Mart Mägi

Visualising the logs of Shenandoah Garbage Collection algorithm

Bachelor's Thesis (9 ECTS)

Supervisor: Vladimir Šor

Supervisor: Pelle Jakovits

Tartu 2017

Visualising the logs of Shenandoah Garbage Collection algorithm

Abstract:

The aim of current thesis is to implement a GC log parser for Red Hat's Garbage Collection algorithm Shenandoah by extending an open-source project GCViewer. Additional aim is to take a further look into the Garbage Collection in Java.

The thesis is split into two main parts. The first part describes the background of Garbage Collection in Java and upcoming changes to the logging system in Java 9. The second part covers the implementation of the parser itself.

Keywords: Java, JVM, Garbage Collection, Shenandoah, visualization, logs, GCViewer

CERCS: P175 Informatics, systems theory

Shenandoah mälukoristuse algoritmi logide visualiseerimine

Lühikokkuvõte:

Käesoleva lõputöö eesmärgiks on Red Hati uue prügikoristusalgoritmi Shenandoah logide parseri implementeerimine avatud lähtekoodiga projekti GCViewer raames. Lisaeesmärgiks on anda ülevaade Java mälukoristuse abstraktsioonist.

Lõputöö koosneb kahest osast - esimeses osas uuritakse süvendatult prügikoristuse abstraktsiooni Javas ning selle erinevaid implementatsioone, logisüsteemi plaanitavaid muudatusi Java 9 ning teises osas kirjeldatakse parseri implementeerimist.

Võtmesõnad: Java, JVM, prügikoristus, Shenandoah, logid, visualiseerimine, GCViewer

CERCS: P175 Informaatika, süsteemiteooria

Table of Contents

1. Introduction

One of the many core values of the Java Virtual Machine (JVM) is its automated memory management. JVM applies an abstraction called Garbage Collection (GC) to the application which marks all objects that are reachable from the main application thread and the rest as garbage. The garbage collector then attempts to reclaim memory occupied by the objects marked as garbage and may compact the memory depending on the GC algorithm chosen. This solves many problems that arise from manual memory management such as basic memory leaks (forgetting to free objects) or dangling pointers (clearing memory while references remain) [1]. While this is a major boost to developer productivity, in some cases the GC starts to impact the application's performance. In those cases, the engineer needs to know the internal workings of the Garbage Collection.

This brings us to the three main performance aspects of the Garbage Collection:

- throughput - how many operations the system can complete in a given time unit;
- latency - the average response time of the application;
- capacity - this includes limits for memory, computing resources, monetary cost.

Different GC algorithms have been implemented to achieve better results in certain aspects. For example, Concurrent Mark and Sweep GC provides great latency with the drawback of capacity problems (fragmentation). Parallel GC provides best throughput with the drawback of latency problems in case of big heaps due to non-concurrent collections [2].

A new GC algorithm has been developed by developers at Red Hat to combat latency problems of applications with large heaps - Shenandoah. It was first added to OpenJDK in October 2015. Since then, it has been constantly improved to reach their long term goal of creating a pauseless GC [17]. It is aimed at large multi-core machines with large heaps. Shenandoah's goal is to manage heaps of 100GB and larger with pauses of 10ms or shorter. [3]

Being able to monitor those performance aspects of an application allows the developer to measure how the GC affects the performance of the application and make adjustments

accordingly. This is achieved by analyzing the GC logs generated by the JVM (mostly in production environments). However, the GC logs can quickly surpass the point of human-readability. Therefore for performance analysis a developer needs some form of concise visual representation of it.

1.1 Problem Statement

At the time of choosing the thesis topic, Shenandoah had no tools available to visualize GC logs. This means that developers who want to run their application with Shenandoah GC don't have a human-readable method of analyzing the performance of the GC. The aim of this thesis is to offer a visual representation of Shenandoah GC logs by extending GCViewer, an open-source GC visualization tool that currently supports all major GC algorithms up until Shenandoah.

1.2 Contributions of the thesis

The main contributions of the thesis will be, in order of importance:

- Implementing a visual log parser for Shenandoah GC logs.
- Contributing to an open source GC log visualization program GCViewer.
- Analysing the key points of Shenandoah logs that require visualisation for developers' performance analysis needs.
- Providing insight on the changes to JVM logging in Java 9.
- Providing an overview of the Garbage Collection in Java.

1.3 Outline

The thesis contains 5 main chapters - introduction, background, Shenandoah log parser, validation and conclusion.

The first chapter contains the introduction to the Garbage Collection in Java and describes the motivation for the thesis. Main contributions of the thesis are brought out.

The second chapter describes the background of Garbage Collection in Java and the upcoming changes to JVM logging in Java 9. A brief overview of the Shenandoah algorithm and GCViewer will be given.

The third chapter discusses the implementation of the Shenandoah log parser, displaying GCViewer's and solution's architecture, design choices and expected output. Comparison to other related works will be provided.

The fourth chapter is dedicated to validation of the parser. The parser will be validated by using the GC performance aspects and unit tests. Screenshots of the created application are provided.

The fifth chapter summarizes the thesis and discusses possible future work on the topic.

2. Background

2.1 Garbage Collection in Java

Garbage Collection (GC), also known as automatic memory management, is the automatic recycling of dynamically allocated memory. Garbage Collection is done by a garbage collector that performs following three main steps (mark, sweep, compact):

1. the collector marks reachable objects,
2. removes objects that can't be reached,
3. compacts the heap [1].

The third step is optional and only performed by some GCs. The garbage collector may perform GC without stopping the application threads (concurrent collector), while the application is completely stopped (stop-the-world collector) or perform GC as a series of small discrete operations with (potentially long) gaps in between (incremental collector). It can also use multiple CPUs to perform GC (parallel collector) [2]. A combination of these is mostly used, for example Shenandoah has a parallel and concurrent collector with some short stop-the-world (STW) pauses [3].

In Java, GC is done by a daemon thread that assumes that most objects quickly become unreachable (die young) and old objects rarely reference new objects. Based on this assumption, JVM splits the heap (application's memory) into two main categories - young and old (tenured) generations. Young generation is split into Eden and Survivor regions, where Eden is for the initial allocation of objects and Survivor for those that have survived a Garbage Collection [1]. This however is no longer the case with Shenandoah, where there is one continuous heap that has no separate young generation [4].

When a new object is created, JVM tries to allocate memory for it in the young generation. If it is unable to do so, the garbage collector causes a Minor GC event. This consists of a stop-the-world pause, during which the young generation gets cleared - live objects are copied

from Eden to a Survivor region and unreachable objects in young generation are removed. Some GCs also compact the Survivor regions but never the Eden region [2].

If an object hasn't been reclaimed for a fixed amount of minor GCs from the Survivor region, it will be promoted to the old generation during the next minor GC. Given enough of these, eventually the old generation will get full as well and a major GC will trigger, clearing the old generation. In cases when the entire heap fills up, GC threads will clean the entire heap during a Full GC, collecting the young generation first [1]. From Java 8 and up these are called Mixed collections, and in Shenandoah all collections are "Full" GCs as there is no separate young generation [4].

Java 8's default GC algorithm is the Parallel collector. Other existing GC algorithms, which can be activated with JVM options (flags) included in brackets, include:

- Serial (-XX:+UseSerialGC) is a fully single-threaded GC. It uses mark-copy for young and mark-sweep-compact for old generation, pauses the application threads for the whole duration of GC while providing small footprint and minimal overhead [2].
- Parallel Old (-XX:+UseParallelOldGC) is a fully multi-threaded GC. It uses mark-copy for young and mark-sweep-compact for old generation, provides higher throughput and shorter collection times than Serial GC. It performs all GC work while application threads are stopped [2].
- Concurrent Mark and Sweep (-XX:+UseConcMarkSweepGC) is a fully multithreaded GC. It only pauses the application for initial and final marking phases, while the GC thread runs concurrently with the application threads during the sweeping phases of young and old generations and compacting of the young generation. It provides minimal pause times due to compacting only the young generation. As CMS does not compact the old generation, using CMS leads to fragmentation and poor performance for applications that run for an extended period of time [1].
- Garbage First (-XX:+UseG1GC) GC introducing incremental garbage collection - it divides the heap into (mostly 2048) smaller regions. Garbage First (G1) only collects subsets of the regions with most garbage first. This allows G1 to provide more predictable pause

times to applications even with large heaps as the entire heap isn't collected at once. G1 stops the application threads for young collections, initial and finishing marks of old collection and does the rest of the GC concurrently except for compacting the old generation, which requires a STW pause for the full duration of the evacuation. G1 will also be the default GC in Java 9 [5].

● Shenandoah (`-XX:+UseShenandoahGC`) GC uses a heap that is split into regions like G1, however a region may contain newly allocated objects, long lived objects, or a mix of both. Shenandoah doesn't have a separate young collection and introduces concurrent compacting of the heap. It stops the world briefly for the initial mark of the heap, performs concurrent marking, stops the world again for final mark and proceeds with concurrent evacuation. This results in very high response times with slightly lower throughput (10%) than other HotSpot GCs [4].
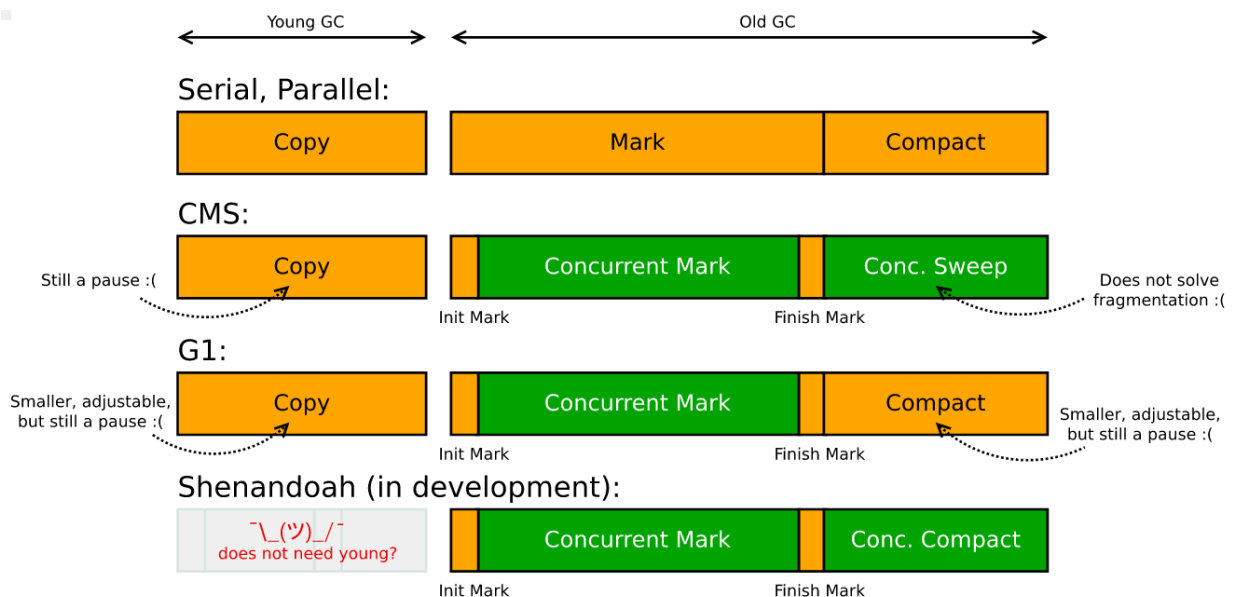

Figure 1. Yellow are stop-the-world phases, green are concurrent phases [6].

2.2 Unified Logging in Java 9

Java 9 introduces a completely revamped format of Garbage Collection logs. The main goal of the related JDK Enhancement Proposals (JEPs), Unified JVM Logging (JEP 158) and Unified

GC Logging (JEP 271), is to make the log format as consistent as possible by typically keeping it down to one line per one GC message [7,8] .

This unfortunately means that the existing log parsers will not work in current state with logs created by applications running Java 9. Therefore, it was not possible to reuse any log parsing code from GCViewer when creating the Shenandoah parser. From JEP 271: "It is not a goal to ensure that current GC log parsers work without change on the new GC logs." [8]. Some GC related JVM flags will also be marked as deprecated, e.g. PrintGC or PrintGCTimeStamps as they will be incorporated into the default JVM logging flag `-Xlog` [9]. However the JVM will automatically use aliased ones and inform the user, e.g.

`-XX:+TraceClassLoading is deprecated. Will use -Xlog:class+load=info instead.`

The new format of GC messages can be "decorated" in either the program arguments or at runtime with the default decorations being `uptime, level, tags`. An example of a GC message with default decorations:

`[6.567s][info][gc,old] Old collection complete.`

From `java -Xlog:help` we can see that there will be 6 available log levels - `off, trace, debug, info, warning, error`, 98 total log tags with most relevant for GC - `os, gc, ergo, phases, heap` and 12 log decorators such as `level, tags, pid` [7].

Prior to Unified Logging, at least 5 different JVM flags were required to get the needed output. This was considered a bad design choice, as the user had to know the flags existed in the first place and then write them one by one. Unified Logging removes this complexity, consolidating the customization to a single JVM flag. For example, using `-Xlog:gc=debug:file=gc.txt:none` logs only messages tagged with 'gc' tag using 'debug' level to file 'gc.txt' with no decorations. Unless wildcard (*) is specified, only log messages tagged with exactly the tags specified will be matched. For example, using only *Xlog:gc* without the wildcard will only log messages that contain just the [gc] log tags, ignoring tags such as [gc, ergo] etc [7].

2.3 Shenandoah garbage collector

Shenandoah is an open-source region-based low-pause parallel and concurrent garbage collector. Shenandoah is very similar to G1 with similar heap structure and incremental collections. However, there are two major differences compared to previous GC algorithms - Shenandoah introduces complete removal of generations from the heap and concurrent compaction [4].
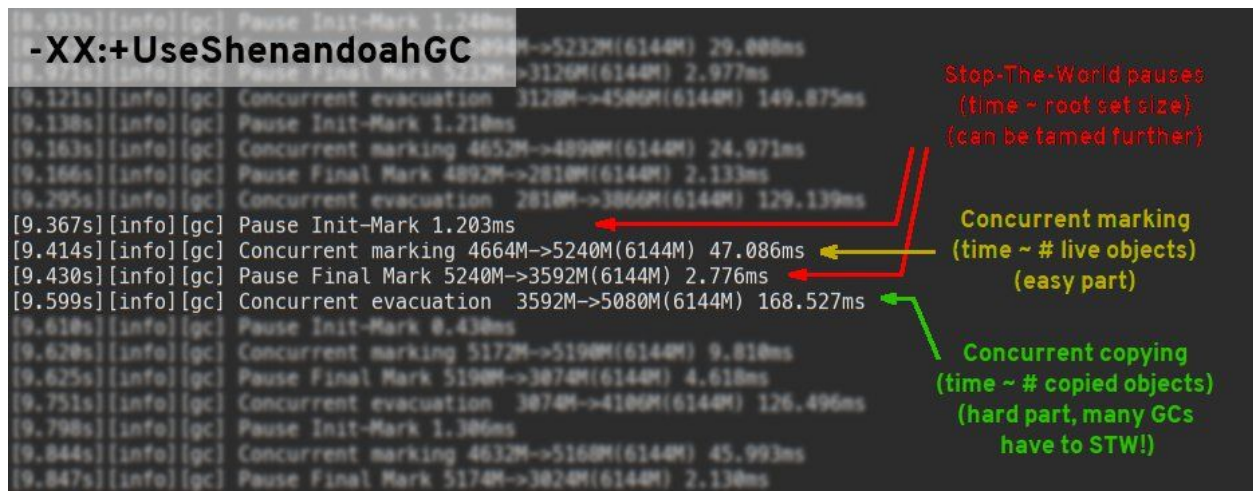

Figure 2. Shenandoah in one picture (Appendix 2)

Garbage collection in Shenandoah contains 2 main phases - a marking phase followed by an evacuation phase. Both phases mostly run concurrently with application threads, using parallel GC threads. Shenandoah collector claims regions with the most garbage (least live objects) regardless of their age, unlike G1 which prioritizes young generation regions. [10]

The concurrent marking phase starts with a short stop-the-world (STW) pause for initial marking, followed by concurrent marking of all objects and another short STW pause for final marking. To achieve synchronization with application threads during the concurrent mark phase, Snapshot at the Beginning (SATB) algorithm is used. SATB utilizes a write barrier, a hook that triggers when an application thread tries to change an object or its reference during concurrent marking. After the concurrent mark phase, any changes from hooks are presented to GC. This ensures that no changes in heap get lost. [11]

The concurrent evacuation phase is made possible by the use of Brooks forwarding pointers. Every object in the heap has an extra reference field. When an object is initialized, that field points to itself. After moving an object, the field is updated to point to that new location. This ensures that during the concurrent evacuation phase no threads attempt to reach an object that had been moved. Instead, all reads requested by application threads go through the Brooks pointer to find the exact current location of that object. [12]

Having the extra fields on every object for synchronization protection results in overhead and therefore Shenandoah will never become a "go-to" GC for every JVM application. Shenandoah is designed for very large heap of 20GB and higher by making pause times independent of heap size resulting in low latency.

2.4 GCViewer

GCViewer is an open source tool for parsing and analyzing GC log files. It's a fast and efficient way to visualize Garbage Collection logs and get instant feedback about the JVM's health [13]. It easily identifies following pain points in the application's performance:

- Low throughput. GCViewer calculates throughput on the following formula: $\frac{application\ running\ time\ -\ total\ pause}{application\ running\ time} * 100$. e.g. if an application logs contain 5 minutes of info and 3 minutes of that was spent on GC, the throughput would be 40%. It depends on the application, but generally throughput below 90% means that the application may require GC optimization.

- Excessively long individual GC pauses. If a fraction of the users complain about long response times, it may be worth consulting GCViewer for spikes in GC pauses. This is especially important for latency-critical applications where the minimum response time may not exceed for example 1000ms.

- High heap consumption. When the application suddenly stops responding or becomes sluggish, the problem may lie in wrong heap size. When the heap is too small for the application, even after a garbage collection the heap will remain close to being fully

utilized, resulting in another GC pause soon enough. Therefore, the application requires optimization and GCViewer helps detect it [1].

GCViewer currently supports parsing of the most prevalent GC algorithms other than Shenandoah. The project doesn't support Unified JVM Logging format yet and it's not under development at the moment. [14] Therefore, in order to parse Shenandoah logs, support for Unified Logging format had to be added first.

3. Shenandoah log parser

The aim of the Shenandoah parser was to support logs created with the -Xlog default configuration, i.e. the equivalent of:

`-Xlog:all=warning:stderr:uptime,level,tags.`

At the time of writing the thesis, Linux operating system was mandatory to run Shenandoah. Shenandoah was only included in OpenJDK builds which cannot be installed on Windows or Mac machines. For this purpose, "Bash on Ubuntu on Windows" was used to run Java applications with Shenandoah. A Shenandoah developer's guide for building OpenJDK 9 was followed [15].

The command used for running the Java applications with Shenandoah was:

```
bash -c
"/mnt/c/test/shenandoah-jdk9/build/linux-x86_64-normal-server-release/images/j
dk/bin/java  -Xlog::log_file_name -XX:+UseShenandoahGC -Xmx8g java_app_name"
```

The code written as part of this thesis is in a GitHub code repository shown in Appendix 1.

3.1 Log analysis

The solution currently parses the 6 main GC messages of Shenandoah - initial mark, concurrent mark, final mark, concurrent evacuation, concurrent reset bitmaps and allocation failure. Those messages are only a small fraction of the log file, but since *"most of the information generated is ignored by GCViewer"*, same applies to Shenandoah parser as well. [13]

Example logs from a Java application running Shenandoah (skipping lines for brevity):

```
[0.730s][info][gc,start     ] GC(0) Pause Init Mark
[0.731s][info][gc           ] GC(0) Pause Init Mark 1.021ms
[0.731s][info][gc,start     ] GC(0) Concurrent marking
[0.735s][info][gc           ] GC(0) Concurrent marking 74M->74M(128M) 3.688ms
[0.735s][info][gc,start     ] GC(0) Pause Final Mark
[0.736s][info][gc           ] GC(0) Pause Final Mark 74M->76M(128M) 0.811ms
[0.736s][info][gc,start     ] GC(0) Concurrent evacuation
```

```
[0.736s][info][gc             ] GC(0) Concurrent evacuation  76M->84M(128M)
0.452ms
[0.736s][info][gc,start     ] GC(0) Concurrent reset bitmaps
[0.736s][info][gc             ] GC(0) Concurrent reset bitmaps 0.051ms
...
[29.628s][info][gc             ] Cancelling concurrent GC: Allocation Failure
[29.657s][info][gc,start     ] GC(831) Pause Full (Allocation Failure)
[29.658s][info][gc,phases,start] GC(831) Phase 1: Mark live objects
[29.675s][info][gc,phases     ] GC(831) Phase 1: Mark live objects 17.256ms
[29.675s][info][gc,phases,start] GC(831) Phase 2: Compute new object addresses
[29.699s][info][gc,phases     ] GC(831) Phase 2: Compute new object addresses
23.568ms
[29.699s][info][gc,phases,start] GC(831) Phase 2: Adjust pointers
[29.725s][info][gc,phases     ] GC(831) Phase 2: Adjust pointers 26.061ms
[29.725s][info][gc,phases,start] GC(831) Phase 4: Move objects
[43.914s][info][gc,phases     ] GC(831) Phase 4: Move objects 14078.141ms
[43.948s][info][gc             ] GC(831) Pause Full (Allocation Failure)
7943M->6013M(8192M) 14289.335ms
```

As confirmed by a developer working on Shenandoah, *"Anything starting with "Pause…" is STW"* and *"Phase "X" are the phases of Full GC, and Full GC is STW".* [16] Therefore, initial mark, final mark and allocation failure GC events are STW pauses. Allocation Failure GC event consists of 4 different GC phases, all of which are STW pauses that are summarized in a single line at the end. Concurrent marking, evacuation and reset are concurrent GC events that run concurrently with the application threads.

Due to the concurrent nature of evacuation phase, memory may still be allocated by the application threads. This may result in a heap state that has a higher occupancy of the heap after the evacuation, e.g.

```
[686.993s][info][gc ] GC(1050) Concurrent evacuation 7612M->7832M(8192M)
43.968ms
```

3.2 Implementation details

The pre-existing architecture of GCViewer, on top of which the parser was built, is as follows:
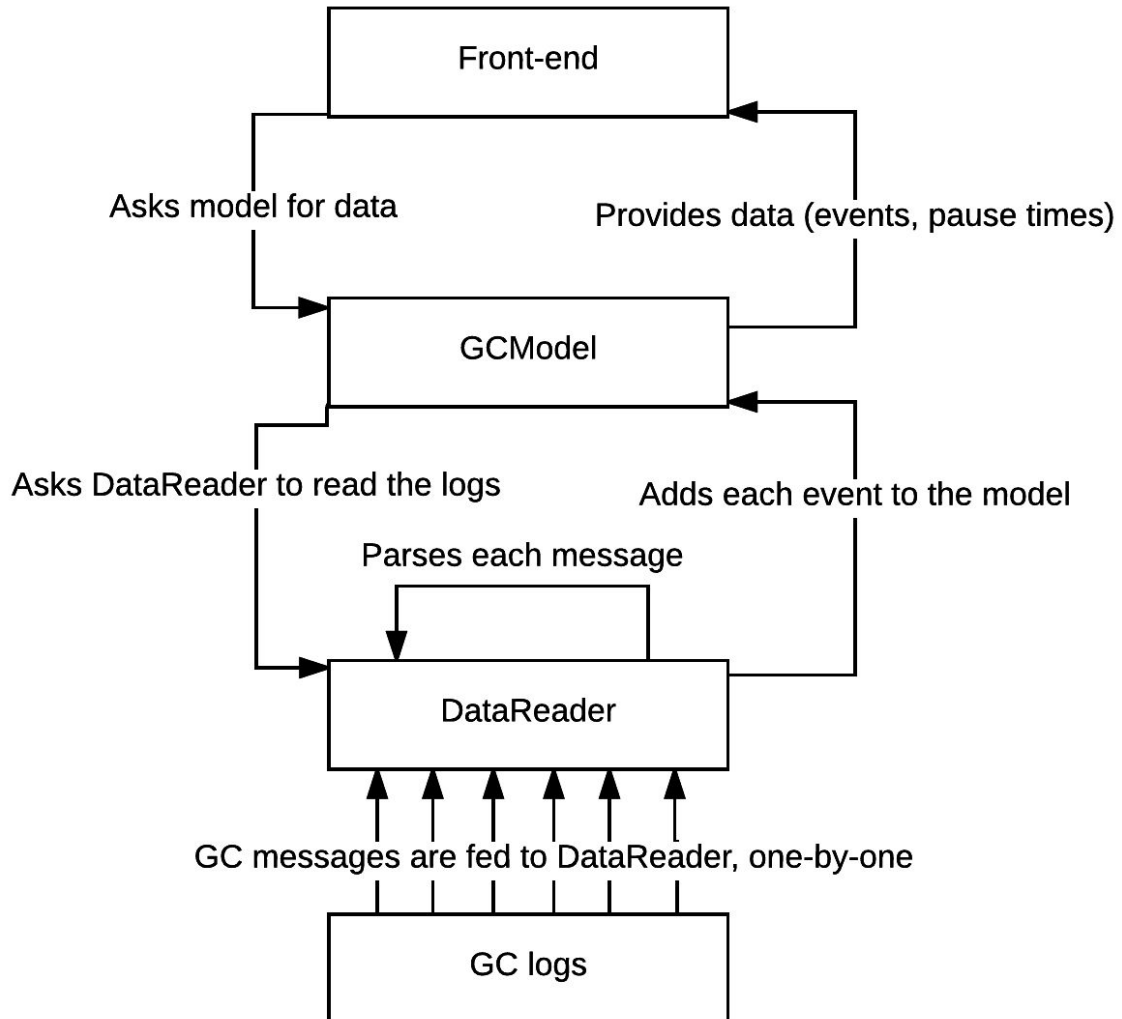

Figure 3. Visual demonstration of GCViewer's architecture (self-created).

The application's front-end consumes data from `com.tagtraum.perf.gcviewer.model.GCModel`.

`GCModel` is populated with data from a `com.tagtraum.perf.gcviewer.imp.DataReader` which is chosen based on the log format. It then cycles through all the `com.tagtraum.perf.gcviewer.model.GCEvent` objects and provides data to the application's front-end based on the newly acquired data.

`DataReader` fills a `GCModel` with `GCEvent` objects, which carry information such as pause times, timestamp, GC duration, concurrency, heap changes (if present).

The main additions to the project were `com.tagtraum.perf.gcviewer.imp.DataReaderShenandoah` and `com.tagtraum.perf.gcviewer.model.ShenandoahGCEvent` classes. The `GCModel` class was also changed to handle `ShenandoahGCEvent` objects differently from `GCEvent` objects. `DataReaderShenandoah` class was implemented to handle the main parsing of Shenandoah GC logs. Its takes a stream containing bytes of the log file as input and outputs a `GCModel`. `ShenandoahGCEvent` extends the previously existing `GCEvent` class. It was created to hold extra information about GC events and to simplify unit conversion.

The parser uses regular expressions to extract the useful information about GC messages. Using the default configuration of -Xlog adds a lot of log messages that contain information that is not relevant to GCViewer. Therefore most of the log entries are not matched by the regular expressions and thus ignored by the parser. All lines of the GC log are one by one either excluded or parsed. When the parser matches one of the 6 Shenandoah GC events, it creates a `ShenandoahGCEvent` object and adds the relevant info to it. The object is then added to the `GCModel` and the program repeats the process with the next line.

The parser uses 2 different regular expression patterns, one for matching events with heap information and another for those without. For example, the parsing process gets log messages as input (lines skipped for brevity):

```
[575.388s][info][gc,start        ] GC(2244) Pause Init Mark
[575.389s][info][gc              ] GC(2244) Pause Init Mark 0.705ms
...
[596.755s][info][gc,phases       ] GC(2247) Phase 4: Move objects
20530.821ms
```

```
[596.758s][info][gc         ]  GC(2247)  Pause  Full  (Allocation  Failure)
7948M->7004M(8192M) 20723.759ms
```

The first line is skipped, because it doesn't match either regular expression. The second line is matched by the pattern, and the matcher gathers groups from it - group 1 contains the timestamp (575.389) and group 2 holds the duration of the pause (0.705). A `ShenandoahGCEvent` object is created, and it's values are set - timestamp, pause time of the event and its `com.tagtraum.perf.gcviewer.model.Type` as Initial Mark. Type contains more info, the description of the event, the generation it covers (for Shenandoah, all are marked as Tenured) and concurrency (in this case, STW).

The third line matches the pattern, but it's excluded by program logic because it's more efficient to parse just one message that contains all the allocation failure information - the next line. The pattern matches the fourth line, and gathers 3 different groups from it. Group 1 carries the timestamp (596.758), group 2 contains changes to the heap (7948M->7004M(8192M)) and group 3 contains the pause duration (20723.759). A `ShenandoahGCEvent object` is created and information is added to it - the timestamp, heap size before the collection, heap size after the collection, total heap size at the time of the event, the duration of the GC event and the type as Allocation Failure. The type contains description, generation and concurrency (STW).

Some statistics that were displayed in previous parser implementations are not displayed in Shenandoah. This is due to changes with heap and how GC events affect it, such as the removal of generations and concurrent evacuation. Therefore, some statistics, such as memory usage in tenured, young, perm heap spaces or GC Performance, are not displayed in the Shenandoah parser. Concurrent evacuation stats are also not displayed.
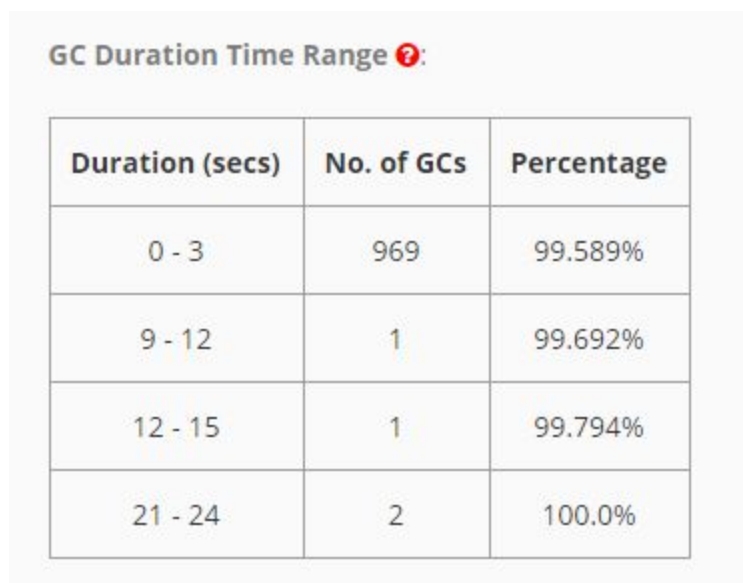
A pull request (request for code to be integrated with a project) was made to integrate the Shenandoah parser to GCViewer project (Appendix 7).

3.3 Related work

Several tools exist to monitor Java applications heap or to analyze the GC logs such as VisualVM, GCEasy or GCViewer. At the time of choosing the thesis topic, none of those supported Shenandoah.

During the writing of the thesis, GCEasy (Universal GC Log Analyzer) started to provide basic support for Shenandoah logs. An example of GCEasy's output is shown in Appendix 2.

Using logs generated with the code with a memory leak in Appendix 4, GCEasy's log parser however made a lot of mistakes, completely nullifying its reliability. For example, it claims to have only 1 GC event with pause time between 9-12 seconds.

GC Duration Time Range ❓:

Duration (secs)	No. of GCs	Percentage
0 - 3	969	99.589%
9 - 12	1	99.692%
12 - 15	1	99.794%
21 - 24	2	100.0%

Figure 4. GC Durations from GCEasy's output (Appendix 3)

However, manually looking at the logs reveals that there's clearly multiple:

```
[668.523s][info][gc] GC(1042) Pause Full (Allocation Failure)
7923M->7004M(8192M) 10776.917ms
...
[680.753s][info][gc] GC(1044) Pause Full (Allocation Failure)
7912M->7004M(8192M) 11824.150ms
```

GCEasy also claims to have throughput of over 90%, however it can't be the case when application's thread were stopped for the most part of the application's run time.

1 Throughput **?** : 90.442%

Figure 5. Throughput statistic from GCEasy's output (Appendix 3)

4. Validation

4.1 Performance aspects

The parser was validated against the three performance aspects of Garbage Collection - throughput, latency, capacity. The main goal was to verify that it's possible to clearly see the pain points in the application's performance (if there are any) and therefore take action to improve it. The example output can be seen from Figure 6.
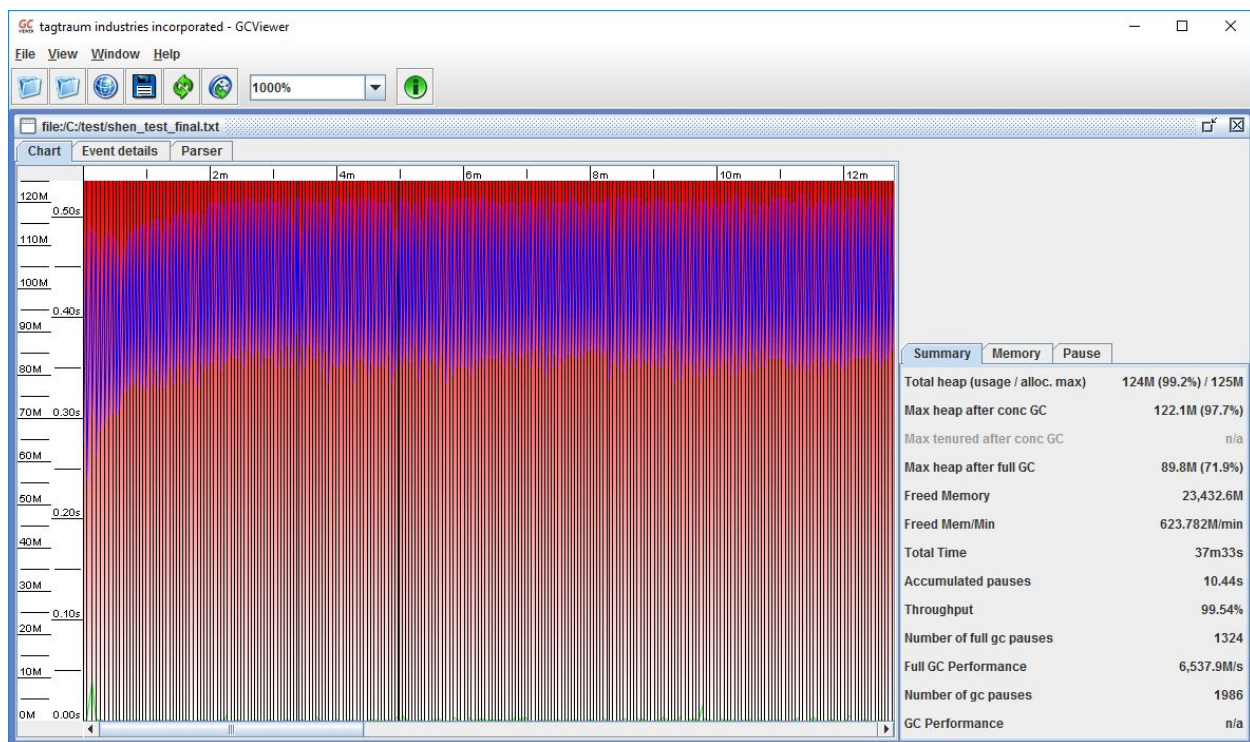

Figure 6. Parser output generated with a well-performing Java application (Appendix 4)

1. Throughput

   The applications throughput is displayed by the percentage of how long the application threads were working. For a Java program with a heavy memory leak (Appendix 6) used for testing, it is a mere 1.44% (Figure 7) which shows that a very large portion of the application's runtime is spent on STW GC pauses.

21

Total Time	11m48s
Accumulated pauses	697.84s
Throughput	1.44%
Number of full gc pauses	2039

Figure 7. Shenandoah parser's output of logs generated with code in Appendix 5

2. Latency

Latency as a GC performance aspect is very largely dependant on the worst-case pause time. A throughput of 1.44% in Figure 7 means terrible latency and slow response times as the application cannot respond to the user during the STW pauses. A high throughput does not necessarily mean great overall latency, as a single high pause doesn't heavily affect throughput but may cause timeouts for single users. Latency spikes can be detected with the parser as well - the green lines in Figures 6 and 8 show the GC times. In Figure 6, a program without a memory leak, the green lines are barely visible, whereas in Figure 8 the pauses spike as high as 22 seconds. If that was a single spike, that would mean only a small group of users would be affected and that problem would be nearly impossible to detect without analysing the logs with a visual parser.

3. Capacity

Figure 8 shows that in case of a severe memory leak, the heap quickly gets full (blue lines) and the application never recovers from it. The application gets stuck in a loop where it fails to clear enough of the heap during the GC event, the heap quickly becomes full again, triggering another GC event.

The statistics, "Max heap after conc/full GC" show exactly how much of the heap was still in use after a collection.
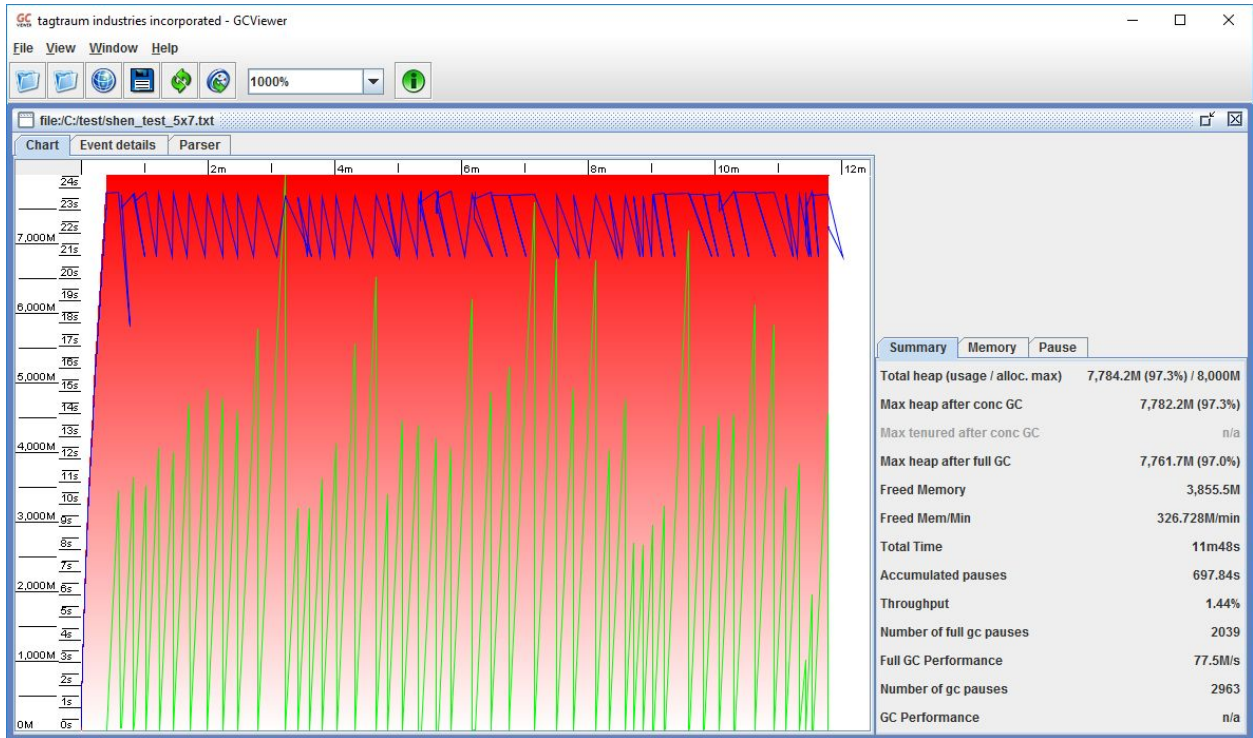
Figure 8. Parser output generated by a Java application that causes a memory leak (Appendix 4)

The performance aspect validation concludes that the Shenandoah parser provides clear benefit to the developer by allowing them to analyze how Shenandoah GC affects the GC performance.

4.2 Unit tests

Two unit tests were written for the parser in the class `com.tagtraum.perf.gcviewer.imp.TestDataReaderShenandoah`. The parsing of the main GC pause consisting of 5 events and the parsing of an allocation failure event was validated. It was tested whether the correct tags are assigned, heap changes are parsed in correctly and that post-parse the model has correct information about the events. The unit tests along with their input can be found in Appendix 6.

The unit tests confirmed that the GC messages were being parsed correctly and the correct information was extracted from them. They also confirmed that the `GCModel` receives the correct

data from the `DataReader` and stores it correctly. The testing concluded that the parser is working correctly.

5. Conclusions and future work

The aim of this thesis was to offer a visual representation of Shenandoah Garbage Collection (GC) logs. That was achieved by implementing a functional Shenandoah log parser as a contribution to an open-source project GCViewer. It allows the developer to analyze Shenandoah GC's performance and make improvements to the application accordingly.

The parser was validated by seeing if it's possible to analyze an application's GC performance aspects by using the parser to detect problems. Unit tests were written that passed successfully.

A pull request (request for code to be integrated with a project) was made to integrate the Shenandoah parser to GCViewer.

During the writing of the thesis, the author learned a lot about the Java ecosystem, Garbage Collection, JVM logging and how to find information regarding the topics.

Since Shenandoah hasn't been released yet, there are changes planned to the logging system and log structure. Therefore a need for additional changes to the core of the parser will arise later on. As the program currently supports just the default configurations of `-Xlog`, additional configurations would have to be parsed such as decorators, currently the most important addition would be to parse the `datetimestamps` decorator.

6. Bibliography

[1] Salnikov-Tarnovski, N., Smirnov, G., "Java Garbage Collection handbook", 2015,

https://plumbr.eu/java-garbage-collection-handbook (03.05.2017)

[2] Java Garbage Collection Basics,

http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html (11.05.2017)

[3] JEP 189: Shenandoah: An Ultra-Low-Pause-Time Garbage Collector, 2014,

http://openjdk.java.net/jeps/189 (03.05.2017)

[4] Flood, Christine H., et al. "Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK." Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools. ACM, 2016,

https://www.researchgate.net/publication/306112816_Shenandoah_An_open-source_concurrent_compacting_garbage_collector_for_OpenJDK (03.05.2017)

[5] Standard Edition Oracle JDK 9 Migration Guide, 2017,

https://docs.oracle.com/javase/9/migrate/toc.htm#JSMIG-GUID-1F270BDA-50B0-49C8-807E-0B727CCC5169 (03.05.2017)

[6] A. Shipilev. "GC Design and Pauses", 2017,

https://shipilev.net/jvm-anatomy-park/3-gc-design-and-pauses/ (03.05.2017)

[7] JEP 158: Unified JVM logging

http://openjdk.java.net/jeps/158 (03.05.2017)

[8] JEP 271: Unified GC logging, 2014,

http://openjdk.java.net/jeps/271 (03.05.2017)

[9] Unified GC Logging presentation by Marcus Larsson, 2016,

https://static.rainfocus.com/oracle/oow16/sess/14634110693470018RiI/ppt/unified-logging-j1.pdf (03.05.2017)

[10] Shenandoah's page on IcedTea

http://icedtea.classpath.org/wiki/Shenandoah (03.05.2017)

[11] Novotný, Matej. "Garbage Collector Shenandoah: server applications", 2015,
http://is.muni.cz/th/374505/fi_m/GarbageCollection.pdf (03.05.2017)

[12] Richter, Jan. "Garbage Collector Shenandoah: desktop applications.", 2015,
http://is.muni.cz/th/373939/fi_m/dp.pdf (03.05.2017)

[13] GCViewer's README file
https://github.com/chewiebug/GCViewer (08.05.2017)

[14] GCViewer's GitHub issue regarding Unified JVM Logging,
https://github.com/chewiebug/GCViewer/issues/155

[15] 6.5 Building latest OpenJDK9 source on Ubuntu
http://arturmkrtchyan.com/building-openjdk-9-on-ubuntu (07.05.2017)

[16] A. Shipilev, response to a question in Shenandoah mailing list,
http://mail.openjdk.java.net/pipermail/shenandoah-dev/2017-May/002298.html (05.05.2017)

[17] R. Kennke, C. Flood, Shenandoah An Ultra-low Pause Time Garbage Collector for
OpenJDK, 2015,
https://rkennke.files.wordpress.com/2014/02/shenandoahtake4.pdf (03.05.2017)

7. Appendices

Appendix 1: Author's fork of GCViewer

https://github.com/ophillan/GCViewer (10.05.2017)

Appendix 2: Shenandoah in one picture by Aleksey Shipilev

https://twitter.com/shipilev/status/820010017294262273 (10.05.2017)

Appendix 3: GCEasy log display

http://kodu.ut.ee/~ophillan/BSc/GCEasy%20output.html (10.05.2017)

Appendix 4: Code with a memory leak used to generate Shenandoah logs

http://kodu.ut.ee/~ophillan/BSc/GCApplication_memoryleak.java (10.05.2017)

Appendix 5: Code without a memory leak used to generate Shenandoah logs

http://kodu.ut.ee/~ophillan/BSc/GCApplication_no_leak.java (10.05.2017)

Appendix 6: GitHub commit with added Unit tests

https://github.com/ophillan/GCViewer/commit/25e6d9f7246601c87991bd20052994471cc60933 (10.05.2017)

Appendix 7: GCViewer pull request

https://github.com/chewiebug/GCViewer/pull/187 (10.05.2017)

Appendix 8: Non-exclusive licence to reproduce thesis and make thesis public

I, Mart Mägi,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

1.1 reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

1.2 make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

"Visualising the logs of Shenandoah Garbage Collection algorithm",

supervised by Vladimir Šor and Pelle Jakovits,

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 11.05.2017