MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Garbage Collector Shenandoah: server applications

DIPLOMA THESIS

Matěj Novotný

Brno, 2015

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by 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.

Matěj Novotný

Acknowledgement

I would like to thanks my advisor RNDr. Adam Rambousek for his guidance and relaxed attitude. I would also like to thank RedHat for giving me the opportunity to write this thesis as well as for granting me the access to SPECjbb benchmark and high performance computer for testing. My sincere gratitude belongs to Shenandoah developers who willingly answered my questions. Last but not least, I would like to thanks my friend Lucy who had broadened my knowledge regarding formal English writing.

Abstract

This thesis sheds light on garbage collection mechanics in Java programming language and explains its basic concepts as well as currently used solutions. Furthermore a new low pause collector, Shenandoah, is introduced. Latter part of this work contains a garbage collector performance testing using SPECjbb. Amongst the tested collectors there is Shenandoah as well as other commonly used collectors. Test results are then interpreted from several points of view such as pause time, heap usage and SPECjbb throughput.

Keywords

Java, Garbage Collection, Shenandoah, Memory Management

Contents

1 Introduction

In Java world, one can be an experienced developer and still not know much about the underlying garbage collection, or about means with which one can affect memory management. That is what automatic memory management was designed for after all. But sometimes a need arises to optimize an application or to better understand things such as weak references and object finalization. In order to optimize Java application from a memory management perspective, one has to understand collectors. The information on collectors is sparse and oftentimes outdated, making it difficult to delve into this problematic. And even though Oracle provides some guidance with its tutorials, they are by no means thorough.

This thesis, in its first half, explains how one can indirectly manage memory in Java and what are the consequences of this for garbage collector. Reference types and finalization are briefly discussed. The focus of the work then shifts to garbage collectors and an in-depth look into basic collector concepts is presented. The following chapter aims to describe collectors which are present in recent Java Virtual Machine (JVM) and also outlines their typical use cases. Having covered currently used garbage collectors, the thesis moves to Shenandoah, a low pause collector designed for huge heaps; mechanics and goals are explained as well as developer's aim to integrate this technology into OpenJDK.

Second part of this thesis is about garbage collector performance testing in terms of server applications. For this purpose, SPECjbb was chosen as it is a well known benchmarking application simulating server-like application in Java. Not only does this test provides a reasonable load on a targeted computer, but it also measures application throughput. Since test conditions were equal and only collectors changed, SPECjbb results should reflect garbage collector suitability for this kind of application. In the thesis, after explaining the test suite, there is a section about hardware which was used and also another section regarding monitoring tools. Finaly, there are the test results themselves, which are first presented per collector and at the end there is a comparison of results from different angles including pause times, heap usage and SPECjbb throughput.

2 Memory Management

Every programming language has a need for some sort of memory management. In certain languages, like C for instance, one can explicitly allocate and free memory. While this approach allows programmers to gain control over memory, it also introduces a risk of memory leaks. Source code 2.1 shows two fuctions, both of which allocate memory, but the latter one fails to free it again.

```
1  void allocation(void) {
2     /* allocate memory */
3     name = malloc(10 * sizeof(char));
4
5     /* do some actual work */
6     /*and free the memory again */
7     free(name)
8  }
9
10 void allocation_with_leak(void) {
11    /* allocate memory */
12    surname = malloc(15 * sizeof(char));
13
14    /* do some actual work; the memory is not freed this time */
15 }
```

Source code 2.1: Memory Allocation and Memory Leak in C

But memory leak is not the only problem one can come across when manually operating memory. Other issues can be constant need for re-allocation due to size changes, accessing variable before allocating memory or using allocated structures beyond their lifetime. Since manual memory management can be challenging, other languages such as Java and Ruby have chosen to use garbage collector (GC). While this approach limits programmer's ability to work with memory, it enables it more or less automatic.

2.1 Memory Management in Java

In Java all objects are situated in an area called *the heap*. The heap is created when JVM starts and its size grows as the application executes. At some point in time or once the heap is getting full, garbage

is collected - objects which are no longer needed are removed hence leaving free space. The way collectors determine whether object can or cannot be removed is based on which algorithm is currently in place. There is no way to allocate memory manually in Java except for native code execution (e.g. running C code using native interface). However, there are other means to control memory such as finalization and references which are mentioned below.

2.1.1 Object Finalization

Finalization can be seen as an indirect method of memory management in Java. It gives every object an option to perform additional action once it should be garbage collected. Programmer cannot precisely determine when this is going to happen but is guaranteed that it will eventually do so. An example of finalization usage can be seen in Source code 2.2.

Despite the seeming usefulness of finalization, it is not recommended practice due to its impact on GC. Each object declaring this method has to be 'collected' twice. At first, once the object becomes garbage, the collector attempts to collect it. Upon inspecting each object the GC has to look for finalize method and if there is any, execute it. This obviously causes delays due to the execution time itself. Once this method is finished, the GC has to track all the changes it caused; new objects might have been created, or dead ones could have been brough back to life. Tracing these changes will only require minimum time. Nevertheless this is a serious overhead and it amplifies the disruptive nature of GC. It is also possible that objects declaring finalize method will be collected in the next cycle so as not to prolong memory clearing pauses. This will make the pauses shorter but will not free as much space in one invocation. Every kind of garbage collector deals with finalization in slightly different way; there is no general approach.

```
1 public class SampleWithFinalization{
2    protected void finalize () throws throwable {
3       // following method will be executed once GC attempts to
            collect this object
4       cleanAfterYourself();
```

```
5       }
6   }
```

Source code 2.2: Finalization in Java

2.1.2 Weaker Object References

Apart from finalization, one can also use different object references as yet another indirect memory managment method. They are not nearly as common as finalization and their use can be very specific but still, they are worth mentioning. There are four kinds: strong, weak, soft and phantom references.

- Strong references are common object references. GC wil not collect these unless there is at least one reference to them.
- Weak references will be collected in the next GC invocation if there is no strong reference to them. This can be viewed as an eager approach since GC attemps to collect objects with weak references as soon as possible. The use for this kind of reference is a cache, or meta data storage where one does not mind having objects garbage collected often. Also when retrieving such objects, there has to be a null check to make sure it still exists.
- Soft references are very similar to weak references but GC does not use eager approach meaning that objects will not get collected during the very next invocation. Quite the opposite, they will hang on the heap until JVM starts running out of memory in which case it will collect all soft references to create some free space. Therefore soft references are the best candidates for cache implementation. Also, null check is required as one cannot be sure whether the object was garbage collected or not.
- Phantom references are the most tenuous and the most difficult to implement; also they are rarely ever used. One can never obtain object via this reference (the result is always null) and the GC can collect objects with only phantom references at any convenient time. Furthermore while creating phatom references

one has to deal with ReferenceQueue as well. This queue will contain phantom referenced object right after it was garbage collected (it is an exception to standard behavior of ReferenceQueue). This allows programmer to determine exactly when this objects gets garbage collected and react appropriately. So while this approach might be slightly more confusing it can fully replace finalization while not presenting any serious overhead for garbage collectors.

3 Garbage Collection

Every GC has to do two things. Firstly, it has to detect "garbage" and secondly, the heap space occupied by such objects has to be freed. Identifying garbage can be simplified to traversing rooted graph because when an object is reachable from root it is considered "live" and will not be removed. Remaining unreachable objects can be garbage collected, because they can no longer affect a running program.

3.1 Various Types of Garbage Collectors

There are several kinds of GCs varying in the way how they mark objects as garbage, how they move objects on heap as they free space and how they separate heap space in general. Not all of them are used in current JVM implementations but they are worth mentioning as they became a basis for what Java uses now. There are seven types of GC mentioned below - Reference Counting, Tracing, Compacting, Copying, Generational, Increment and Adaptive Collector. Every author of a book which focuses on GC may state slightly different number of collectors. This is because some collector types are typically used together, for instance generational and incremental. The list and description of collector types used in this thesis follows Bill Venners' book [1].

3.1.1 Reference Counting Collector

This strategy was amongst the earliest in garbage collecting. For each object it counts the number of references beginning with object creation and assignment to a variable. For each other reference a counter is increased by one and, unsurprisingly, whenever a reference is lost (object is assigned a new value, reference goes out of scope) the counter decreases. As long as the counter is bigger or equal to one, the object is still considered "live". Otherwise it can be garbage collected; once this happens any other object that was reffered by this one has its counter decreased. This can lead to a chain reaction, as multiple objects can be subjected to garbage collection as their counters are decreased to zero.

While this approach may seem outdated, it still has one great advantage - the garbage collection executes quickly and does not pause the application for extended period of time. It runs in small time chunks as only few objects are collected each time and owing to counters the GC knows precisely which objects are to be collected. This behaviour is desired for several use cases such as real time systems. However there is a significant drawback which led to abandonment of this approach; it does not detect *cycles*. If there are two objects reffering to each other, reference counting collector will fail to recognize this and will mark such objects as garbage.

3.1.2 Tracing Collector

Tracing collector makes use of the root nodes and walks down the graph through all the nodes. The root nodes differ based on GC and JVM implementations but one can imagine them as start points from which all referred objects can be reached. In this graph the node is an object and every edge is a reference. Each node the collector reaches is marked in some way, meaning it is reachable and therefore *alive*. Marking can be done separately for each node or the GC can make use of an external structure such as bitmap. Objects which have not been reached, and therefore marked, are considered garbage as there are no references to them. Tracing collectors have two phases; first phase consists of traversing the graph and marking objects; second phase is garbage collecting of all objects which the collector could not reach.

This approach became a basis for current collector strategies. Its simpliest version is known as *mark and sweep* algorithm. As the name suggests it has two parts: marking (traversing the graph to learn which objects are reachable) and sweeping (collecting unreachable objects and actually freeing some heap space). Further information about this approach can be found in the Current GC Implementations chapter.

It is worth mentioning that the 'marking' part of this algorithm may be very long for huge applications (the graph will be enormous) and while the collector executes, the application itself is paused. This would inevitably lead to irresponsive applications so there is a need to modify this approach further in order to make it effective.

3.1.3 Compacting Collector

Once the GC frees some space, it is likely to be fragmented. Compacting collectors are battling the fragmentation by moving live objects towards one part of the heap. This is done simultaneously with marking/sweeping hence leading to no additional delays. Once the algorithm finishes its run, garbage is collected, and all live objects are in one part of the heap leaving an adjacent free space for new allocations.

Moving objects on the heap requires an update to references so that objects point to a correct spot on the heap. There are well known techniques to resolve this issue - a separate table with handles can be used. An object does not refer to the heap but instead refers to a row in a table which then points to a heap. When an objects is moved during collection, only it's handle in a table (one row) will be updated. While compacting collector generated no overhead by moving objects on the heap it does generate some overhead by this reference maintenance.

3.1.4 Copying Collector

Yet another way to deal with fragmentation is introduced by copying collectors. In this case the heap is divided in two parts - part A and part B. Only one part is used at a time. While the program is running; all objects are allocated to one part (A) and once this part is filled, the program stops, and garbage collection begins. The root graph is traced and all live objects are copied to the other part (B) where they ale alligned side by side hence there is no free memory inbetween allocations. This is a notable use (or and extension) of compacting GC. Once the GC finished tracing the graph, the remaining objects in area A are now considered garbage and can be collected. The program then continues running using part B of the heap until it is full again in which case this process repeats.

This proccess is depicted in Figure 3.1; the heap is shown in eight different situations (labelled by numbers). In the first three states the heap only uses the upper half (say part A) and the rest is unused. Once it is completely full (3) the program stops and GC starts searching for live objects. Those are then moved to the other part of the

heap (4) and the remaining objects are considered garbage. The program then continues and stores objects to the lower part of the heap until this part is full again (5-7). Then the proccess repeats and the upper part of the heap will be used again.

There is an obvious drawback in this approach; one needs to have very large heap to maintain this approach. Even twice as big compared to other GC mechanics which can be utilized instead. Furthermore this approach may cause noticeable program pauses as the GC takes place because many objects need to be copied. However copying collectors are very important as they made a way for generational collectors (see section below) which adopted these principles and took them several steps further.

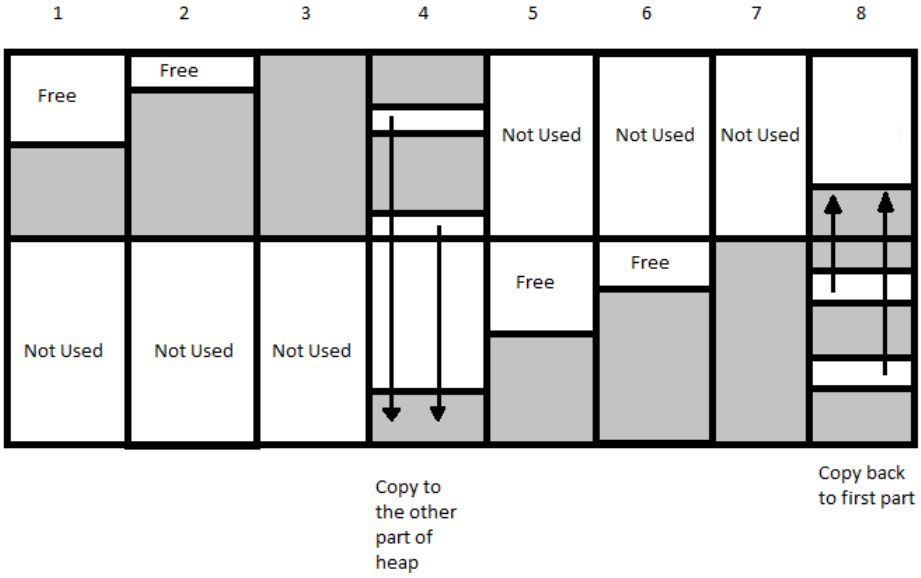

Figure 3.1: Copying Collector Example

3.1.5 Generational Collector

Generational Collectors, as mentioned above, are based on copying collectors but take into consideraition two factors. Firstly, in object oriented languages most objects are short-lived. And secondly, there are usually objects which have very a long lifetime. In Source

code 3.1 there is an example of both - a static final object such as *NOT_GETTING_COLLECTED* variable in the sample below will not get garbage collected unless some extraordinary condition is met (heap corruption or classloader becoming unreachable). On the other hand an Integer object *sum* will only be referenced as long as the method where it was created is executing. Once it comes to an end, this object becomes a garbage.

```
1  public class SimpleSample{
2        // long living object
3        public static final String NOT_GETTING_COLLECTED = "This
             will hang in memory for long time";
4
5        // some method which gets ArrayList of Integers as input
6        public void doSomeWork (List<Integer> input){
7              // short living object
8              Integer sum = 0;
9              for (Integer i : input) {
10                   // do some work
11                   sum += i;
12             }
13             System.out.println(sum);
14       }
15 }
```

Source code 3.1: Long and Short-Lived Objects in Java

With copying collectors, all live objects have to be moved to another location which is time consuming and inefficient. This is exactly what Generational Collectors focus on. Objects can be seen as younger and older based on how many garbage collections they survive. The heap is then respectively divided into two or more sub-heaps, each holding one or more generations of objects. The area with young objects will be garbage collected more often, as most objects are not likely to survive their first collection. Those who survive will be moved to another sub-heap holding older objects; the amount of moved objects is but a fraction of what it was with Copying Collectors. Sub-heaps holding older ('mature') objects will be garbage collected less frequently as these objects are not expected to be a garbage for running program and hence are not candidates for free memory.

The idea of generations is commonly used in current GCs and can be used to improve efficiency of either Copying or Mark and Sweep

collectors.

3.1.6 Incremental Collector

The algorithms for GC have a *disruptive nature* - meaning once they start, the program has to be stopped since it works with the very same data. These delays are ranging from very short to extensive periods of time which are noticeable by application users or may make the application useless in real-time system. With generational collectors these issues get somewhat reduced in importance as the heap is divided; it is possible to make the GC work with just one of the heaps at a time. This kind of GC is known as incremental collector and it is typically a part of generational collector. When there are sub-heaps, they can be garbage collected one by one and the application shall not suffer from long delays - each GC invocation is most likely going to take less time than maximum amount of time (where maximum time is the time needed to collect the whole heap).

Train algorithm was previously used in HotSpot implementation and it allowed for efficient dealing with mature object space (for young object space there were other algorithms). This algorithm is oftentimes mentioned in older literature and throughtout Internet articles/forums as the information on garbage collection is frequently outdated. This thesis will shed some light on it too as it is a good example of how incremental collector works. However the algorithm itself is no longer used; according to Oracle update notes[2], it was replaced by Garbage First collector (known also as G1 collector) in Java 7 Update 4.

Train Algorithm Mechanics

The train algorithm divides heap space into blocks of the same size (which is set in advance). The blocks are then organized into sets; each block belonging exactly to one set. The sets and the blocks within them are organized. The name of the algorithm is a metaphor which was used by original authors (Richard Hudson and Eliot Moss) to explain the mechanics; mature object space can be seen as a railway station while each set of blocks is a train and each block a car. This is depicted by Figure 3.2 which is also used to exaplain the

algorithm.

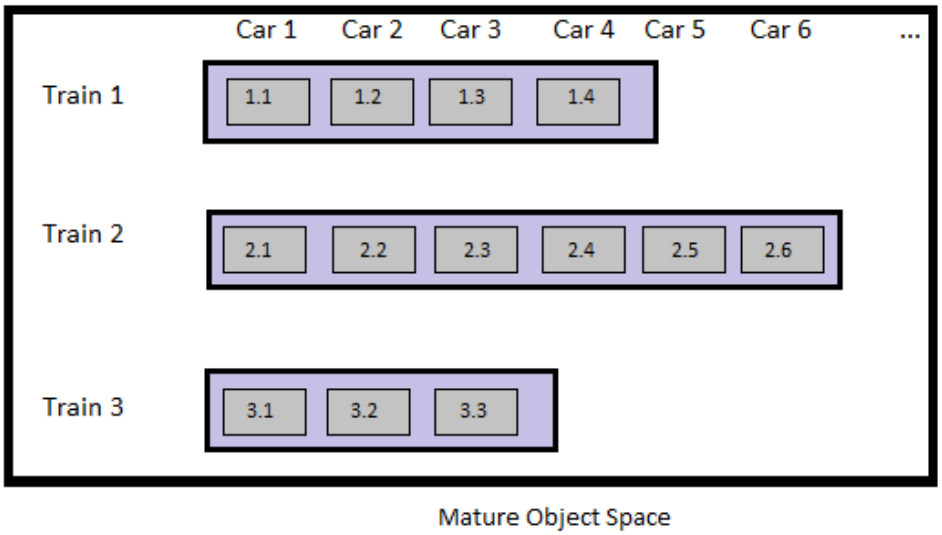

Figure 3.2: Copying Collector Example

In the picture there are three trains, each with different number of cars. The numbers of cars and trains are important because, as mentioned above, the trains (sets) are ordered and so are the cars. The first train to come to the railway station (mature object space) pulls into first track and becomes 'train 1', next train to arrive will become 'train 2' and so forth. When appending cars to trains, one can append them only to the end of the train; this enforces the ordering of cars within each train. This way it is obvious that trains with lower numbers are older and within one train (for instance Train 1), the leftmost car is the oldest one (1.1) while the rightmost car (1.4) is the youngest. To address the labelling of trains and cars in the picture - train are simply annotated by number (1, 2, 3) and car labels consist of a train number, period and car number. To clarify the order of objects in mature object space, car 1.1 preceeds car 1.2 which preceeds 1.3 and so on. But also the last car in the first train (1.4 in the picture) preceeds all cars from the second train. One algorithm invocation always collects only one car, beginning with the lowest number car in the oldest train; in the picture this would be 1.1. Given the order of objects explained above, the very next invocation will attempt to

collect car 1.2 and so forth.

Object which get promoted to mature object space (as they survive several garbage collections) can be added in two different ways. They can be either cars, which will be appended onto any existing train except for the one with the lowest number, or there can be a new train/trains created to hold them.

Every invocation of train algorithm attemps to collect the lowest numbered car or the whole train this car belongs to. Firstly the algorithm finds whether there are any references from outside this train to objects which are anywhere within the lowest numbered train. If no such reference exists, the cars within this train only refer to themselves and the whole train can be garbage collected. If this was the case, the invocation of algorithm stops. This step allows for cycle detection and collecting of large data which would not fit into single block. However, if the train was not collected (e.g. did not contain cyclic data), the algorithm focuses on the lowest numbered car only. It goes through all the objects within this car and moves referenced objects to another car. Once it is done, the remaining objects within this car are considered garbage and the space they occupy is reclaimed and the algorithm returns.

The mechanics of this algorithm guarantee that all cyclic data will end up in the same train which in turn will later become the lowest numbered train and will be garbage collected. This is ensured by the way the algorithm moves the objects. Following rules are applied:

- Any object being referenced from outside the mature object space (e.g. younger object) is moved to another train.
- Any object being referenced from another train is moved into that train.
- Algorithm scans moved objects for any references back to their original train in order to discover any other objects which should be moved.
- Algorithm repeats and ends when none of the objects in the car being collected is referenced by any other train.
- Algorithm will create a new car and append it to the end of the respective train if it runs out of space.

When this process ends objects in the lowest numbered car are not referenced by any other objects in other trains or outside mature object space. However there are still references coming from other cars of the lowest numbered train which need to be resolved before the targeted car can be collected. The movement of objects is very similar:

- Any object referred by objects from other cars is moved to the last car of the same train.
- Algorithm scans moved objects for references back into the lowest numbered car.
- Algorithm moves any newly found referenced objects into the last car.
- Process ends when there are no more references into the lowest numbered car from any other car.
- At this point the lowest numbered car is collected.

The algorithm then uses support structures (remembered sets) to cut down the execution time. This, however, will not be further elaborated as the explanation above allows for sufficient understanding of incremental collector mechanics. To summarize, the algorithm firstly attempts to collect the whole train (to allow for cyclic detection). This will in most cases fail and the algorithm will fall back to collecting the lowest numbered car. It might be able to collect it straight away, or it might need to modify several more cars or even trains before the collection. Nevertheless it attemps to collect the smallest possible unit so as not to be disruptive and in the same time it prepares the heap space for future collecting by moving objects into proper trains and cars.

3.1.7 Adaptive Collectors

Adaptive Collectors take into consideration that under certain circumstances one approach might be better than other. Hence rather than using one specific mechanism, adaptive GC monitors the current heap situation and based on this information it chooses the best

mechanism. This, in practice, means that the algorithm switches in-between several implementations on the fly. It can even choose to separate the heap into several sub-heaps and select a different approach for each of them. An example of this was a generational GC used in HotSpot where Train algorithm was used for mature object heap while other approaches were used for heaps containing younger objects.

3.2 Current GC Implementations

To give a clear view of current implementations, this thesis takes into consideration HotSpot JVM and Java 7 so as to include G1 collector. For clarification, there are no real differences between Oracle's JDK and OpenJDK when it comes to garbage collection and most information used here originate from Oracle website as their documentation is more detailed.

According to Oracle tutorial [3], there are four garbage collectors which can be used - Serial, Parallel, Concurrent Mark Sweep (CMS) and Garbage First (G1). The first three are briefly covered in sections below, the G1 collector is discussed more in depth as Shenandoah is based on it.

Furthermore certain types of collectors can be combined together in way that one collects young generation while the other works with older generation. These options will be mentioned alongside the collectors below.

3.2.1 Serial

Serial GC is a so called 'stop-the-world' algorithm which uses 'mark-compact' method. Stop-the-world means, that once the garbage collection begins, JVM will stop the whole Java application until garbage is collected; no part of GC can run concurrently with the application. Mark-compact method is the above described method for tracing collector combined with compacting collector. Hence after each invocation the heap is traced for living objects which are swept onto one side and a continuous space is left for new allocations. This speeds up further object creation. Other importat aspect of serial GC

is that it uses but one virtual CPU - when talking GC, 'serial' means there is only one thread.

Even though this might at first seem out of date, it still has some use cases. For instance low performance systems with limited memory and cores will make use of this. Oracle also states that "[a]nother popular use for the Serial GC is in environments where a high number of JVMs are run on the same machine (in some cases, more JVMs than available processors!). In such environments when a JVM does a garbage collection it is better to use only one processor to minimize the interference on the remaining JVMs, even if the garbage collection might last longer. And the Serial GC fits this trade-off nicely"[3].

3.2.2 Parallel

Parallel collectors are similar to serial but they use multiple threads. Therefore they still stop-the-world hence arel disruptive in their nature, but the pause times are likely to be shorter than with serial collector.

Usage is very clear - when one needs to do a lot of work (in this case memory cleaning) and can still accept some application delays, parallel collector is the choice. This is for instance batch processing or huge database queries.

Using a computer with X CPUs, this collector will use X threads, if the user does not specify anything else (there are command-line options to change number of threads). In case the user attemps to use parallel collector on a single CPU computer, serial collector (being a default) will be forced instead. Also it takes minimum of three CPUs to make parallel collector more efficient than serial. With two CPUs, they both perform approximately the same.

With parallel collector one can use two different settings:

- *-XX:+UseParallelGC* will use single threaded collector (serial) for older generation, and multiple threads (parallel) for younger generation
- *-XX:+UseParallelOldGC* will use multithreaded collector for both generations. In this case compacting will be used for older generation and copying for younger one.

3.2.3 Concurrent Mark Sweep

Concurrent Mark Sweep collector is commonly known as CMS and it was the most commonly used collector before G1. It is worth noting that CMS focuses on older generation; younger one gets collected in the very same way as with parallel collector. CMS attemps to do as much work as possible without interrupting Java application. Therefore there is usually no compacting/copying and if there is no more space left (due to fragmentation), the algorithm asks for more memory. However, this does not mean that there is no stop-the-world effect. The algorithm will eventually run out of memory and in that moment a copying collector will be invoked (and along with it the whole application will come to a halt). After dealing with fragmentation, the algortihm will go back to CMS approach. The CMS keeps an information about free memory ('freelist') in order to quicken allocation in fragmented space. After each cycle the algorithm updates this freelist.

To shed some more light on how this algorith works, it is good to separate it into phases as described by Jon Masamitsu on Oracle Blog site [4]:

- Initial Mark - **Stop-the-world phase**, algorithm collects root references.
- Concurrent Mark - Aplication starts running again and algorithm traces the graph to find most live objects.
- Precleaning Phase - Optimization phase which monitors changes to objects done while previous phase was running.
- The Remark Phase - **Stop-the-world phase**, CMS needs to stop the application to finally determine all the remaining live objects. This cannot be fully done while the program is still executing as it keeps constantly changing the objects. However, when the algorithm reaches this phase, it has already mapped the majority of live objects and now it only needs to catch-up.
- The Sweep Phase - CMS has a complete list of dead objects and can now determine which space can be freed; in this phase the freelist gets updates.

- The Reset Phase - A clean-up phase; any remaining information about phases and objects are reset and algorithm is prepared for next cycle.

Oracle has it that CMS "should be used for applications that require low pause times and can share resources with the garbage collector. Examples include desktop UI application that respond to events, a webserver responding to a request or a database responding to queries" [3].

3.2.4 Garbage First

Garbage First collector, known more commonly as G1, has several characteristics of CMS, but is way more concurrent and uses entirely different approach concerning heap generation separation. It is considered to be 'one to rule them all' collector, which means it can replace other previously used collectors and still perform well. Amongst other positives, it is less disruptive in its nature and allows user to set GC pause time per each invocation; this is a great improvement for real time applications. Majority of information on G1 was collected from InfoQ blog article[5] and presentation[6] which were both given by the authors of the G1 collector. Another good source (yet not so detailed) is Oracle tutorial website[7].

Heap space in G1 collector is no longer separated into two generations. Instead the whole heap is divided into regions of equal size. The size can vary, ranging from 1MB to 32MB based on total heap size. The aim is to have about 2048 regions. There are five types of regions: eden, survivor, humongous, old generation or unused. Eden is a region where new objects get allocated. When objects age, they get promoted to survivor regions, this can happen several times before they can enter old generation. Eden and survivor regions create a young generation. Objects which have size bigger than 50% of region size get allocated into humongous regions; it is worth noting that algorithm is not optimized for this kind of objects as they should not commonly appear in applications. One can manually set region size to avoid allocating objects into these regions but it is better to investigate why there are so many huge objects and deal with it accordingly. Another region type is reserved for old generation; this is

simply where objects end up if they survive enough collections. Last but not least there are unused/available regions which will be used for future allocations. Figure 3.3 shows a snapshot of heap, with all the above mentioned regions.

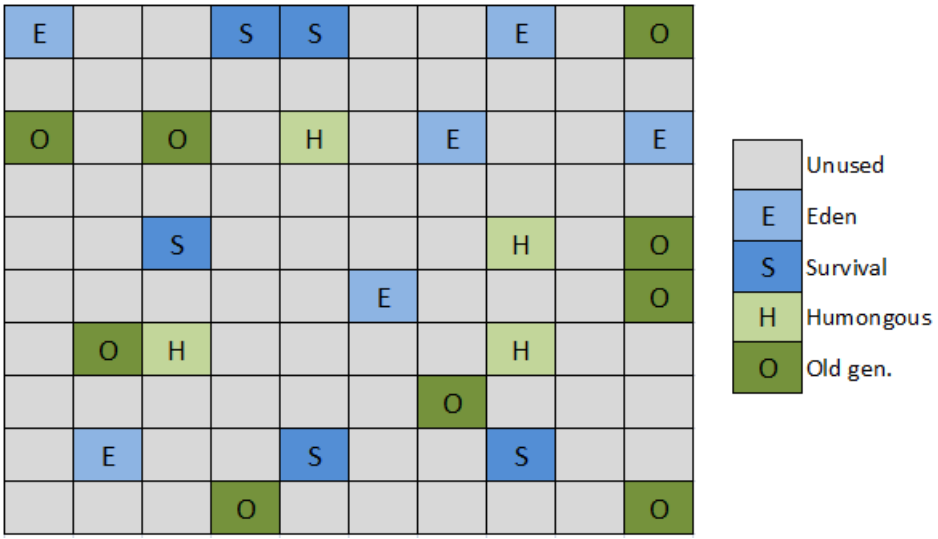

Figure 3.3: Heap structure in G1

There are two important structures in this alrogithm - remembered set (Rset) and collection set (Cset). There is one Rset per each region and it is a structure that tracks references into that region. Cset is then a set of regions which are to be collected in one invocation.

First and the most common event in G1 is young generation collection. It is a STW event, where the algorithm scans eden and survivor regions for live objects (via Rsets, which get updated in the process). Found objects are then copied either to survivor or old generation regions based on the so called 'tenuring threshold', which determines how many collections does the object have to survive in order to be promoted.

When the heap occupancy reaches a certain percent (45% by default, but can be changed) the algorithm starts marking cycle. So far there is no information regarding old generation regions and as it can be huge, it will take some time to complete. This is why most of the work is done concurrently with young generation collection. Hence

avoiding STW pause times as much as possible. The phases of old regions collection are following:

- The Initial Mark - This part is done during STW young collection and it marks the roots.
- The Root Region Scanning - Concurrent with application, scans survivor regions for references into old regions and marks relevant objects.
- The Concurrent Marking - Concurrent with application, algorithm searches for live objects across whole heap. Young collection is expected to interrupt this phase several times before it is completed.
- The Remark - STW phase, finish the marking, revise changes done to objects during previous phase, visit remaining objects. Should be very quick as most of the work has been already done.
- The Cleanup - Partly STW while identifying completely free regions and mixed ones, concurrent during reset and turning empty regions into unused.

After this has been done, all complelety free old regions were already converted into unused regions and there is enough information to collect the rest. Following GC invocations on young generation will be transformed into 'mixed collection'. Mixed collection collects eden and survivor regions (as does ordinary invocation) but on top of that it collects some of the old regions too. Regions with most garbage are prioritized, hence 'garbage first'. The amount of invocations it takes to clear out the whole space depends on the collector pause time; the algorithm simply tries to clear as many as possible in the given time.

4 Shenandoah

Shenandoah is an open source garbage collector developed by RedHat and designed to be a part of OpenJDK. Two lead developers are Roman Kennke and Christine H. Flood who gave most information about this project via WordPress blogs [8][9]. Those blogs along with slides used for presentation at Fosdem 2014 [10] are invaluable sources of information for this thesis. There is also a recording of Strange Loop presentation given by Christine Flood which was later on uploaded to YouTube [11].

4.1 Goals

Shenandoah can be compared to G1 as it uses similar mechanics. However it is not designed to be the 'one to rule them all' collector; Shenandoah focuses only on very large heaps only (100 GB and more) and is likely to have worse performance for small applications. It is presented as 'ultra-low pause time GC' as the aim of the project is to have less than 10 ms pauses for huge heaps. This is achieved by doing concurrent parallel evacuations. The reason why its worth developing such specialized GC is that currently used GCs (G1, CMS) can perform badly once the heap starts growing rapidly; the pauses they make can be disruptive and noticeable to users.

Long term goal for Shenandoah is to make a pauseless collector. While this remains but a dream for now, Shenandoah is still a huge leap in the right direction. As of now there is no other open source project which would offer similar solution.

4.2 Mechanics

As for the heap, it is structured into regions as in G1. The difference is that Shenandoah does not respect the presumption that most objects die young because not all applications follow this rule. Therefore there are no generations and garbage is chosen from all the regions. Apart from this, Shenandoah uses compaction as it avoids fragmentation and fastens allocation. Also Shenandoah works with from-

regions (original region where object resides) and to-regions (objects will be evacuated into these regions) as does G1.

Since Shenandoah is mark-copy collector it deals with these basic phases - marking of live objects and evacuation. Marking is very similar to G1; first of all there is a very short STW phase to scan the root set. After this, the application resumes and GC traces the graph and searches for all live objects. Some objects are likely to change before GC finishes scanning. For this reason snapshot-at-the-beginning (SATB) technique is employed. This is nothing new, in fact G1 uses this as well, but I skipped it previously in order to keep the explanation simple. SATB introduces a write barrier, an action hook which gets called whenever Java application thread attempts to change an object (write into it). The information about this change is stored and presented to GC after it finishes scanning therefore GC has a complete information about all the changes. So now comes a time where all the changed objects are re-visited by GC to see whether they are garbage or not. Once marking is complete, we know what is garbage and which objects are to be evacuated plus since regions are used, we can determine which regions contain most garbage. To sum this phase up, it is more or less the same as G1 except it is not limited to young generation.

Evacuation phase is where the procces gets complicated because both Java and GC threads will attempt to reach the same objects and while objects get copied into to-region one has to be sure that no thread is using the old copy. To achieve this a new level of indirection was added - forwarding pointers. As shown in Figure 4.1 such pointer can point either to the object it belongs to (as with object B) or to its copy in to-region (A -> A') once it has been evacuated. All running threads which attempt to reach any object first go through these pointers. Once object (A) gets evacuated (creating A' in to-region) the pointer of A will now point to the location of A' but the old object (A) still hangs in place and other objects have references to A. This gets resolved via lazy approach; as the GC traverses the heap in marking phase, it also updates these references. After this is done, all objects will have their references updated and from-regions can be safely reclaimed.

Once object gets marked for evacuation it ultimately belongs to from-region and another write barrier appears. Here a simple rule

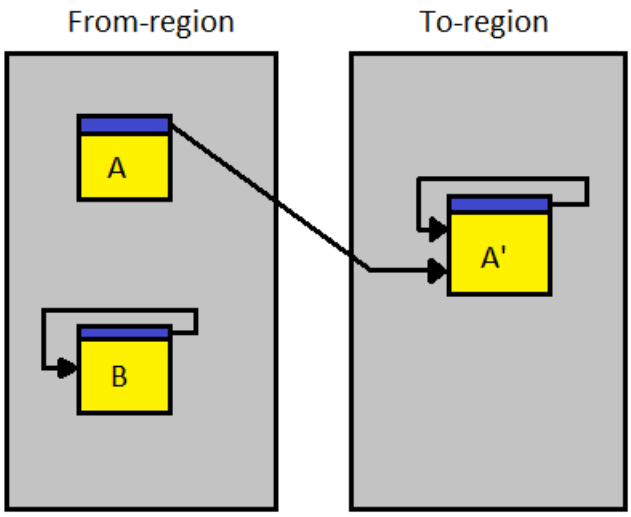

Figure 4.1: Forwarding Pointers in Shenandoah

applies - all writes must occur in to-regions. Now imagine application thread working with these objects. As long as the requested operation is read, everything is fine and access is granted. However if the thread wants to write, the barrier takes effect and forces the application thread to firstly evacuate the objects to to-region and then write into it. Obviously the forwarding pointer will be updated in the process. This means that not only GC but also the application threads themselves can do evacuation of objects. The write barries are a bit more complicated as they need to ensure consistency; when multiple threads attempt to change the same object, write barrier ensures only one will succeed and the others have to rollback.

4.3 Integration with OpenJDK

There is already a JEP for Shenandoah at OpenJDK but it has not been resolved yet. Unless it becomes proper JDK project, it will be hosted on IcedTea servers where one can find all the neccessary materials to build and try this GC with OpenJDK.

5 GC Performance Testing

In this part of the thesis, Shenandoah is put to a performance test and the results are compared with those of other current GC implementations (parallel, serial, CMS and G1 as mentioned above). SPECjbb 2005 was used as a benchmark test because it behaves in the same way at each invocation, assuring same conditions for all collectors. Furthermore it simulates server application in Java while producing heavy load on computer.

5.1 SPECjbb

SPECjbb is a well known Java benchmark software used to measure hardware components. It has several releases, the most recent being 2013. However, for the purpose of this thesis, SPECjbb 2005 was used as it was generously provided by RedHat so the explanation below will stick to the version used.

"SPECjbb2005 is a Java program emulating a 3-tier system with emphasis on the middle tier. Random input selection represents the first (user) tier. SPECjbb2005 fully implements the middle tier business logic. The third tier is represented by tables of objects, implemented by Java Collections, rather than a separate database"[12]. SPEC simulates a company with number of warehouses and starts running transactions. The amount of active warehouses increases as the benchmark proceeds. By default, SPEC will create one warehouse, test it for certain amount of time (30s) and add another one. Once an expected peak of warehouses is reached (detected by HW specs, or set manually) the test duration is increased to 4 minutes per number of warehouses.

Since this way it takes too long to reach a reasonable load (e.g. enough active warehouses), the settings were changed. The benchmark starts at 20 warehouses and climbs up to a maximum of 40. The test duration was set to 3 minutes per number of warehouses and the peak was moved to 20 so that the 3 minutes applied from the very start.

SPECjbb comes in many variations - single JVM instance, multiple or even distributed amongst more computers. But in order to

test garbage collection no more than one JVM is needed; in fact if only one is used, there is likely to be a higher demand on memory. Hence this thesis uses so called 'single-JVM run'. After the test is over, the results are stored in a separate directory. Most information found there is of no use in terms of garbage collection. Instead a log with all GC information can be created and later on analyzed using different tools.

As for requirements to run this benchamark, only Java is needed. Because Shenandoah is not yet a part of OpenJDK, one needs to build his own JDK distribution. Such distribution contains not only Shenandoah but all the other collectors needed for this testing making it a viable choice for this thesis. IcedTea (where Shenandoah is hosted at the moment) has a JDK8 and JDK9 branch. JDK9 was chosen as at the time of testing it included slightly better GC logging for Shenandoah.

Figure 5.1 shows a piece of console output from the running benchmark. At the top one can see that threads are started for each warehouse (in this case there are 38 warehouses already); soon after a timer is started and as explained before, it runs for three minutes. Finally at the lower part of the picture there are the results. Interesting part is the "Heap Space" column (lower right corner), which monitors heap space every three minutes, giving a rough information about memory usage. Rest of the columns are more or less self-explanatory, only stating how many certain actions were taken and how much time did it take. There is one 'magic' number though - throughput. This is a special number calculated from all the other data and would be used to compare the result of the benchmarking computer to other computers if one was to publish the result. Since all the tests in this thesis are run under same conditions (identical computer, heap space and Java), this number can be used to see how different GCs affect application throughput. Several runs with the same GC showed that the throughput is more or less the same number.

As a part of the SPEC, the application forces two full GC collections to make sure that everything is running smoothly. This takes place shortly before initiating warehouses, e.g. before the testing itself begins. These collections have the longest pause times, and it is taken into consideration in the tests below because forcibly calling

System.gc() is very bad practice in any Java application.

```
 started user thread for Warehouse 23
 started user thread for Warehouse 24
 started user thread for Warehouse 25
 started user thread for Warehouse 26
 started user thread for Warehouse 27
 started user thread for Warehouse 28
 started user thread for Warehouse 29
 started user thread for Warehouse 30
 started user thread for Warehouse 31
 started user thread for Warehouse 32
 started user thread for Warehouse 33
 started user thread for Warehouse 34
 started user thread for Warehouse 35
 started user thread for Warehouse 36
 started user thread for Warehouse 37
 started user thread for Warehouse 38
Timing Measurement began Wed Mar 25 07:20:33 EDT 2015 for 3 minutes
Timing Measurement ended Wed Mar 25 07:23:33 EDT 2015


Calculating results

Minimum transactions by a warehouse = 3500810
Maximum transactions by a warehouse = 5818998
Difference (thread spread) = 2318188 (39.84%)

================================================================================
TOTALS FOR: COMPANY with 38 warehouses
................. SPECjbb2005 1.07 Results (time in seconds) .................
                  Count     Total    Min     Max     Avg    Heap Space
    New Order:  68334762   1802.95   0.000   0.072   0.000   total   153600.0MB
      Payment:  47125945    773.50   0.000   0.072   0.000    used    11018.8MB
  OrderStatus:   4712594    111.06   0.000   0.071   0.000
     Delivery:   4712590   2829.12   0.000   0.074   0.001
  Stock Level:   4712592    191.44   0.000   0.055   0.000
  Cust Report:  25917103   1000.06   0.000   0.072   0.000

          throughput =     863965.88 SPECjbb2005 bops

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Figure 5.1: SPECjbb 2005 Output

5.2 Hardware And Environment

The computer used for this benchmarking is a Dell PowerEdge R720; access was granted by RedHat. Here is a brief list of the specifics, more in-depth information can be found on vendor's website:

- CPU - Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
  - Cores: 20
  - Processors: 40
- Memory - 258285 MB

- Disk - 250 GB
- Operating System - Red Hat Enterprise Linux 7.1 (workstation)
- Java - built from JDK9 branch of Shenandoah (20.3. 2015)

Basically, any system with big memory (more than 32 GB) and Linux based operating system would be just fine. Linux is mandatory since Shenandoah can be only used via OpenJDK which cannot be installed on Windows or Mac machines. Demands on memory capabilities are high because the test should show how Shenandoah, which is designed for huge heaps, deals with them in comparison to other collectors.

5.3 Performance Monitoring Tools

Because testing is done remotely on a different computer (using SSH connection), tools such as JConsole proved to be difficult to use. Most reasonable way is to store a GC log files which can later be analyzed. The problematic part is that the tests are executed with JDK9 which is not yet supported by most tools and the logs it creates are slightly different hence a lot of tools fail to recognize it. This was a case with G1 collector and for this reason the logs were tampered with to resemble those created by JDK7, making it parseable.

GCMV [13] was used as a monitoring tool. It was developed by IBM and it can be added as plugin into Eclipse IDE. This tool was chosen as it had the least problems with log files generated by JDK9 branch - the only problematic output was that of a G1 collector for which there is a workaround.

Other tools had problems recognizing the logs from most GCs mainly because the Java used was version 9. In some rare cases it was due to the fact that the JDK did not come from official distribution but was self-compiled instead.

5.4 Shenandoah GC Output

Shenandoah, as it is still under development, does not provide full scale logging. In fact the only thing it did was to log a summary of

its action when the application terminated. From such information it was clear how much time was spent in a STW phases and how much was done concurrently. However, one cannot determine the number of collections and there is no information about memory.

After contacting the developers (via IcedTea mailing list [14]) it became clear that as of now Shenandoah does not implement the necessary APIs. There was also a discussion about the way memory is managed in this GC. The conclusion was that any easily obtainable information about memory would most likely be inaccurate. However based on this conversation Christine H. Flood implemented slightly more verbose GC log on JDK9 branch which was then used for testing. This allows to at least see time information about each invoked collection separately.

Due to the above mentioned difficulties, Shenandoah cannot be analyzed by any existing tool. All the information about memory is extracted from SPECjbb output and cannot be linked to separate collections. The timing information however should be sufficient as Shenandoah is aiming not for maximum throughput but for minimum pause times.

5.5 Command Line Options

In order to keep the benchmark as simple and generic as possible I chose not to optimize either of the collectors. Many of them offer options which would allow to scale certain generation size, or maximum pause time (G1), or number of threads (parallel collector). However using these options would require to seek for optimal settings for each and every collector and would require a lot more testing and time and since access to testing computer was limited I could not do that. Following JVM options were used for the tests:

- *server* uses advanced compiler, tuned for server side applications
- *-Xms150g* sets initial heap size to 150GB
- *-Xmx150g* sets maximum heap size to 150 GB

- *-XX:+UseDesiredGC* picks the GC to be tested (for instace -XX:+UseG1GC)
- *-Xloggc:gc.log* redirects GC output to log file
- *-XX:+PrintGCDetails* enables GC output
- *-XX:+PrintGCTimeStamps* adds time stamps to GC output
- *spec.jbb.JBBMain -propfile SPECjbb.props* mandatory commmands, required by SPEC, which only point to property files

5.6 Test Results per Garbage Collector

Test results for each collector are presented by several graphs generated via GCMV and by extracting information from SPEC results. The comparison and conclusion of the benchmark is located in a separate section below. Minimum and maximum values in nearly all tables are caused by initial full GC system call. They are intentionally kept in statictics as they show how effectively each collector deals with such problem, even though, as mentioned above, using full GC is not advisable. Following information is tracked whenever it is obtainable: used heap (before/after collection), nursery size, pause time, STW time. With some collectors, additional information is included.

Used heap is a self explanatory aspect. Nursery is (with generational collectors) a space for new object allocation; once this space gets full, a collection is invoked. SPECjbb proved to allocate a large number of new short living objects hence triggering a lot of nursery-wide collections, while leaving almost no objects in older generations. Pause time is explained for each collector separately as it sometimes equals STW and sometimes it does not. STW is the most important information; it is the actual time for which the application comes to a halt. This is what Shenandoah aimes to cut down as much as possible.

Used heap (before collection)

Mean	Minimum	Maximum
heap (GB)	heap (GB)	heap (GB)
3.92	0.0	12.3

Used heap (after collection)

Mean	Minimum	Maximum
heap (GB)	heap (GB)	heap (GB)
2.1	0.0	10.5

Nursery size

Mean	Minimum	Maximum
heap (GB)	heap (GB)	heap (GB)
2.05	2.05	2.05

Amount freed

Mean	Minimum	Maximum	Total
heap (GB)	heap (GB)	heap (GB)	heap (GB)
1.82	0.0	1.94	16435

Pause time

Mean	Minimum	Maximum	Total
time (ms)	time (ms)	time (ms)	time (ms)
160	138	1956	1447932

Figure 5.2: Table Data for CMS Collector

5.6.1 CMS Collector

Figure 5.2 shows basic infromation extracted from CMS log. This GC works with nursery which had a size of 2.05 GB (chosen by collector itself). This led to a rather extensive amount of collections - 9 036 collections triggered by allocation failure. To clarify, the term 'allocation failure' means that running application attempted to create new object. Such object would land in nursery area. Nonetheless this area is already full, hence the failure to allocate. Because of the nursery size, most of the heap stays unused during execution; one can see this in the 'Used heap' tables. Mean heap usage before collection was slightly below 4 GB and afterwards it was 2.1 GB. Even when the heap size was growing steadily as there were more and more warehouses added, the amount of space freed by each collection nearly equaled nursery size. The difference between nursery size and amount freed are the objects which were promoted to older generation.

Pauses in CMS are STW in case of young generation (parallel collector is used there) and partially STW when it comes to older gen-

eration. However amongst all collections there was not a single old generation collection invoked because there was no need for it. The heap was large enough and since SPECjbb generates primarily short-living objects; majority of them does not live long enough to get promoted. While the mean pause time (160 ms) is quite good, the total time spent on collecting is huge. One has to consider that in 160 ms the collectors barely cleared 2 GB of space.

Figure 5.3 contains a graph showing all the above discussed attributes on a time axis. It is clear that the heap usage grows higher as the test proceeds; the amount freed stays the same all the way through, as only nursery gets collected. Within pause times, there are several visible spikes but the differences are not as significant as it may seem. While minimal pause time is 138 ms, maximal is 283 ms (ignoring full GC collection at start).

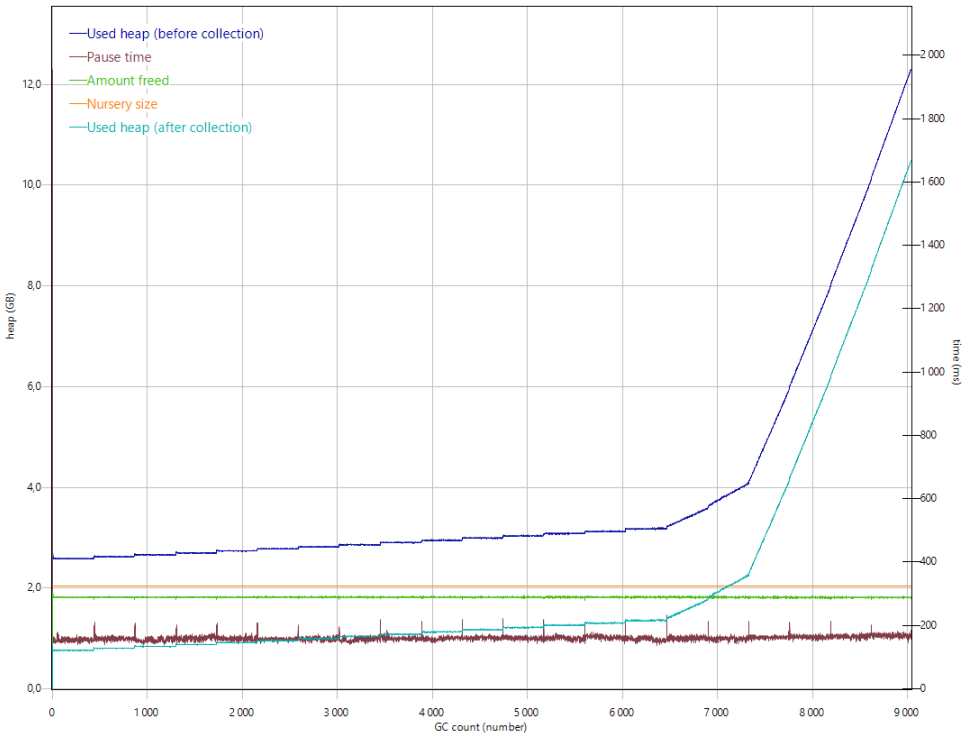

Figure 5.3: Line Plot for CMS Collector

According to SPECjbb results the test run with CMS had a 446

038 business operations per second (bops) This throughput was very steady throughout the whole test, ranging from a minimum of 459 218 bops (at 40 warehouses) to a maximum of 476 011 bops (at 34 warehouses).

5.6.2 G1 Collector

G1 GC log file was modified to resemble a log created by Java 7/8. This means removing numbering in front of each collection and also deleting redundant information about pause. With these modifications, log file can be parsed by GCMV (and other analyzers such as GCViewer).

Used heap before gc

Mean	Minimum	Maximum
heap (GB)	heap (GB)	heap (GB)
87.8	0.0	91.3

Stop the world pause time

Mean	Minimum	Maximum
time (ms)	time (ms)	time (ms)
35.9	12.9	746

Used heap after gc

Mean	Minimum	Maximum
heap (GB)	heap (GB)	heap (GB)
1.11	0.0	1.56

Pause time

Mean	Minimum	Maximum	Total
time (ms)	time (ms)	time (ms)	time (ms)
35.9	12.9	746	12468

Figure 5.4: Table Data for G1 Collector

Compared to CMS, G1 collector behaves very differently. As seen in Figure 5.4, mean heap usage was 87.8 GB, and this dropped all the way to 1.11 GB after GC. Collections were only executed on young generations, as SPECjbb created short living objects - by the end of execution, there was merely 0.25 GB of memory used by objects in older generations (survivor area), meaning most objest were considered garbage. But G1 collector adapts to this behavior; it uses 'Eden' as an area where new objects are allocated and it can modify its size according to the application's needs. First ten collections were executed on a heap size of about 8 GB and collections were invoked every second. Then there was a collection on 30 GB heap and right

after the heap increased to 90.6 GB where is remained for the rest of the SPECjbb test run. The intervals between GC invocations increased to about twelve seconds. This adaptation happened during the first thirty seconds including the time for full GC called in the very beggining.

There were 346 collections invoked during the test; the first two were full GC collections and were the longest ones making application stop for 746 ms. From Figure 5.4, mean STW pause was 35.9 ms. That is a fairly good result considering that nearly 90 GB of memory was freed at a time. Minimum STW pause was 12.9 ms and it occured during the heap adaptation phase in the beggining where only 8 GB heap was collected.

Figure 5.5 nicely depicts the immense heap growth in the beginning. It also shows that heap usage after collection increased very slowly. At first glance the pause times seem to contain many spikes, while in fact the minimum time to collect 90 GB heap was 21.1 ms and the longest one was 71.2 ms.

As for SPECjbb throughput, the result was 817 718 bops - a high ranking result indeed.

5.6.3 Parallel Collector

Parallel collector, triggered by *-XX:+UseParallelGC*, uses parallel collection for young generation only (nursery). Another algorithm will be used for older generation in order to avoid excessive pause time. When talking about GC throughput (e.g. how much memory can a GC during certain time), parallel collector is by far the most efficient way. However it has a disruptive nature caused by long STW phases.

Figure 5.6 shows that the nursery was at first set at 43.8 GB and was later increased to 49.9 GB. Mean pause time was 14.6 ms and most of the collection times were almost identical. At first two full GC collections were invoked, both of which were accompanied by separate nursery collection - that is where the minimal pause time originated. The interesting part is the maximum time (261 ms) for collection - it does not belong to full GCs at start (those took 205 and 212 ms); in fact it is an ordinary nursery collection with a heap of about 48 GB. Minimal time for this size of heap is a mere 7.72 ms. Those are huge deviations from mean time and prove that a paral-

Figure 5.5: Line Plot for G1 Collector

lel collector can cause enormous STW pauses but it also has high throughput.

Used heap (before collection)

Mean	Minimum	Maximum
heap (GB)	heap (GB)	heap (GB)
50.5	0.0	51.5

Pause time

Mean	Minimum	Maximum	Total
time (ms)	time (ms)	time (ms)	time (ms)
14.6	1.62	261	9415

Used heap (after collection)

Mean	Minimum	Maximum
heap (GB)	heap (GB)	heap (GB)
1.19	0.0	1.81

Nursery size

Mean	Minimum	Maximum
heap (GB)	heap (GB)	heap (GB)
49.8	43.8	49.9

Figure 5.6: Table Data for Parallel Collector

In Figure 5.7 one can see that at the very beginning pause times were hugely diverging - this is where full GCs were invoked. For parallel collector, each invocation such as this has to consist of a nursery collection plus an old generation collection, because each of these generations is managed by different collector. Obviously, nursery heap is smaller and so it takes less time to collect, than older generation. The rest of the high peak is the first ten collections where heap size was being adjusted. After that the pause time more or less closes to the mean time of 14.6 ms. The total number of GCs triggered by allocation failure was 645 and total time spent in collections was 9 415 ms.

SPECjbb results indicates a very good performance, 862 694 bops.

5.6.4 Parallel Old Collector

Parallel Old Collector, invoked by *-XX:+UseParallelOldGC*, is a collector which forces a parallel collection for both generations - young and old. This can lead to even longer pause times, as old generations can considerably increase in size. On the other hand it might increase the throughput. This collector can be useful as an optimization for batch processing for instance.

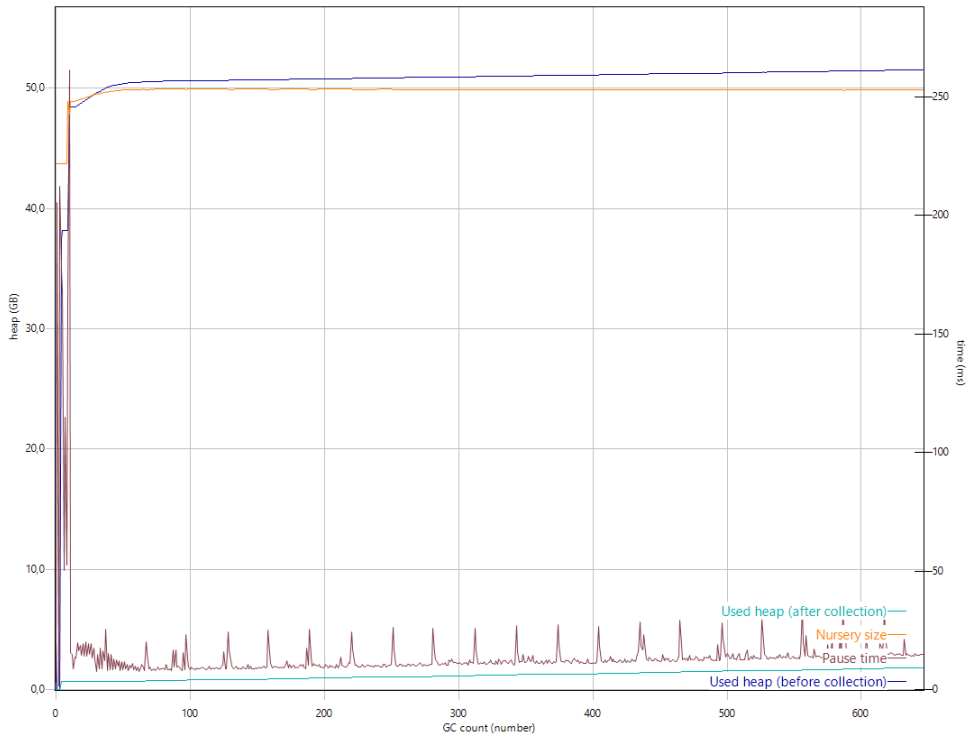

Figure 5.7: Line Plot for Parallel Collector

From Figure 5.8 and 5.9 one can see that the numbers are almost identical to those of Parallel GC above. So is the nursery and heap usage before/after collection. This is due to SPECjbb's nature. As it creates piles of short living objects which get collected at the first occasion, tenured heap space does not get populated enough to trigger any collection. Hence the results in the table and the graph below are but a proof that multiple equal runs of test setup used within this thesis give almost identical results.

Used heap (before collection)

Mean	Minimum	Maximum
heap (GB)	heap (GB)	heap (GB)
50.5	0.0	51.5

Nursery size

Mean	Minimum	Maximum
heap (GB)	heap (GB)	heap (GB)
49.8	43.8	49.9

Used heap (after collection)

Mean	Minimum	Maximum
heap (GB)	heap (GB)	heap (GB)
1.2	0.0	1.81

Pause time

Mean	Minimum	Maximum	Total
time (ms)	time (ms)	time (ms)	time (ms)
14.5	1.35	262	9323

Figure 5.8: Table Data for Parallel Old Collector

Last but not least, the SPECjbb throughput rating was 856 519 bops. Again very similar to previous run, showing that the metrics given by SPEC are reliable - the deviation is less than 1%.

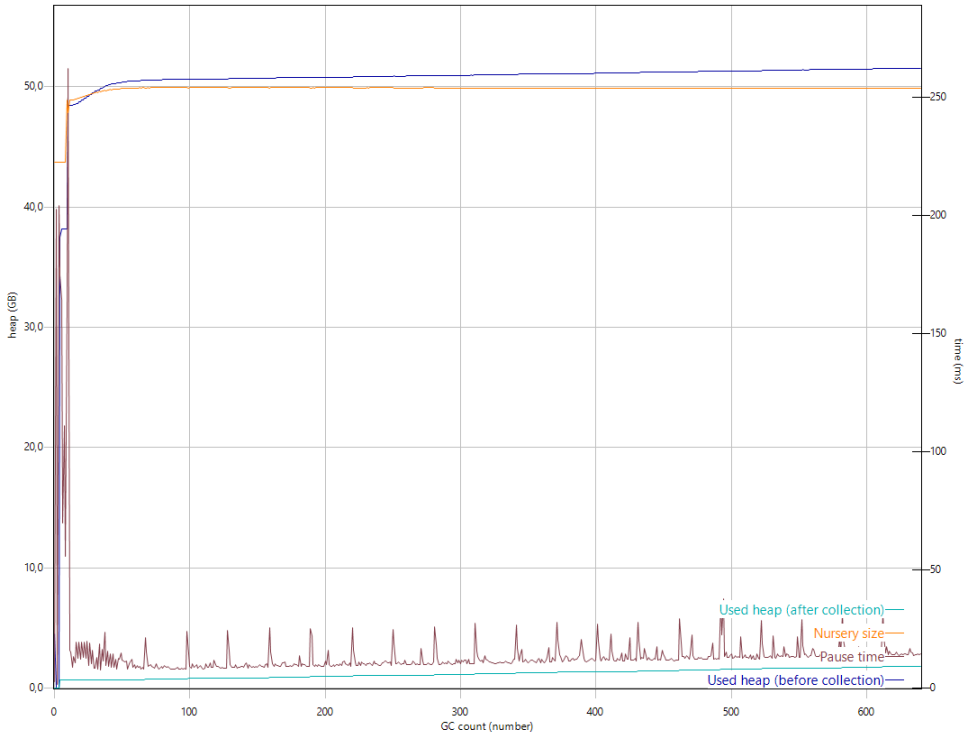

Figure 5.9: Line Plot for Parallel Old Collector

5.6.5 Serial Collector

Serial collector is a predecessor to parallel collector, or to be exact to the parallel old collector. The whole heap is collected in a STW pause using a single thread. This makes it efficient in slower environments (with one processor); furthermore there is no need for thread communication which removes some additional overhead. Nevertheless, as expected, the performance should be worse in the scenario used in this thesis.

From figure 5.10 one can see that the heap usage was around 41 GB while the nursery has the size of 45 GB. Hence serial collector starts collecting when there is still some memory available. The pause times are again the most interesting - surprisingly full collections in the beginning are the shortest ones (11.3 and 4.98 ms respectively). This might be due to the single thread being used and the fact that the heap was more or less empty, only 3.2 GB were used when the second full GC was invoked. Single thread can easily trace the heap to find live objects when it is this small. Maximum time (1 035 ms) is a first nursery collection in the whole test. Total time spent in collections is 105 140 ms which is way more than in case of parallel collector.

Used heap (before collection)

Mean	Minimum	Maximum
heap (GB)	heap (GB)	heap (GB)
40.9	0.0	41.4

Pause time

Mean	Minimum	Maximum	Total
time (ms)	time (ms)	time (ms)	time (ms)
177	4.91	1035	105140

Used heap (after collection)

Mean	Minimum	Maximum
heap (GB)	heap (GB)	heap (GB)
1.04	0.0	1.43

Nursery size

Mean	Minimum	Maximum
heap (GB)	heap (GB)	heap (GB)
45.0	45.0	45.0

Figure 5.10: Table Data for Serial Collector

Figure 5.11 indicates that heap usage never reaches the limit of nursery. GC is called sooner, even though the reason for collection is still the allocation failure (e.g. no free memory to create a new object).

Regarding pause times there is a noticeable growth as the GC count increases. Also the divergence is much higher, individual plots can differ by up to 50 ms. On the left side, there is the spike reaching over 1 000 ms, which was mentioned in previous paragraph.

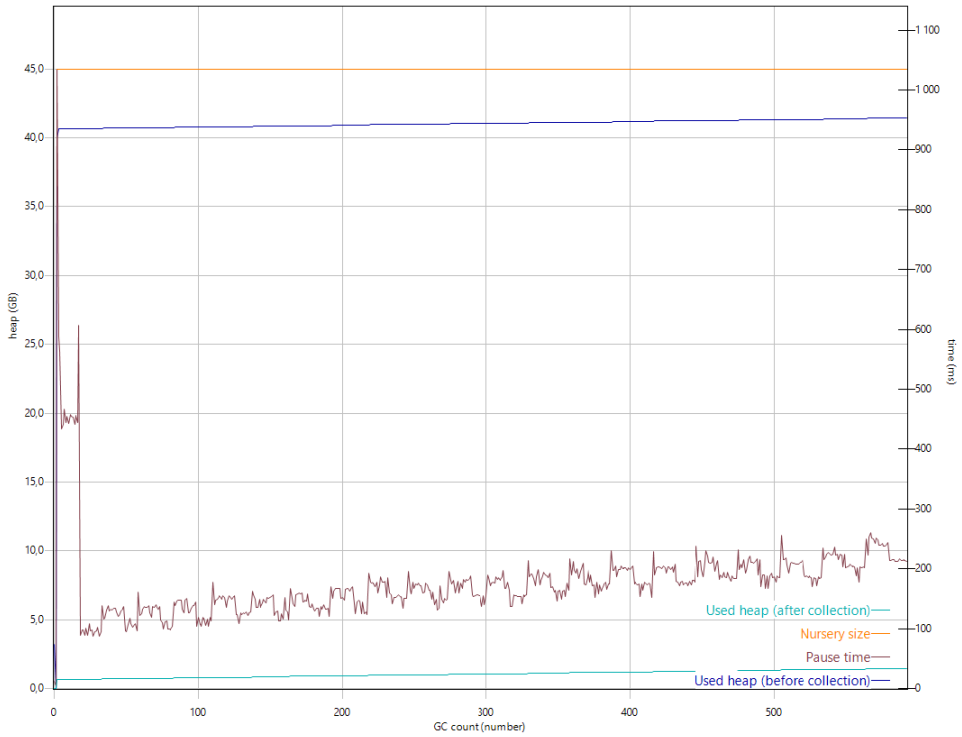

Figure 5.11: Line Plot for Serial Collector

SPECjbb result is 636 289 bops. Right after test start, there is a severe performance drop to less than 600 000 bops. Performance starts to increase when the test reaches 29 warehouses. From this point on, there is a steady growth in throughput.

5.6.6 Shenandoah Collector

Shenandoah test results were left as last in this review as they are a bit trickier to interpret. Due to the absence of tool which could analyze the log, figures were added with rough data, followed by explanations. Focus is mainly on pause times, as Shenandoah claims to be an

'almost pauseless collector'.

As mentioned before, there is no direct information on memory usage, the only available information is gathered from SPECjbb result page, where heap was monitored after each run with given amount of warehouses. In most cases the memory was about 3.5 GB, but in three cases it went way higher. Once it was over 31 GB and two times it even hit more than 80 GB. Furthermore, from the log file one can read that there were 96 collections in total (excluding two full GCs in the begging). These information combined together imply that the collector is by no means eager and rather waits for heap occupancy to reach a certain high threshold before it is invoked.

Each collection in Shenandoah generates a versobe output in the log. One such invocation can be seen in Figure 5.12 - each phase is marked by at least two lines, start and end. Starting from the top there is *InitMark* and *ConcurrentMark*. These two are fairly common and known from other collectors; it is the marking phase in which collector traces the the heap and looks for live objects. *FinalMark* is the last marking phase; upon reaching this phase, the application needs to be stopped to catch up with changes done during previous concurrent scanning and also starts object evacuation. *RescanRoots* is where graph roots are rescanned in case there was a change, then there is *DrainSATB*, *DrainOverflow* and *DrainQueues* which are all linked to buffers created before/during marking phase and which now need to be emptied. Evacuation is prepared, areas where live objects will be moved into are chosen. Also *WeakRefs* (weak references) are resolved here. After these tasks, *FinalMark* ends and *ConcurrentEvacuation* takes place, followed by *FinalEvacuation* which, similarly to *FinalMark*, needs to stop the app to finalize evacuation and make sure everything goes smootly. Final stage is reference updating (*FinalUpdateRefs*); roots get updated (*UpdateRoots*) as well as empty regions get recycled (*RecycleRegions*). Lastly there is a TLAB resizing phase; TLAB stands for thread local allocation buffer. For completness the number on the left side of each line is test execution time in seconds and the the number *#50* is a number of collection.

Noticeable part in Figure 5.12 is the full GC invocation; it happened 44 times during the whole execution (excluding two user called GCs in the beginning). A number equal to amount of allocation failures (see Figure 5.14). In case of Shenandoah, allocation fail-

```
1906.065: #50: [GC InitMark start]
1906.070: #50: [GC InitMark end, 0.005081 secs]
1906.070: #50: [GC ConcurrentMark start]
1912.781: #50: [GC ConcurrentMark end, 6.711048 secs]
1912.782: #50: [GC FinalMark start]
1912.782: #50: [GC RescanRoots start]
1912.784: #50: [GC RescanRoots end, 0.001664 secs]
1912.784: #50: [GC DrainSATB start]
1912.785: #50: [GC DrainSATB end, 0.001031 secs]
1912.785: #50: [GC DrainOverflow start]
1912.785: #50: [GC DrainOverflow end, 0.000004 secs]
1912.785: #50: [GC DrainQueues start]
1912.787: #50: [GC DrainQueues end, 0.002456 secs]
1912.787: #50: [GC WeakRefs start]
1912.809: #50: [GC WeakRefs end, 0.022195 secs]
1912.809: #50: [GC PrepareEvac start]
1912.809: #50: [GC PrepareEvac end, 0.000197 secs]
1912.809: #50: [GC InitEvac start]
1912.810: #50: [GC InitEvac end, 0.001016 secs]
1912.810: #50: [GC FinalMark end, 0.028635 secs]
1912.811: #50: [GC ConcurrentEvacuation start]
1915.363: #50: [GC ConcurrentEvacuation end, 2.552887 secs]
1915.365: #50: [GC FinalEvacuation start]
1915.367: #50: [GC FinalEvacuation end, 0.001467 secs]
1944.407: #50: [GC FinalUpdateRefs start]
1944.407: #50: [GC UpdateRoots start]
1944.421: #50: [GC UpdateRoots end, 0.014121 secs]
1944.421: #50: [GC RecycleRegions start]
1944.421: #50: [GC RecycleRegions end, 0.000069 secs]
1944.421: #50: [GC ResizeTLABs start]
1944.421: #50: [GC ResizeTLABs end, 0.000015 secs]
1944.421: #50: [GC FinalUpdateRefs end, 0.014303 secs]
1944.421: #51: [GC FullGC start]
1944.424: #51: [GC ConcurrentMark start]
1944.543: #51: [GC ConcurrentMark end, 0.118625 secs]
2016.867: #51: [GC FullGC end, 72.445820 secs]
```

Figure 5.12: Single Collection in Shenandoah

ure equals to all of the heap space being depleted - there are no generations. As the test proceeded it grew more frequent up to a point where every usual collection was followed by full GC invocation. After contacting the developers it became clear that this is a bug they are trying to resolve. Shenandoah ocassionaly runs out of memory, which should be avoided but sometimes it happens nevertheless. The cause might the memory overhead of pointers (approximately 20%) or bad heuristics which fail to fire GC in time. Besides when allocation failure happens during usual collection, the collection ought to be stopped immediately and full GC should be invoked; but at the moment it finishes usual collection and then invokes full GC. Just this happened in the log showed in Figure 5.12 and it is likely to create a noticeable performance loss. The positive thing is, that the bug is already known and being addressed. Knowing about this bug, one has to reconsider the amount of collections fired in the whole test run; if there were 44 allocation failures and 96 collections is total, then 44 collections were only partially useful as they were immediately followed by triggered full GC. Judging from other collector results, average time to clear a full heap can be several seconds, but cannot take nearly a minute. It is more likely that the log file has incorrect information about full GC collection as Shenandoah is still under development and even the information displayed now were added after I contacted the developers.

At the end of the log file, there is a summary of all pauses made by the collector; see Figure 5.13 for this output. The phases we already described and as one can see, not all of them were disruptive. Each pause states a total and average time spent in that phase. The only vital pause is the *FinalMark*, which is by far the longest with a duration of 29.56 ms in average and 1.54 s in total. There is no note about full GC pause/time but it is clear there has to be some or rather that the full GC is one big STW phase because after all, there is no free space left for allocation.

Figure 5.14 contains the very end of a log file with a summary of time spent in concurrent and STW phases. It is here that one can easily see that Shenandoah spent in total only 2.0 seconds in STW phases. Other phases were more time demanding, but they were concurrent - marking time (307.67 s), evacuation time (124.18 s). Lastly there is an information about full GC time which was in average 56

```
Initial Mark Pauses          =     0.16 s (avg =     3.00 ms)
(num =    52, std dev =     1.55 ms, max =    12.84 ms)
Final Mark Pauses            =     1.54 s (avg =    29.56 ms)
(num =    52, std dev =     2.44 ms, max =    36.93 ms)
  Rescan Roots                 =     0.10 s (avg =     1.90 ms)
  (num =    52, std dev =     0.36 ms, max =     3.03 ms)
  Drain SATB                   =     0.06 s (avg =     1.08 ms)
  (num =    52, std dev =     0.08 ms, max =     1.27 ms)
  Drain Overflow               =     0.00 s (avg =     0.01 ms)
  (num =    52, std dev =     0.01 ms, max =     0.05 ms)
  Drain Queues                 =     0.17 s (avg =     3.31 ms)
  (num =    52, std dev =     2.16 ms, max =    10.23 ms)
  Weak References              =     1.09 s (avg =    20.98 ms)
  (num =    52, std dev =     0.43 ms, max =    22.19 ms)
  Prepare Evacuation           =     0.01 s (avg =     0.20 ms)
  (num =    52, std dev =     0.02 ms, max =     0.28 ms)
  Initial Evacuation           =     0.10 s (avg =     2.01 ms)
  (num =    52, std dev =     0.89 ms, max =     3.88 ms)
Final Evacuation Pauses      =     0.08 s (avg =     1.50 ms)
(num =    52, std dev =     0.20 ms, max =     2.34 ms)
Final Update Refs Pauses     =     0.23 s (avg =     4.46 ms)
(num =    52, std dev =     4.73 ms, max =    19.38 ms)
  Update roots                 =     0.23 s (avg =     4.36 ms)
  (num =    52, std dev =     4.70 ms, max =    19.22 ms)
  Recycle regions              =     0.00 s (avg =     0.04 ms)
  (num =    52, std dev =     0.02 ms, max =     0.10 ms)
  Resize TLABs                 =     0.00 s (avg =     0.02 ms)
  (num =    52, std dev =     0.01 ms, max =     0.08 ms)
```

Figure 5.13: Summary of Pauses in Shenandoah

s. Even serial collector can clean the whole heap is less time. The reason it took so long was yet another bug in Shenandoah. For finding live objects during full GC, Shenandoah uses concurrent marking which makes no sense. The application is stopped so there is no reason for concurrency; using simple parallel collector for full garbage collection would be much better option. This bug is also a known and being resolved.

```
Concurrent Marking Times     =   307.67 s (avg =  3139.52 ms)
(num =    98, std dev =  3287.32 ms, max = 10220.65 ms)
Concurrent Evacuation Times =   124.18 s (avg =  2388.01 ms)
(num =    52, std dev =   539.81 ms, max =  4144.33 ms)
Full GC Times                =  2615.78 s (avg = 56864.76 ms)
(num =    46, std dev = 17779.34 ms, max = 90344.19 ms)
User requested GCs: 2
Allocation failure GCs: 44

Total                        =     2.00 s, avg =     9.63 ms,
max =    36.93 ms
```

Figure 5.14: Shenandoah Log File Summary

The last thing to discuss are the SPECjbb results. Unsurprisingly, Shenandoah ranked very low with mere 117 063 bops. At first, with 20 warehouses the throughput was 227 846 bops, but this number was gradually lowering to approximately 160 000 bops and later on, beginning with 24 warehouses, it dropped to about 100 000 bops. Performance drop is most likely connected to full GC invocations becoming more and more frequent. Nevertheless, even if it was not for this bug, Shenandoah was not expected to rank first in application throughput - it forces application threads to evacuate some of the objects on the heap as the execution proceeeds. This will inevitably cause a performance drop but on the other hand it lowers the STW time collector created hence leaving more time for the execution itself. Having some extra time to perform more operations might increase the throughput again, but that is yet left to be discovered once the bugs are fixed.

5.7 Test Results Comparison

In this section, three major aspects of the collectors are compared: heap usage, pause time and SPECjbb throughput. Parallel old collector was left out because of its results being identical to those of parallel 'new'. Furthermore the data displayed in the following tables exclude information related to the first two full GCs invoked by the application.

5.7.1 Heap Usage Before GC

Heap usage is important in order to see if the collector takes advantage of big heap space. For instance Figure 5.15 shows that CMS collector failed - it used mere 12.3 GB at most while it had 150 GB at its disposal. Generations have to be taken into consideration; collectors separate heap space in between collections and CMS collector probably left most space for tenured objects. On the other hand, G1 collector did a very good job adjusting the heap size for younger generation which dominated this test. It started at 7.48 GB and went up to 91.3 GB used. Parallel and serial collectors did not show any big dynamic adjustments of heap size although they both chose to

allocate a considerable part of the heap to nursery. Mean heap usage indicates that collectors which adjusted their heap sizes did so very rapidly, as the mean size more or less equals max size. With Shenandoah there is no information about mean and minimum heap sizes, but one can deduct that the max size was the whole heap. The reasoning behind this is that there are no generations, hence no heap separation. In such situation, when allocation failure occurs, it means that the whole heap space is exhausted. There probably will be some small reserve but the total heap usage will likely be higher than with G1. Shenandoah is designed for huge heaps and therefore it is expected that it takes advantage of all the heap space. From heap usage viewpoint, Shenandoah would be the winner and G1 is ranking second.

	CMS	G1	Parallel	Serial	Shenandoah
Mean (GB)	3.92	87.8	50.5	40.9	N/A
Min (GB)	1.82	7.48	37.5	40.0	N/A
Max (GB)	12.3	91.3	51.5	41.4	FULL*

*Probably whole heap (150 GB)

Figure 5.15: Heap Usage For All Tested Collectors

5.7.2 Pause Times

Pause times are very important when it comes to real-time applications but they do not matter with batch applications. With batch application parallel collectors are the best choice as they offer the biggest GC throughput (memory freed per unit of time) but with real-time applications one needs the lowest possible pause time as well as to make sure that the pause time will never exceed certain limit. And not do only real-time applications require this - oftentimes service providers have to sign a service level agreement (SLA) which obliges them to keep up to limitation such as maximum application response time.This, in case of Java, means managing garbage collection. Many collectors may have enourmous throughput and sufficient performance (CMS, parallel) but will fail to meet requirements defined by SLA as there might be occassional performance drops due

to full GC invocations. That is where G1 and Shenandoah become developers' best choice.

In accordance with Figure 5.16, CMS collector had the absolutely worst result, because it only used very low heap for nursery and therefore many collections were invoked. Also the minimum timeper collection was long considering the space freed in that time. Right after CMS was the serial collector; it had very decent result considering its single threaded nature. Although there are serious differences in minimum (86.7 ms) and maximum time (1 035 ms) for one collection making serial collector useless whenever pause time matters. What might seem surprising is that parallel collector ended with lower total pause time than G1. This has several reasons. Firstly with G1 one can set a target pause time, which has not been done in this case (default was used). Secondly G1 was colleting a heap of 90 GB in each invocation while parallel collector only collected 50 GB. With different settings G1 should have a better result than parallel collector. Looking from a different angle, parallel collector has, once again, significantly bigger difference in minimum and maximum time of collection whilst G1 collector can keep the longest invocation below 72 ms. With Shenandoah, Figure 5.16 contains only partial information. Total time is only from collections which were not full GC invocations, in those Shenandoah spent mere two seconds. In these incovations, majority of time was spent in concurrent phases making Shenandoah really almost pauseless. However due to the above described bug,log was populated with many full GC calls as well, tainting the actual results. It is unclear how much time would be spent in STW phases if it was not for full GCs, but judging from the beginning of the log where there were none, it would still have the lowest number.

	CMS	G1	Parallel	Serial	Shenandoah
Total	1447932	12468	9415	105140	2000
Mean	160	35.9	14.6	177	N/A
Min	138	12.9	7.72	86.7	N/A
Max	283	71.2	261	1035	36.93

Figure 5.16: Pause Times For All Tested Collectors

5.7.3 SPECjbb Throughput

Last comparison is done in SPECjbb throughpout, measured in bops. Parallel collector has the best results in this case, tightly followed by G1. Serial collector is third with a decent result over 630 000 bops. However there is a noticeable difference between its minimal and maximal bops whilst in case of G1 and parallel collector, these numbers barely diverge. CMS collector took fourth place with steady output of approximately 446 000 bops, this result could have been way better if the nursery size was higher. Last place belongs to Shenandoah with mere 117 063 bops in total and enormous differences in minimal and maximal bops. Maximum was reached right at the beginning where there were only few full GCs and then the performance quickly dropped. Another reason why it is so low is that Shenandoah makes use of multiple threads for GC which run at the same time as the application itself. With SPECjbb there were 20 to 40 threads running in the test plus another requested by GC. If the tests were ran with less warehouses (e.g. with less threads), Shenandoah might have had better result too.

	CMS	G1	Parallel	Serial	Shenandoah
Total	446038	817718	862694	636289	117063
Min	459218	827277	886120	593698	92622
Max	476011	872600	924049	751799	227846

Figure 5.17: SPECjbb bops For All Tested Collectors

6 Conclusion

The first chapter of this thesis elaborates on indirect memory management methods in Java. Weak, strong and phantom references along with finalization are all discussed with focus on how they influence garbage collection. The subsequent chapter moves on to garbage collection itself. Various types of collectors such as copying, tracing, adaptive or incremental are described. Many of them are accompanied by schemes allowing for easy understanding. Following section is dedicated to current garbage collector implementations; four collectors are explained in this section: serial, parallel, CMS and G1. Every collector has a note about usual use-case as well as clarification of its mechanics. The connection to previously mentioned garbage collector basics is also mentioned so that one can easily see how each collector evolved and how does it work. At this point, the thesis gets to Shenandoah, a very low pause garbage collector developed by RedHat. Principles as well as goals are explained since one has since on has to bear in mind that Shenandoah is a very specific collector not suitable for all conditions. Included is an information on Shenandoah's integration with OpenJDK in the future.

Second, and more practical part of the thesis is performance testing of various garbage collectors. Since server applications were targeted, SPECjbb was chosen as a benchmark; it is considered a standard for performance benchmarking in Java. First few sections in the performance testing chapter clarify test environment; SPECjbb is explained more in-depth, followed tightly by hardware specification of the machine which was used. Furthermore there is a section regarding command line options with which the test was executed. Last but not least, monitoring tools are introduced. As for the collectors, all currently used collectors mentioned in the first part of the thesis were put to this test and obviously so was Shenandoah. Test results are presented by tables and graphs with comments and they are sorted per collector. For each collector, three main aspects were observed - pause time, heap usage and SPECjbb throughput. Last section is a conclusion of the test results, which compares all the collectors together in each aspect, one at a time.

While executing the tests, several problems with Shenandoah

were discovered and developers were contacted to clarify several issues. One has to bear in mind that Shenandoah is under development and is by no means ready to be used in real environments. It still has quite a lot of problems and bugs. Firstly, log files are very simple and incomplete, plus there is no tool which could parse the results. Secondly, one of the bugs was a very long full GC collection invoked due to allocation failure. This effectively put Shenandoah to the very last place when measuring SPECjbb throughput, even behind a serial collector which works with but one thread. Nevertheless, regarding pause time in non-full collections, Shenandoah had top results, succeeding in making majority of work concurrently. As for heap usage there is no reliable way of telling how much of the heap was used; however, SPECjbb and allocation failures indicated that Shenandoah was using much more heap than any other collector. In fact some of the collectors failed to make use of the enormous heap space and kept using minimal amounts leading to extensive numbers of collections.

Shenandoah is first of its kind - first garbage collector targeting low to no pause times when running on huge heaps (100 GB and more) which will be free to use and probably even integrated into OpenJDK. There are other existing solutions to low pause collectors such as Azul, but none of them can be used for free and with OpenJDK. Shenandoah seems like a very promising project and it is indeed a great leap forward. But with every such innovation comes a lot of troubles; there are many things which are yet to be polished. Firstly, major bugs, such as those described in this thesis, have to be addressed. Also adding some detailed information about memory used/freed would provide more reliable information about collector mechanics and usability. Last but not least, when Shenandoah gets more stable, it should also implement APIs so that tools such as JConsole could read its output. Also being able to parse the log with analytic tool would be convenient as it is likely to attract more early testers which will in turn discover more weak points and help to further improve Shenandoah.

Bibliography

[1] Venners, Bill. *Inside the Java 2 Virtual Machine*. 2nd edition. McGraw-Hill Companies, 2000.

[2] *Oracle. Java™ SE 7 Update 4 Release Notes.* [online]. [cit. 2015-25-1]. URL: *<http://www.oracle.com/technetwork/java/javase/7u4-relnotes-1575007.html>* .

[3] *Oracle. Java Garbage Collection Basics.* [online]. [cit. 2015-11-2]. URL: *<http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>* .

[4] *Oracle Blogs. The Unspoken - Phases of CMS.* [online]. [cit. 2015-12.2]. URL: *<https://blogs.oracle.com/jonthecollector/entrys/hey_joe_phases_of_cm>* .

[5] *InfoQ. G1: One Garbage Collector To Rule Them All.* [online]. [cit. 2015-12.2]. URL: *<http://www.infoq.com/articles/G1-One-Garbage-Collector-To-Rule-Them-All>* .

[6] *InfoQ. Deep Dive into G1 Garbage Collector.* [online]. [cit. 2015-12.2]. URL: *<http://www.infoq.com/presentations/java-g1>* .

[7] *Oracle. Getting Started with the G1 Garbage Collector.* [online]. [cit. 2015-12.2]. URL: *<http://www.oracle.com/technetwork/tutorials/tutorials-1876574.html>* .

[8] *Roman Kennke's Blog. Shenandoah: A pauseless GC for OpenJDK.* [online]. [cit. 2015-27.2]. URL: *<https://rkennke.wordpress.com/2013/06/10/shenandoah-a-pauseless-gc-for-openjdk/>* .

[9] *ChristineFlood. Kick Butt, Eat Lunch, and Take out the Garbage.* [online]. [cit. 2015-27.2]. URL: *<https://christineflood.wordpress.com/2013/06/19/kick-butt-eat-lunch-and-take-out-the-garbage/>* .

[10] *Shenandoah. An ultra-low pause time Garbage Collector for OpenJDK.* [online]. [cit. 2015-27.2].

URL: <*https://rkennke.files.wordpress.com/2014/02/shenandoahtake4.pdf*> . (slides from Fosdem 2014)

[11] *YouTube*. *"Shenandoah: An open source pauseless GC for OpenJDK." by Christine Flood.* [online]. [cit. 2015-27.2]. URL: <*https://www.youtube.com/watch?v=QcwyKLlmXeY*> .

[12] *SPECjbb2005 User's Guide*. *Background.* [online]. [cit. 2015-25.3]. URL: <*https://www.spec.org/jbb2005/docs/UserGuide.html*> .

[13] *IBM developerWorks*. *IBM Monitoring and Diagnostic Tools - Garbage Collection and Memory Visualizer.* [online]. [cit. 2015-2.4]. URL: <*http://www.ibm.com/developerworks/java/jdk/tools/gcmv/*> .

[14] *IcedTea*. *March 2015 Archives by thread.* [online]. [cit. 2015-2.4]. URL: <*http://icedtea.classpath.org/pipermail/shenandoah/2015-March/thread.html*> .

Electronic Attachements

As a part of this thesis there are following electronic attachements:

- SPECJBB results for all tested collectors. Please note that HW specifications listed in these results do not correspond those used for testing. They are hardcoded in property file when test starts executing.
- Log files for all tested collectors. Some files had their headers modified in order be parseable via GCMV.