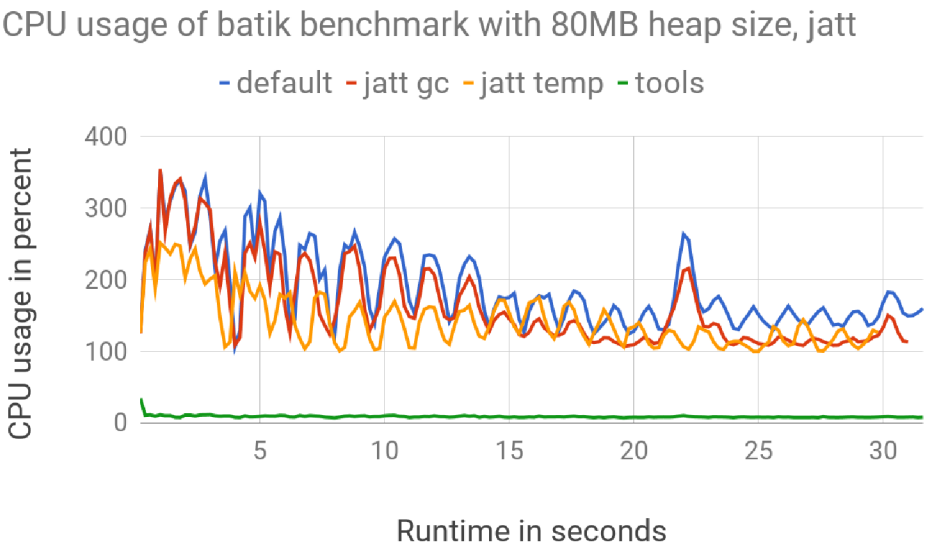

Accumulated GC pauses of 20 iterations xalan benchmark with 20MB heap size
Accumulated GC pauses in seconds
0
1
2
3
4
5
6
7
default
jatt compiler
jatt gc
jatt temp
ils gc
ils temp

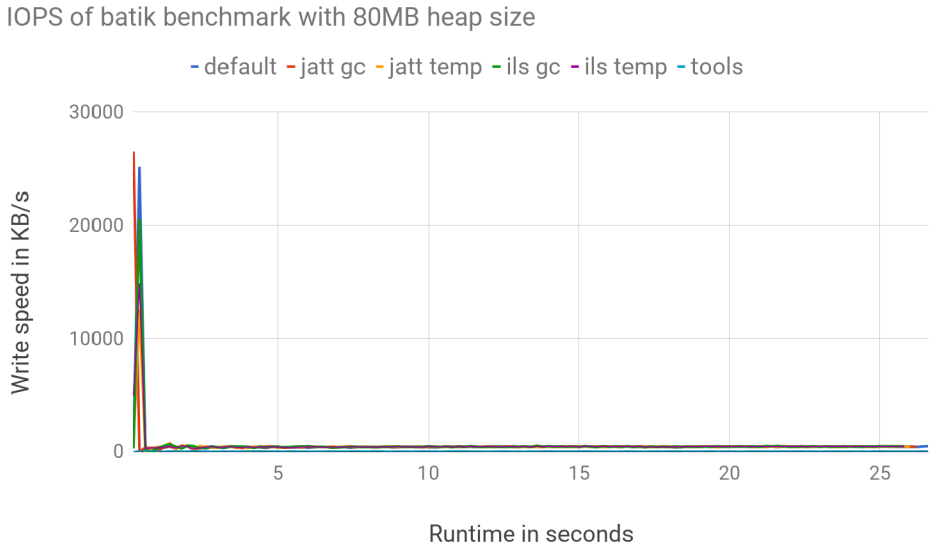

Heap occupation of xalan benchmark, default, 20MB heap size
young KB
old KB
metaspace KB
15000
10000
5000
0
KB
20.2
20.4
20.6
20.8
21
Runtime in seconds

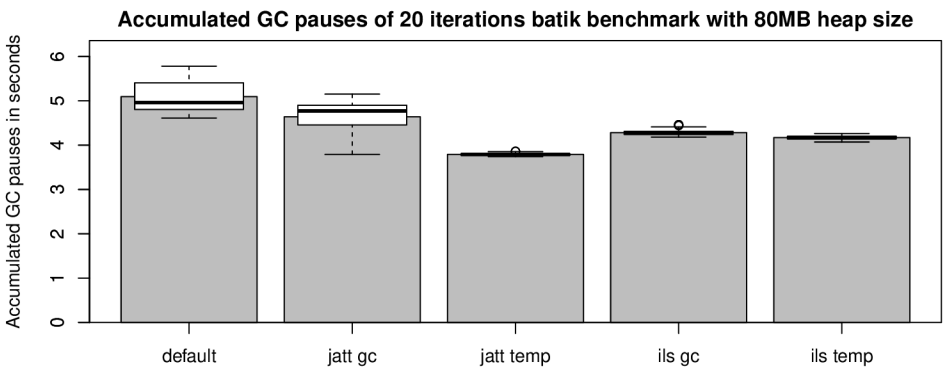

Heap occupation of xalan benchmark, jatt temp, 20MB heap size
young KB
old KB
metaspace KB
10000
7500
5000
2500
0
KB
20.2
20.4
20.6
20.8
21
Runtime in seconds

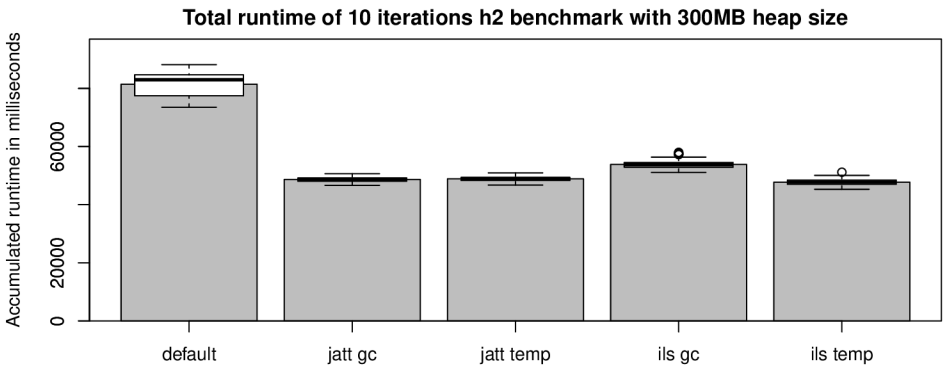

Total runtime of 20 iterations batik benchmark with 80MB heap size
Accumulated runtime in milliseconds
0
10000
20000
30000
default
jatt gc
jatt temp
ils gc
ils temp

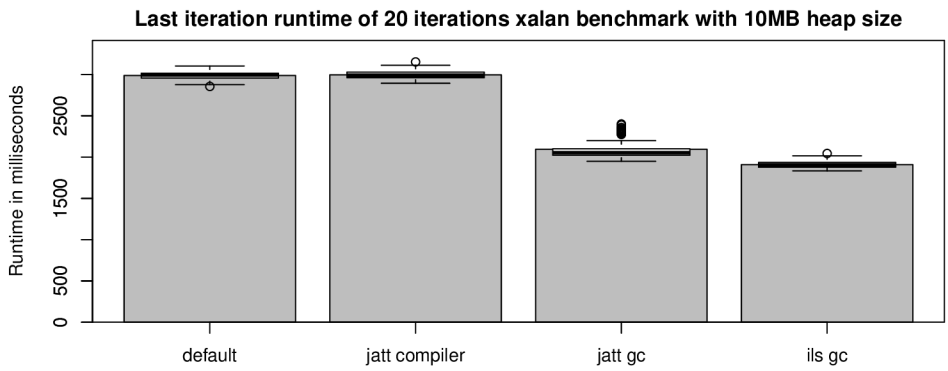

Last iteration runtime of 20 iterations batik benchmark with 80MB heap size
Runtime in milliseconds
0
500
1000
1500
default
jatt gc
jatt temp
ils gc
ils temp

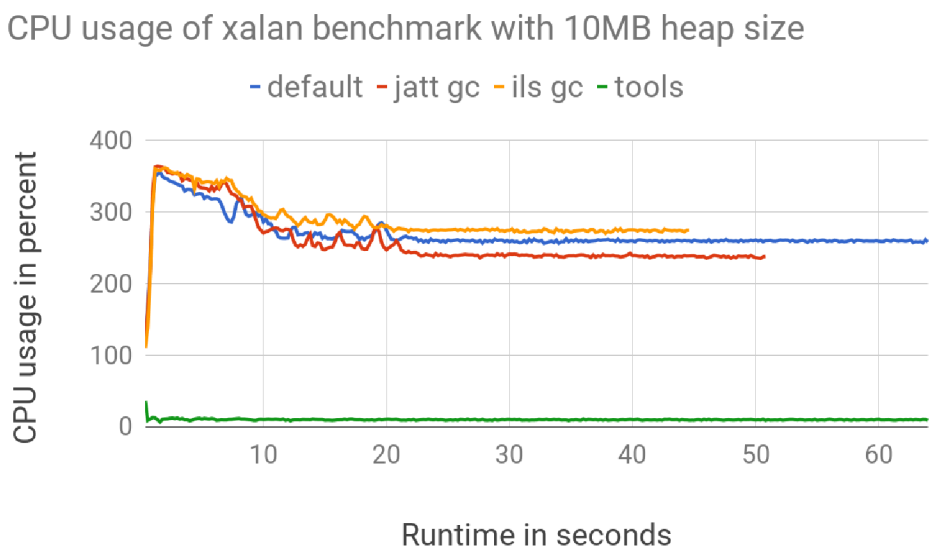

CPU usage of batik benchmark with 80MB heap size, jatt
default
jatt gc
jatt temp
tools
CPU usage in percent
0
100
200
300
400
5
10
15
20
25
30
Runtime in seconds

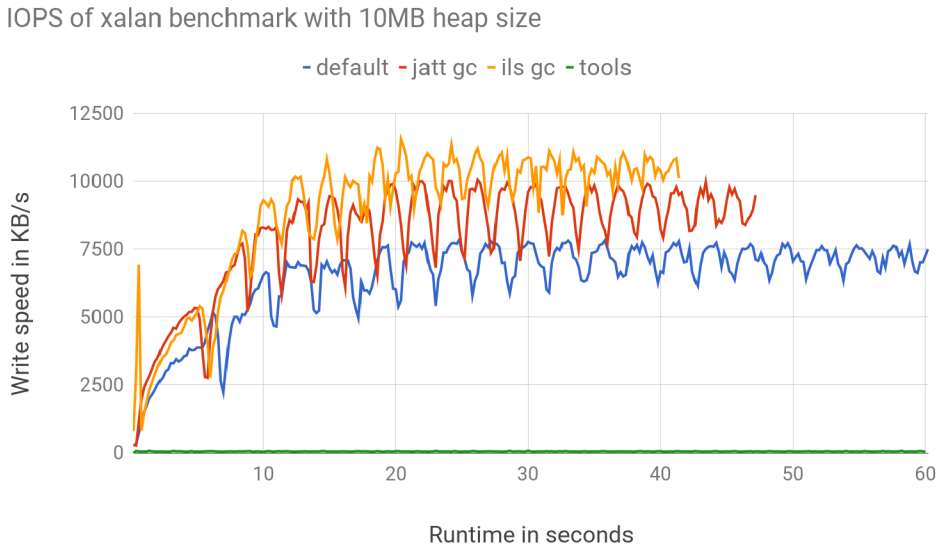

IOPS of batik benchmark with 80MB heap size
default
jatt gc
jatt temp
ils gc
ils temp
tools
Write speed in KB/s
30000
20000
10000
0
5
10
15
20
25
Runtime in seconds


Accumulated GC pauses of 20 iterations batik benchmark with 80MB heap size
Accumulated GC pauses in seconds
0
1
2
3
4
5
6
default
jatt gc
jatt temp
ils gc
ils temp


Total runtime of 10 iterations h2 benchmark with 300MB heap size
Accumulated runtime in milliseconds
0
20000
60000
default
jatt gc
jatt temp
ils gc
ils temp


Last iteration runtime of 20 iterations xalan benchmark with 10MB heap size
Runtime in milliseconds
0
500
1500
2500
default
jatt compiler
jatt gc
ils gc


CPU usage of xalan benchmark with 10MB heap size
default
jatt gc
ils gc
tools
CPU usage in percent
400
300
200
100
0
10
20
30
40
50
60
Runtime in seconds


IOPS of xalan benchmark with 10MB heap size
default
jatt gc
ils gc
tools
Write speed in KB/s
12500
10000
7500
5000
2500
0
10
20
30
40
50
60
Runtime in seconds

Accumulated GC pauses of 20 iterations xalan benchmark with 10MB heap size
Accumulated GC pauses in seconds
0
5
10
15
20
25
30
default
jatt gc
ils gc