DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING, SECOND CYCLE, 30 CREDITS

STOCKHOLM, SWEDEN 2020

Performance evaluation of Java garbage collectors for large heap transaction based applications

NICOLE JAGELID

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Performance evaluation of Java garbage collectors for large heap transaction based applications

NICOLE JAGELID

Master in Computer Science
Date: July 1, 2020
Supervisor: Hamid Reza Faragardi
Examiner: Mads Dam
School of Electrical Engineering and Computer Science
Host company: Scila AB
Swedish title: Utvärdering av prestanda hos Java garbage collectors i transaktionsbaserade applikationer med stor heap

Abstract

The automated memory management mechanism known as garbage collection is a vital part of the Java Virtual Machine (JVM) and has considerable influence on the overall performance of a running application.

Java Development Kit 13 (JDK13) offers several garbage collectors for the developer to choose from when configuring the environment. Among those Garbage First GC (G1GC), Shenandoah GC, and the Z Garbage Collector (ZGC). In this study, we investigate the performance of the mentioned three collectors placed in the same environment. The collectors are tested on a subset of the DaCapo Benchmark Suite, and two real-world transaction-based applications with heap sizes of 63GB and 256GB respectively. The collectors are evaluated on multiple metrics with various GC configuration setups (additional GC- and JVM options) for each collector. Latency and throughput are key performance values in this study.

The findings from the study demonstrate that one garbage collector configuration does not fit all applications. Shenandoah GC and ZGC perform significantly better than G1GC in terms of latency, average pause time, maximum pause time, and total accumulated stop-the-world time, for all evaluated applications in the study when the right tunings are applied. It is possible to achieve more than 80% improvement on latency values. However, in regards to throughput and execution time, G1GC performs similarly to Shenandoah GC and ZGC. G1GC is also a very stable collector in terms of heap usage and is not as sensitive as Shenandoah GC and ZGC towards heap over-provisioning.

This study also shows the possible performance impact of GC tuning on application throughput and latency.

Sammanfattning

Den automatiska dynamiska minneshanteringsmetoden känd som ’garbage collection’ eller skräpsamlare, är en avgörande del av Java(s) Virtual Machine (JVM) och har betydande inflytande över prestationen hos en aktiv applikation.

Java Development Kit 13 (JDK13) erbjuder ett flertal skräpsamlare för systemutvecklare att välja bland. Bland dessa finns Garbage First GC (G1GC), Shenandoah GC och Z Garbage Collector (ZGC). I denna studie undersöks prestandan hos nämnda skräpsamlare i samma miljö. Skräpsamlarna testas på en delmängd av DaCapo Benchmark Suite, och två transaktionsbaserade produktionsapplikationer med stor tillgång till dynamiskt minne (heap). 63GB respektive 256GB. Skräpsamlarna evalueras på flera mätvärden med diverse konfigurationer av skräpsamlaren (extra parameterjusteringar). Reaktivitet (latency) och genomströmning (throughput) är nyckelvärden i denna jämförelsestudie.

Resultaten av denna studie visar att en och samma skräpsamlarkonfiguration inte passar alla applikationer. Shenandoah GC och ZGC presterar märkbart bättre reaktivitet jämfört med G1GC, för alla utvärderade applikationer i undersökningen när rätt modifikationsparametrar appliceras. En möjlig förbättring på 80% av reaktivitetsvärden. Däremot, med hänsyn till genomströmning och exekveringstid, presterar G1GC liknande Shenandoah GC och ZGC. G1GC är också en stabil skräpsamlare i termer av heap användning och är inte lika känslig för överprovisionering av heap, som Shenandoah GC och ZGC är.

Denna studie visar också på möjliga prestandafördelar av justeringar av skräpsamlar-parametrar.

Contents

1 Introduction

In the Java programming language the automated memory management process known as garbage collection is a critical component of the Java Runtime Environment (JRE) and Java Virtual Machine (JVM). The Garbage Collector (GC) mechanism has considerable influence over the overall performance and efficiency of a running application [1] [2] [3]. Today there exists a great need for continuous and uninterrupted services in modern software applications. It is therefore valuable to analyze how different garbage collectors are performing when managing memory and how they impact the application execution.

With the release of Java 13, multiple garbage collectors are offered by the JVM. Serial GC, Parallel GC, Concurrent Mark Sweep GC, Epsilon GC, Garbage First GC (G1GC), Shenandoah GC, and Z Garbage Collector (ZGC). G1GC is the default collector of Java 13 while ZGC and Shenandoah GC are 2 out of the 3 latest released GCs for Java (along with Epsilon GC) and have been highly praised for their performance benefits.

Typically when looking at the performance of a garbage collector the key measurement values are latency and throughput. Latency is the responsiveness of an application, the delay experienced by the application due to garbage collection activity. For certain collection phases, the garbage collector needs to suspend all application threads to conduct necessary memory management. This suspension duration, known as a pause time or Stop-The-World (STW) event, results in latency as the application is unable to progress. The goal when optimizing for low latency is to reduce the duration of pause times and/or the number of times they occur [4]. Throughput is instead a measurement of the workload managed by an application within a specific unit of time. It is a measurement of application production efficiency. For example, throughput can be measured by the number of completed transactions in an hour or percentage of the total execution time of an application not spent on garbage collection activities, measured over long periods of time. When opting to maximize throughput the workload managed per time unit, in the long run, is prioritized and higher pause times can often be overlooked. However, in large scale real-

world systems a delicate balance of throughput and latency is required [4].

Garbage collectors traditionally try to make assumptions and/or observations of application behavior to form the memory management after memory behavior. For example, the Garbage First GC concentrate space-reclamation efforts on young objects based on the assumption that objects more recently allocated are more likely to have turned into garbage. Another example is the Adaptive heuristic of Shenandoah GC which initiates a garbage collection cycle based on observations of time, heap occupancy, and allocation pressure from previous garbage collection cycles. Thereby relying on the assumption of repetition of application memory behavior. The more the GC can work in symbiosis with the application the lesser the overhead caused by GC activity will be. However, because applications vary significantly in memory behavior based on their implementations, goals, and resources one single GC configuration does not fit all. Each collector is therefore equipped with a set of optional parameters. These parameters may be utilized by the developer to tune the garbage collector to the needs of the running application in order to favor, for example, throughput or latency. In addition to the GC parameters, the JVM also offer adjustable optional parameters for modification.

Java 13 was very recently released and the performance of the mentioned garbage collectors has not been extensively evaluated or compared. Furthermore, sometimes performance evaluations in standardized, limited, and controlled scientific environments are unable to reflect actual performance in real-world applications. For these reasons it is of interest to evaluate the performance of Java garbage collectors further, and in an environment where efficiency is critical and requirements are true to production reality.

The Processing Node Server application and the Query Node Server application are two production applications part of a distributed transaction-based system. The system handles billions of transactions per day resulting int specific throughput and latency requirements. The processing node translates data into a normalized format and sequences it into a time-series. Given the normalized stream, the processing node analyzes the data for specific given patterns in near real-time. Additionally, the processing persists and index the data for later use by the query node. Due to the near real-time evaluation, the processing node opts for high throughput and requires latency values to be no higher than 1s. The query node handles historical queries, searches, report generations, and evaluation algorithms (on historical data). In view of the fact that the query node focuses on historical data, throughput is more important than latency, although latency values should be kept below 10-20s. Given the heavy lifting of the processing node and query node, they are allowed large

heap memory sizes - 63GB and 256G respectively. The large heap memory sizes, latency requirements, and throughput aim make these applications of interest when evaluating garbage collection performance and this study will utilize them for exactly that purpose.

This thesis attempts to shed light on the performance of Garbage First GC, Shenandoah GC, and ZGC against each other in real-world large heap transaction-based environments, taking a set of tunable parameters into account.

1.1 Aim and research questions

This thesis aims to provide insight and knowledge into the performance of G1GC, ZGC, and Shenandoah GC when compared in the same environment. The environment refers to the hardware, the applications, and the application configurations unrelated to the GC unless otherwise specified. With applied tuning, the objective is also to improve the application throughput and/or latency of the Processing node and Query node (limited by GC).

The Processing Node Server application with heap restrictions of 63GB and a latency requirement of 1 second corresponds to *Benchmark I*. The Query Node Server application, with heap restrictions of 256GB and a latency requirement of 10-20 seconds correspond to *Benchmark II*.

Specifically, this thesis set out to answer the following research questions concerning Garbage First GC, Shenandoah GC, and Z GC:

1. **RQ1:** *Which garbage collector best suites the applications benchmark I and II respectively, when taking specified collector settings into account and opting for high throughput?*

2. **RQ2:** *Which garbage collector best fits applications with high heap utilization variation, running on large heaps, when taking specified collector settings into account and optimizing for low latency?*

3. **RQ3:** *Which garbage collector best fits applications processing data in near real-time (live) when taking specified collector settings into account and optimizing for low latency?*

1.2 Research methodology

The research methodology of this thesis follows an adaptation of the research methodology steps presented by Nicklas Walliman [5]. Figure 1.1 visualize the steps of this thesis' work. The first step focuses on formulating the research objectives. Next, related works are identified and studied. The research questions are defined. Suitable methods of evaluation and tools for measurement are gathered. Subsequently, an in-dept literature study of the garbage collectors is performed. The literature study is followed by the implementation of the test environment and configuration setups for data collection. Data is then generated and collected followed by analysis in an iterative process tuning the implementation. The final results are then thoroughly analyzed and discussed leading up to conclusions.

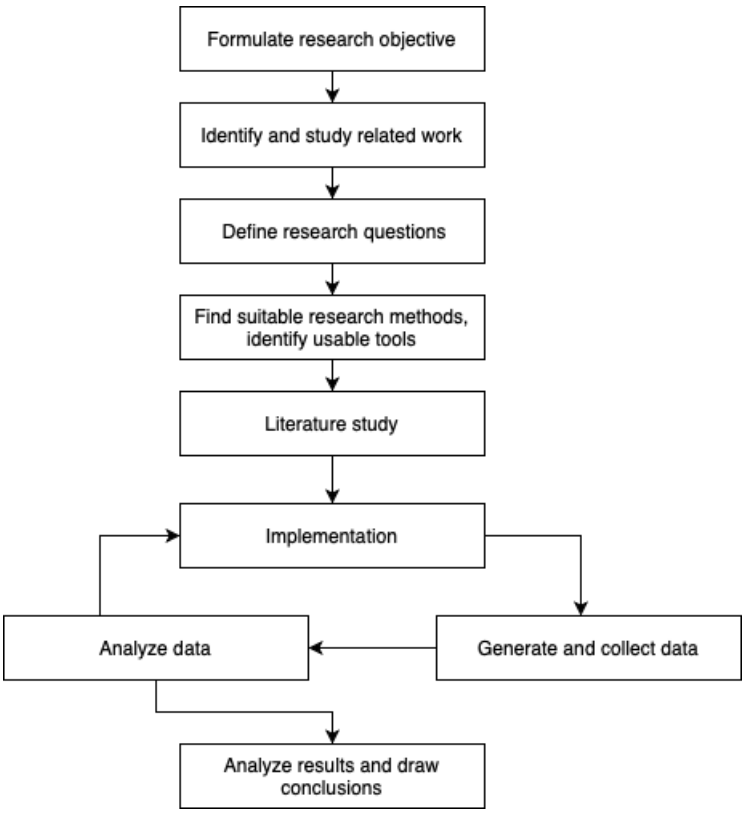


Figure 1.1: Research methodology.

The first steps consist of reading, identifying related work, formulating goals, defining research questions and finding a suitable research method with fitting tools. The second part of the research is to implement the presented

method(s) and by such generate data. The third, and last part, is then to conduct extensive analysis on this data in order to draw conclusions from the results and answer the research questions.

1.3 Scope

The JVM offers a tremendous amount of options to tweak its behaviour, in addition to these, each garbage collector comes with their own modifiable parameters. Therefore evaluating all available tuning options and exhausting the possible combinations lie outside the scope of this thesis. This performance evaluation instead choose to include only a few sets of configurations in addition to the out-of-the-box setups. These configurations will form from reading up on their behaviour and potential performance effect.

1.4 Contribution

Garbage collectors have been researched to a large extent for a long time. The Serial GC, Parallel GC, Concurrent Mark Sweep GC, and Garbage First GC have all been under scrutiny in performance comparison research [6] [7] [1] [8]. However, due to the fact that Shenandoah GC and ZGC were so recently released and are still experimental, there has not been a lot of research involving them specifically other than the presentations given by their respective providers, Red Hat and Oracle. Furthermore there has not been any practical published research comparing the performance of all three collectors in the same environment. This thesis will thus offer more insight regarding the new garbage collectors and how they perform compared to each other on smaller classical performance benchmarks and in a larger scale production environment with both latency and throughput requirements. It will also be valuable to view the performance on the specific hardware used.

Another aspect of the contribution of this thesis is that it will investigate the application performance impact of a set of optional GC parameters in addition to the default configurations. A topic in the investigation area of Gousios et. al, who examined the effects of tuning garbage collectors in an application server environment [9].

1.5 Ethical sustainability and societal aspects

The ethical aspects of this study mainly relates to the implementation and conduction of the experiments. Because this study evaluates the performance of three different Java garbage collectors, it is vital to ensure that the environment in which the collectors are evaluated is equal for all. It is also important for the study to remain objective and to honestly present any and all possible injustices between the collectors caused by evaluation method. Moreover, since the results of this study may lie as argument for choice of garbage collector in future environments (private, research or commercial) by developers, the results must be dependable. This is ensured by openness of method choices and by providing test results on a subset of the open source DaCapo benchmarks (reproducible). The environment setup and method, including configurations, are thoroughly described and all tools used for evaluation are presented.

Increased garbage collection performance can be beneficial in several aspects. Depending on the performance feature reviewed, a better GC configuration can lead to application efficiency, reduced memory footprint, reduced CPU consumption, and/or further along reduced hardware costs. The more efficiently we utilize the resources we already have, the more we can achieve. This study intends to provide insight into three collectors in order for developers to be able to make better informed decisions when choosing GC for their environment.

1.6 Outline

The report is divided into seven chapters. Chapter 2 aim to provide essential information on the subject of garbage collection to understand the analysis and discussion of the results. Chapter 3 presents previous work related to garbage collection and performance evaluation. Chapter 4 describes the method used for evaluation and metrics definitions are described. Chapter 5 presents the results produced by the experiments conducted. Chapter 6 discusses the findings in chapter 5 as well as limitations of the study. The report then concludes with chapter 7 which summarize the thesis findings, provides answers to the research questions and discusses potential future work.

2 Background

2.1 Hotspot JVM

The Java Virtual Machine (JVM) is an abstract computing machine that translates the Java programming instructions into instructions and commands readable by the operating system.

Hotspot JVM is an implementation of the JVM concept and lays at the core of the Java SE (Standard Edition) platform. The architecture of the Hotspot JVM lays on a strong foundation that supports massive scalability as well as the ability to achieve high performance on even the largest available computer systems [10].

From the perspective of tuning performance, there are three major components of the Hotspot JVM concerned; the heap, the garbage collector, and the Just-In-Time compiler (JIT). In newer versions of the JVM, the JIT rarely needs tuning and focus is instead concentrated on the heap and the garbage collector. The heap is the memory space in which object data is stored, and the garbage collector is the component responsible for managing this area.

2.2 Introduction to garbage collection

The idea of a garbage collector in the JVM is a mechanism that automatically manages the applications' dynamic memory allocation requests to free the application developer from this delicate task. By relieving the developer of having to match memory allocation with deallocation and keeping track of object lifetimes, some classes of error can be eliminated. The cost of this, however, is runtime overhead.

A garbage collector is responsible for the heap and achieves management of dynamic memory allocation through several operations; allocate memory from the operating system as well as return memory back to it, deliver memory to an application upon request, determine which memory is still in use by an

application and reclaim unused memory.

GC roots are origin points which the application can use to reach the rest of the objects on the heap. An object is considered garbage if it is unreachable by a walk of the object graph starting from the GC roots. If the object is unreachable from roots no application can access is. An object that is accessible from the roots is referred to as live.

2.3 Garbage First GC

The Garbage First (G1GC) garbage collector is a generational, incremental, parallel, and mostly concurrent collector. It is designed to reduce pause times by monitoring pause time goals in each stop-the-world collection and utilizing the statistics to predict future collection execution time and space. G1GC prioritizes regions with low occupancy of live objects, i.e choosing garbage first. Space reclamation efforts focus on the young objects where it has been proved most efficient to do so, with infrequent space reclamation of older regions [11].

G1GC is not a real-time collector. It aims to achieve set pause time goals with high probability over longer periods.

The goal of G1GC in the default configuration is not to maximize throughput nor to have the lowest latency but instead to produce uniform, small pause times.

2.3.1 Heap layout

The G1 heap is partitioned into a number of regions of equal size. Each region is a continuous range of virtual memory, considered a unit of memory allocation and reclamation. The units may be empty (free), assigned to the young generation or assigned to the old generation. See figure 2.1. Units of memory assigned to the young or old generation are fully, or in part, allocated to one or more applications. The young generations contain eden regions (newly allocated regions) and survivor regions (regions still live after collection), typically laid out in a non-contiguous way in memory. The old generation regions may be humongous, i.e contain large objects that span several regions. The free regions are handed out upon incoming requests and simultaneously assigned to a generation. All regions are assigned to eden in the young generation upon allocation except humongous objects which are directly assigned to the old generation [12].

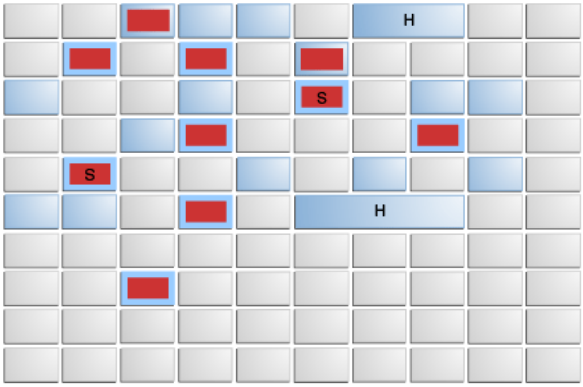


Figure 2.1: Garbage First GC heap layout. Red - young generation eden regions. Red with "S" young generation survivor regions. Blue - old generation regions. Blue with "H" - old generation humongous regions[13].

2.3.2 Collection cycle

The G1GC garbage collection cycle consists of two phases; young-only phase and space-reclamation phase. The young-only phase begins with Normal young collections responsible for promoting objects from the young generation into the old. When the old generation occupancy reaches the Initiating Heap Occupancy threshold (a pre-set occupancy threshold), the transition into the space-reclamation phase begins. At this point, a Concurrent Start young collection is scheduled. The Concurrent Start collection performs a Normal young collection in addition to initiating the concurrent marking process. The marking process tag all live objects within the old generation to be kept in the following space-reclamation phase. At the end of the marking process, two STW pauses occur; Remark and Cleanup. Remark performs global reference processing and class unloading, reclaims regions which are completely empty and cleans up internal data structures. During the Remark pause, the GC chooses which regions to reclaim space from in the next collection. This set of chosen regions is referred to as the collection set. The set contains young regions whose sizes add up to a pre-defined target value. It is in part by the number of regions selected into the collection set that G1GC exerts control over the pause time lengths. In a concurrent cycle between Remark and Cleanup, G1GC prepare the collection set candidate regions chosen for later collection[12].

The Cleanup pause sorts the results of the preparation according to their efficiency. In subsequent mixed collections G1 can then chose more efficient regions that are quick to collect and contain more free space, by order. The collection set of a mixed collection contains both young and old generation

regions.

During the cleanup phase, it is also ruled whether a space-reclamation phase should follow or not. If no space-reclamation is required the young-only phase finishes here with a single Prepare Mixed young collection.

The space-reclamation phase is performed incrementally in steps and in parallel. It involves multiple mixed collections that evacuate young generation regions and live objects of sets of old generation regions. Evacuation is the process of copying live data from one memory area to be collected, into a surviving memory area. Thus active memory is evacuated from memory marked for collection, while additionally compacting living areas in the process. After evacuation completion, the space may be reused for new requests of memory allocation. The destination of the memory copy is determined by the source area (e.g eden, survivor, old generation). The space-reclamation phase proceeds until evacuating more old generation regions won't yield enough free space to motivate the effort. The G1GC determines this weigh-off by using a prediction model that targets the predefined pause time goal, along with the prepared collection set from the Remark pause in the young-only phase. The predictability model is based on G1GC tracking information about previous application behaviour and pause times with their associated costs.

When the space-reclamation phase completes the collection cycle restarts with another young-only phase. If the application were to run out of memory during liveness information gathering, the G1GC performs a stop-the-world full heap compaction [12].

With JDK12 (Java Development Kit, release 12) G1GC was improved by giving G1GC increased control over the pause times. The ability to abort pause times was added. G1GC maintains records of how accurately its heuristics are predicting the number of regions to include in the collection set and proceeds with abortable collections only if needed. G1GC partitions the collection set into two groups: optional regions and mandatory regions. Mandatory regions are always processed and collected while optional regions are collected if enough time remains to stay within the target pause time length [14].

2.4 Shenandoah GC

Shenandoah GC is the ultra-low pause time garbage collector for OpenJDK, developed by Red Hat. Shenandoah does the major GC work concurrently, including concurrent evacuation, which distances the relationship between heap sizes and GC pause times tremendously. Shenandoah aims to produce the

same low pause times regardless of the heap size, meaning 200GB heap and 2GB heap should perform very similarly [15].

Much like Garbage First GC, Shenandoah was not developed to suit all environments. Shenandoah puts focus on responsiveness by delivering consistent short pause times. This means the collector algorithms do not prioritize throughput, memory footprint or other over latency, even though it still aims to perform well in these areas.

2.4.1 Heap layout

The Shenandoah GC is a regionalized collector and maintains the heap as a collection of regions. The regions marking may be 'free', 'newly allocated', 'live data', 'live data in collection set', or 'live data in collection set updating references to'. See figure 2.2. Upon allocation, involved free regions are marked newly allocated. Concurrent marking marks live data. Regions selected for concurrent evacuation are marked live data in collection set. Evacuated regions to update references to are marked live data in collection set updating refs to [15].

Unlike G1GC, Shenandoah is not a generational collector. This means that the heap is not divided into generation sections and thus all regions must be marked every garbage collection cycle.

Shenandoah requires additional word per object for forwarding pointers (Brooks pointers) at all times which entails some smaller memory footprint overhead [16].

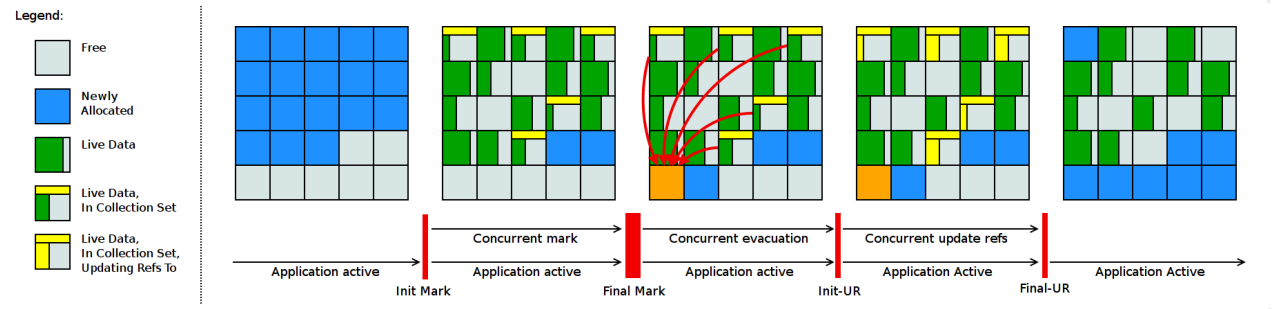


Figure 2.2: Shenandoah GC heap layout [15].

2.4.2 Collection cycle

The regular garbage collection cycle for Shenandoah consists of four major phases, starting with the Initial Marking. Initial Marking is the first STW pause in the collection cycle. It starts off the concurrent marking by preparing the heap and application threads and scanning the roots set. The length of the pause connects to the size of the roots set instead of the heap. Concurrent Marking then traverses the heap to find live objects (reachable objects) and mark those live data. This is done alongside the application threads. Final Marking completes the marking period by draining the pending marking/update queues, re-running a scan of the roots set, and freeing fully evacuated regions from previous compaction phases. This constitutes the second pause time. Again, the size of the roots set along with the number of pending updates determines the pause length. The final marking phase also initiates the concurrent evacuation by preparing the regions to be collected (collection set), pre-evacuating some roots, and prepare runtime for the next step. Concurrent Cleanup then evacuates completely empty regions (no live data). Last is the Concurrent Compaction phase. Live objects from regions in the collection set are concurrently evacuated. This is contrary to the G1GC where evacuation is done during an STW pause. The length of the evacuation is determined by the size of the collection set. In order for the evacuation phase to work concurrently, Shenandoah uses Brooks pointers. Each object on the heap includes an additional reference field (Brooks forwarding pointer). The reference either points to the object itself or to a new location to which the object has been copied to. To ensure consistency, barriers are set up in various places for all that writes or uses the objects. The barriers read the Brooks pointer which forwards the reference to the new location of the object. This enables evacuation concurrent with mutator threads, i.e application threads [15] [17].

As evacuation completes Shenandoah triggers a short pause to confirm that all GC and application threads have finished evacuation. Thereafter references are concurrently updated in the next phase by Shenandoah traversing the heap and updating the references to objects that were copied/reallocated during the evacuation. I.e updating the Brooks pointers. A final update of references re-updates the root set and recycles the regions of the collection set in an STW pause. Concurrent cleanup finalizes the cycle by reclaiming the regions in the collection set, who are now without reference [15][14] [17].

2.5 Z Garbage Collector

The Z Garbage Collector (ZGC) is a low latency GC designed to achieve pause times independent of the heap or live-set size, that never exceed 10 milliseconds, while not overstepping a 15% maximum throughput reduction. The ZGC is concurrent at its core and is suitable for applications with large heaps demanding high responsiveness. The target goals for ZGC are very similar to those of Shenandoah GC, however, their respective implementations and approaches are very different. The ZGC uses multiple-memory mappings and colored pointers to enable more concurrent garbage collection work, contrary to Shenandoah's Brooks pointers.

ZGC is still considered an experimental garbage collector and will support macOS and Windows with the next JDK release (JDK14).

2.5.1 Heap layout

ZGC is a single generation (i.e not generational like G1GC), region-based collector. The regions, also called ZPages, can be dynamically allocated and dynamically sized into one of three group sizes; small (2MB), medium (32MB), large (N X 2MB).

Objects below 256k are assigned to Small Page, objects below 4M are in Medium Page, and above are in Large Page.

The physical heap regions of the ZGC can, unlike other GCs, map into bigger heap address spaces (may include virtual memory). This resolves issues of allocating really large objects and otherwise needing to perform multiple collections in order to free enough memory.

2.5.2 Collection cycle

One of the core concepts of the ZGC implementation is the colored pointers. ZGC stores metadata in unused bits in the 64-bit object pointers (references). Four bits are taken into action and given the purpose to mark the state of an object: finalizable, remap, mark0 and mark1. One problem that arises with color pointers is that it introduces additional work when dereferencing the pointer because you need to mask out the information bits. The Z garbage collector neatly solves this by using multi-mapping. Multi-mapping is a concept of mapping multiple different ranges of virtual memory to the same physical memory. By design only one of remark, mark0 and mark1 may be 1 at any given time. Therefore 3 mappings should be able to resolve the issue[3]. Each

of the state bits (remap, mark0, mark1) maps to a different range in the virtual address space:

- 0001 = Marked0 (Address view 4-8TB)
- 0010 = Marked1 (Address view 8-12TB)
- 0100 = Remapped (Address view 16-20TB)
- 1000 = Finalizable (Address view N/A)

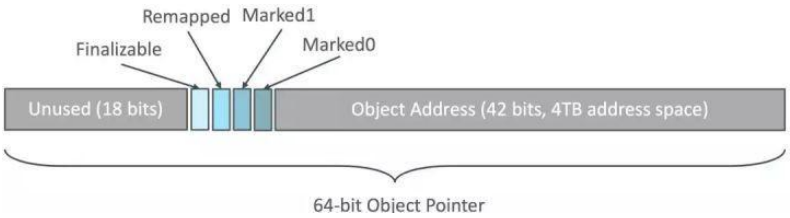


Figure 2.3: Colored pointer ZGC [3]

Pause Mark Start commerce the ZGC collection cycle by marking the GC roots as live. The Concurrent Mark/Remap then traverses the object graph and mark (color-marked bits) all accessible objects as live, working concurrently with the application threads. In this phase, a load barrier is evaluating all loaded references (colored pointers) against a mask in order to establish if they have been marked or not yet. Load barriers are a portion of code that runs each time an application thread loads a reference from the heap. Unmarked references are added to a queue for marking. Pause Mark End is then a brief STW synchronize point allowing for weak roots cleaning and other edge cases. The Pause Mark End finalizes the marking phase.

The next procedure of the collection cycle is relocation, consisting of three concurrent sub-phases; Concurrent Prepare for Relocation, Concurrent Reference processing, and Concurrent Class Unloading. This part processes the references, performs weak root cleaning, and conducts relocation set selection. The relocation set selection is the ZGC dividing the heap up in pages and selecting a set of these pages whose live data needs to be relocated in order to free the regions.

Pause Relocate Start then initiates the reallocation by first handling the objects referenced to as roots (local variables etc.) and remaps their references to the new location. The Concurrent Relocation then performs the relocation of all objects within the collection set by walking the relocation set. Load barriers detect loads of object references pointing into the relocation set. If an

application thread tries to load an object in the relocation set before the GC has been able to relocate them, the application thread may perform the relocation of the object. The relocation forwarding tables are kept off-heap, thus no forwarding information is stored in old object copies. This enables immediate reuse of the heap region evacuated. Once all live objects been relocated the region is immediately ready for usage.

At the end of the relocation process there may still exist references pointing to old locations of objects. To remap these references efficiently, ZGC combines this task with the next marking phase. During the graph traversal, non-remapped references are remapped and marked as live.

Nearly all collection pauses in the ZGC are dependent on the number of GC roots rather than the live set, heap size, or garbage amount [3][18].

ZGC also performs class unloading concurrently (from release JDK12 and later). Class unloading has traditionally been performed during an STW pause. This positively impacts the efficiency of the JITs and runtime subsystems [14].

2.6 DaCapo benchmark suite

The DaCapo benchmark suite is a set of open-source real-world applications with complex logic and non-trivial memory loads [19]. DaCapo focuses on large, realistic, general-purpose programs, and include both client and server-side applications. The benchmarks are suitable for measuring memory management performance and have been used in several other performance studies of garbage collectors [1] [6] [17]. For this thesis, a subset of the DaCapo 9.12 benchmark suite (released in 2009) will be used, namely avrora, jython, h2, lusearch-fix and tradebeans. A short description, provided by DaCapo, of each application:

- avrora - simulates a number of programs run on a grid of AVR microcontrollers.
- jython - a python interpreter written in Java. The test executes (interprets) the pybench Python benchmark. Version 2.5.1.
- h2 - An SQL relational database engine written in Java. The program in DaCapo 9.12 executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application. Version 10.5.3.0 and 1.2.121.

- lusearch-fix - Uses lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible. Version 2.4.1.
- tradebeans - runs the Daytrader benchmark from Apache via a Jave Beans to a Geronimo backend with an in-memory h2 as the underlying database. Version 2.1.4.

2.7 Real-world application

In this project two real-world applications will be the main focus when evaluating the garbage collectors; the Processing node server and the Query node server. Both application types are part of a distributed transaction-based system handling billions of transactions per day, demanding high throughput and low latency. These two application nodes are the only two in the system that could be run properly in isolation.

2.7.1 The Processing Node

A Processing Node will have as primary input public and private market data from security exchanges. The data is translated to a normalized format and sequenced into a monotonically increasing time-series. Given this normalized stream of transactions, the processing node's primary purpose is to analyze the stream of data for any irregularities or criminal behavior of market participants in near real-time. Apart from this, it will also persist and index data for later use by the Query Node.

A typical system may consist of several processing nodes depending on how large quantities of market data needs to be captured. The processing node servers do not need to process data with very low latency, but high throughput is important. There is however a maximum allowed pause time close to one second for user experience reasons and to provide near real-time evaluation.

For this thesis, focus will be on a processing node capturing market data and a participant's private market activity. The server is currently configured with a heap size of 63G with a post-GC average utilization of 70%.

2.7.2 The Query Node

The purpose of a Query node server is to handle historical queries, searches, report generation, and evaluation algorithms (all on historical data). The query

node typically runs with very large heap. Since the Query node is not processing live data, it's not sensitive to long garbage collection pauses, however, maximum allowed pause time is around 10-20 seconds.

In a system, there is only one query node responsible for handling queries related to all markets. The query node is configured with a heap size of 256G. Heap utilization has a high variation for this server type depending primarily on how many concurrent invocations there are of historic analysis of trade monitoring rules.

3 Related Work

Considering how recently the Shenandoah Garbage Collector and Z Garbage Collector were released as parts of the JDK, there is a relatively small public body of literature regarding their respective and compared performance. Although, their respective providers Red Hat and Oracle, have conducted and presented smaller performance evaluations. The Garbage First GC (G1GC) has however been an option for a longer period of time and has thus been more largely involved in garbage collection research.

3.1 Analysis and optimization

While the body of literature of the specific garbage collectors may be limited, work in the general domain of garbage collection performance has been extensively researched. The performance and efficiency of the process of garbage collection are important topics and has been ever since the managed runtime environments first got introduced into the world of large-scale, heavy production and research. For example Philipp Lengauer and Hanspeter Mössenböck presented work on automatically tuning Java garbage collectors in a black-box manner considering all available parameters [2]. Jayasena et. al look into the performance of the JVM itself and how it could be auto-tuned with the JVM parameters (which affects the GC performance) [20]. Detlefs et al. specifically look at the Garbage First GC, its inner workings and performance [11]. Haoyu et. al thoroughly analyze the workings of a Full GC [21] and Suo et. al attempt an optimization of the HotSpot Parallel GC on multicore systems [22]. Gousios et. al examine the effects of tuning garbage collectors in an application server environment [9]. They identified several tuning recommendations. A small subset of which is that a small heap size may result in too many full heap collections, and high allocation rates or too frequent full heap collections could mean too much garbage collected in each collection round [9].

A common method in aforementioned scientific performance studies is to conduct the analysis of the garbage collectors by investigating the garbage collector log produced by the JVM (gc log). The gc log provides great insight into the workings of the garbage collector. The log states when a garbage collection ran, how many collections where made, how long it ran, how long the JVM paused when there was a full garbage collection, what was the total allocated memory, how much memory was released and much more. The investigation of the log files are often done with the help of a garbage collector log analytics tool, and with additional manual inspection. Tools used include Visual VM supported by Oracle, with additional Visual GC plugin [7] [6], GC Viewer [23], GCEasy and jClarity Censum. Visual GC (on top of VisualVM) and GC Viewer do not yet support Shenandoah GC or ZGC, but both GCEasy and jClarity Censum does. Therefore this project will use GCEasy and jClarity Censum for analysis.

3.2 Performance comparison

A couple of studies have measured and compared garbage collectors of the JVM, most of which involve a subset of the collectors Serial GC, Parallel GC, Concurrent Mark Sweep GC (CMS) and Garbage First GC (G1)[6] [7] [1] [8]. However the providers of Shenandoah GC and Z Garbage Collector have presented some performance evaluation values on their respective collector.

A comparison study from Java Oracle, presented by Per Lidén and Stefan Karlsson, presents a small performance evaluation between G1GC and ZGC (+ Parallel GC). On the SPECjbb2015 benchmark on a Oracle Linux 7.4, heap size 128G, mode Composite and hardware Intel Xeon E5-2690 2.9GHz 2 sockets, 16 cores (32 hw-threads), their tests show an almost equal score on max-JOP (throughput) between G1GC and ZGC, but for the critical-JOPS (Throughput with latency requirements) the ZGC scores significantly higher. ZGC is roughly comparable to the Parallel GC in throughput delivery but with immensely better latency values. In the same study on the same benchmark the ZGC performed remarkably better than G1GC when measuring pause times [3].

The developers behind the Shenandoah GC, Red Hat, have also made a comparison to the G1 collector on SPECjbb2015. Their tests where run with 200GB heaps on an Intel Brickland box running RHEL 7. Intel Platform: Brickland-EXCpu:Broadwell-EX, QDF:QKT3 B0 Stepping (QS), 2.2Ghz, 24 Core, 60MB shared cache COD ENABLED Intel(R) Xeon(R) CPU E7-8890 v4 @ 2.20GHz 524288 MB memory, 1598 GB disk space. Their findings

show Shenandoah performing slightly worse than G1GC on max-JOPS but significantly better on critical-JOPS. In addition to the tests on SPECjbb2015, a end-to-end runtime test were conducted on the DaCapo 9.12 benchmarks. For 7/11 benchmark applications the runtime in milliseconds proved longer for the Shenandoah GC than G1GC [17].

Shenandoah was also compared on indexing approximately 200GB of wikipedia data using Elasticsearch, on a 100GB heap. In this environment Shenandoah produced a notable slowdown in runtime. However, in terms of latency, (total pause time, max pause time, number of pause times etc.) Shenandoah outshined its competitors G1GC, CMS and Parallel GC [17].

Popular test environments when comparing performance include the benchmark suites DaCapo and SPEC (Standard Performance Evaluation Corporation) Java benchmarks [1] [6] [21] [7][2]. Benchmark suites are valuable research tools when evaluating a features effect on the quality of systems to report the advantages of a new or improved system. The DaCapo suites consist of a set of real-world Java applications with non-trivial memory loads, heavy throughput and complex logic, suitable for performance and memory management analysis [19]. The SPEC Java benchmarks has been specifically developed for the very purpose of measuring Java server performance. Lengauer et al. found, however, that evaluation results from benchmark suites may prove contradictory to real-life memory-intensive applications. They believed the reason for this was the relatively small memory foot-print of the benchmarks.[1] This conclusion motivates the relevancy of this thesis; conducting research in large real-world applications with heavy workloads.

Lengauer et al. also observed that the Garbage First GC produced maximum pause times of up to 3.5 seconds which in many distributed environments is to be recognized as unacceptable[1].

Georgios et Al. found that applications featuring high memory allocation rates can benefit from multithreaded collection and that generational heaps (generational GCs) can offer good performance for applications with a short lifespan tuning-memman-high.

For this study the DaCapo benchmark suite has been chosen. The suite will be used in addition to the evaluation of the Scila environment. This will be done as a pre-evaluation of the collectors to indicate pros and cons and to provide a wider, deeper, perspective of the garbage collector performances. Unfortunately SPECjbb2015 could not be used in the performance evaluation of this study due to budget restraints.

Research	**GCs under evaluation**	**Application(s)**
Java Oracle, ZGC [3]	G1GC, ZGC, Parallel	SPECjbb2015
Red Hat, Shenandoah [17]	G1GC, Shenandoah, Parallel, Par New/CMS	SPECjbb2015, DaCapo 9.12, Elasticsearch
Sun Microsystems Deflet et. al, G1GC [11]	G1GC, CMS	SPECjbb[1], telco
This paper	G1GC, Shenandoah, ZGC	DaCapo 9.12, Node server, Query Server

Table 3.1: A subset of related performance comparisons

Unique for this study will be to evaluate Shenandoah GC, ZGC and G1GC in the same environment, on several metrics. A new aspect will also be to consider a subset of the tuning options of each collector when evaluating the performance.

[1]No SPECjbb version mentioned. Published in 2004, only existing benchmark at that time was SPECjbb2000.

4 Methods

4.1 Experimental setup

4.1.1 Environment

The DaCapo benchmarks, Processing node and Query node tests run on AdoptOpenJDK Java version 13.0.2 with HotSpot JVM. The tests run on a server with the following hardware:

- Quantity: 1 Artnr: SYS-1029P-WTRT
  Name: 1U Dual Skt P Server 10x 2.5" (2xU.2) 12x DDR-4, 2x 10GbaseT WIO 750W RedPwr
- Quantity: 1 Artnr: CD8067303592700
  Name: Xeon Scalable Gold 6154 18C/36T 3G 24.75M 10.4GT UPI 200W
- Quantity: 6 Artnr: M393A4K40CB2-CTD
  Name: Samsung 32GB DDR4 ECC REG 2666MHZ x4 DR
- Quantity: 2 Artnr: MZQLW1T9HMJP
  Name: Samsung Enterprise SSD PM963 1.9TB U.2 2.5" NVMe 7mm (1.3 DWPD)

4.2 JVM and GC options

This section presents and describes the JVM and GC options used within the various configurations formed for the performance tests.

The options -Xmx and -Xms regulate the boundaries of the heap size. -Xmx sets the maximum heap size and -Xms the initial heap size, i.e minimum heap size boundary.

The number of concurrent GC threads can be specified with the *-XX:ConcGCThreads* parameter. With this parameter, we can tweak the performance of a garbage collector as it regulates the parallelism of GC activity in a concurrent phase. Especially relevant for Shenandoah and ZGC which both conduct a lot of work during concurrent phases. More threads favor the GC over the application and fewer threads the other way around.

The default allocated metadata space size is 115mb for all three collectors (on the processing node and query node). The *-XX:MetaspaceSize=256M, -XX:InitialBootClassLoaderMetaspaceSize=32M,*
-XX:MinMetaspaceFreeRatio=50, -XX:MaxMetaspaceFreeRatio=80
-XX:MinMetaspaceExpansion=4M, -XX:MaxMetaspaceExpansion=16M address this area by increasing the space size to 256mb and by configuring the process of expansion and free space ratios. The values are stolen from the current production configurations with good working metadata management.

The *-:XX+UseCompressedOops* option compresses reference pointers by allowing them to be 32-bit in a 64-bit environment and access close to 32gb heap which is more than 32-bit pointers can. This will save a significant amount of memory and can potentially improve performance. However, since the Z Garbage Collector makes use of colored pointers, the references must be 64-bit. Thus the *-XX:UseCompressedOops* is not supported by ZGC.

By applying the *-XX:+UnlockDiagnosticVMOptions* argument one can unlock the *-XX:ObjectAlignmentInBytes=16* option which changes the object alignment to 16 bytes. In Hotspot, the minimum and default object alignment are 8 bytes. Two things happen when we increase this alignment. 1) when alignments get larger, it means that the average space wasted per object will also increase, thus increasing heap occupancy. 2) it enhances the positive effects of the compressed pointers (enabled with *-XX:UseCompressedOops*) to be able to access 64GB heap, thus decreasing heap occupancy. The object alignment option should therefore be used together with the compressed references option.

Another option is the *-XX:UseStringDeduplication* argument which attempts to eliminate duplicate strings as a part of the collection process. During a GC cycle, the JVM will inspect all objects in memory and try to identify duplicate strings among them and eliminate the duplicate appearance and thereby free more memory. Since the string deduplication adds additional work to the GC cycle it may have a smaller negative effect on GC pauses.

-XX:ParallelRefProcEnabled is an option which enables parallelization of the processing of reference objects. This is useful if the processing is taking up a lot of time.

G1GC specific options

The G1GC specific option *-XX:MaxGCPauseMillis*, defines a pause time soft goal which the JVM will try its best efforts to achieve. One way to tweak it is to relax the parameter value from the default 200ms to attempt better throughput. The idea is to allow for longer pause times, which will affect the generation sizing heuristics automatic young generation sizing and by such decrease the frequency of pauses. Tuning the option the other way around instead favors latency by asking for shorter maximum pause times.

Another option of tuning is to increase the *-XX:InitiatingHeapOccupancyPercent* value (default value 45). The Initiating Heap Occupancy Percent (IHOP) sets the occupancy threshold that triggers a marking cycle (percentage of old generation size), as long as there aren't enough observations for G1 to make a good prediction of the initiating heap occupancy threshold. This remains true when adaptive IHOP is active, which is the default. By adapting this to the application the GC can perform well from the start. An increase, for example, means we allow for more allocation before we want to trigger a collection event.

-XX:G1RSetUpdatingPauseTimePercent can be an option of interest when aiming to improve performance. Decreasing this value moves work from the evacuation pause into concurrent refinement threads. This should lower pause time lengths. Increasing the value has the opposite effect and could thus contribute to improved throughput at the cost of longer pause times. Default value 10.

-XX:NewSize set the lower boundary of the young generation size initialized by the JVM.

The *-XX:+ExplicitGCInvokedConcurrent* option acts when a System.gc() is invoked. Instead of invoking a full garbage collection, a stop-the-world, it invokes a concurrent phase mitigating the effects of a full GC.

The *-XX:G1HeapWastePercent* extension is a G1GC specific option that sets the percentage of the heap that is allowed to be wasted. The Java HotSpot VM does then not initiate a mixed garbage collection cycle when the amount of reclaimable space is under the heap waste percentage.

If a static collection more independent of application behavior is desired, the *-XX:G1PeriodicGCInterval* can be of interest. This argument states the interval in ms to check whether G1GC should trigger a periodic collection or not. In other words, the time to pass before a new GC event is initiated. Default value 0.

Yet another tuning aspect of G1GC is to adjust the percentage of reserve

memory to keep free in order to reduce to-space overflows, with *-XX:G1ReservePercent*. The default mode reserves 10 percent.

Shenandoah specific options

The Shenandoah configurations experiment with the Shenandoah GC heuristics; compact, static, and adaptive (default). The compact heuristic runs GC cycles repeatedly, starting the next cycle upon the previous cycle completion, for as long as allocations occur. This would typically incur throughput overhead in favor of delivering nearly immediate space reclamation. The static heuristic, more traditionally determines when to initiate a GC cycle based on heap occupancy and allocation pressure. This is in contrast to the default adaptive heuristic that considers observations of previous cycles. The adaptive heuristic tries to initiate the next cycle when it estimates it could complete the cycle before the heap is exhausted.

The *-XX:ShenandoahAllocSpikeFactor* allows for the specification of how much heap to reserve for absorbing spikes in allocation. Perhaps we expect the application to spike high during certain application events and thus we would want to reserve more heap in order to soften their impact. If we instead know the application to have a stable allocation rate throughout execution we might instead want to utilize more heap space instead of reserving it, decreasing the spike factor.

The *-XX:ShenandoahFreeThreshold* option sets the percentage of free heap space left at which a GC cycle should be triggered.

ZGC specific options

The ZGC collector is sparse with modification parameters but do offer some collector specific options. *-XX:ZAllocationSpikeTolerance* is an option similar to the
-XX:ShenandoahAllocSpikeFactor, which specify the tolerance factor of allocation spikes. Default is factor 2.

In the default configuration, ZGC uncommits unused memory, returning it to the operating system. This is a helpful feature when memory footprint is of concern for the application or environment. However, if memory footprint is not an issue the uncommit feature can be disabled with *-XX:-ZUncommit*.

The use of transparent huge pages for ZGC is not recommended for latency-sensitive applications as it may cause latency spikes. It might however be worth experimenting with to see how the application throughput is affected by it.

-XX:ZCollectionInterval sets a time interval for which a ZGC collection should be triggered. In other words fixed interval garbage collection.

4.2.1 Configurations

The logging parameters used to generate the gc logs: *-Xlog:gc*, gc+ref*, gc+ergo*, gc+heap*,gc+stats*, gc+compaction*, gc+age* :file=gc.log*.

Benchmarks and size options: *avrora (large), jython (large), lusearch-fix (large), h2 (huge), tradebeans (huge)*.

The JVM configurations for each collector were decided and formed by studying the available options for each garbage collector and reading material for tuning these options with a specific purpose.

Tables 4.1, 4.2, 4.3, 4.4 and 4.5 state the various configuration setups used in the performance tests of the DaCapo benchmarks, the processing node and the query node. Options including a "/" varied between applications. If only 2 values are present, e.g "A/B", then A applies to the processing node and B to the query node. If 3 values are present, e.g "A/B/C", A applies to the DaCapo benchmarks, B to the processing node, and C to the query node. This mainly concerns but is not limited to, the -Xmx option.

The configurations named default, g1_default, sh_default, and z_default correspond to the out-of-the-box configuration of each collector[1]. G1_current is the current configuration used in production for the application, using Garbage First GC. For the DaCapo benchmark programs, the heap size settings were reduced from their original value to somewhat better fit the size of the applications. Option -XX:NewSize (lower boundary of young generation size) 12/16G -> 1G and -Xmx (maximum heap size) 63/256G -> 10G.

For the Query node, the JVM option -XX:ConcGCThreads=10 was added to all ZGC configurations except z_moreConc (which utilized 20 threads). Including the default configuration. This option was added due to inability of the ZGC test to finish the query execution without the extra concurrent GC threads, in the project environment setup. Connection to server broke during the long running GC events and/or allocation stalls, thus terminating the tests prematurely.

[1]Z_default is extended with more concurrent GC threads. Explanation further down in this section.

Garbage First GC - I	Options
g1_newsize	-Xmx63G/256G -XX:+UseG1GC -XX:NewSize=1G/12G/16G
g1_compressed	-Xmx63G/256G -XX:+UseG1GC -XX:+UseCompressedOops
g1_expconc	-Xmx63G/256G -XX:+UseG1GC -XX:+UseCompressedOops -XX:+ExplicitGCInvokesConcurrent
g1_lowHeapWaste	-Xmx63G/256G -XX:+UseG1GC -XX:G1HeapWastePercent=5
g1_init	-Xmx63G/256G -XX:+UseG1GC -XX:MaxGCPauseMillis=2000 -XX:InitiatingHeapOccupancyPercent=50
g1_updatePT	-Xmx63G/256G -XX:+UseG1GC -XX:MaxGCPauseMillis=2000 -XX:G1RSetUpdatingPauseTimePercent=15
g1_meta_pure	-Xmx63G/256G -XX:+UseG1GC -XX:MaxGCPauseMillis=2000 -XX:+ExplicitGCInvokesConcurrent -XX:MetaspaceSize=256M -XX:InitialBootClassLoaderMetaspaceSize=32M -XX:MinMetaspaceFreeRatio=50 -XX:MaxMetaspaceFreeRatio=80 -XX:MinMetaspaceExpansion=4M -XX:MaxMetaspaceExpansion=16M
g1_string	-Xmx63G/256G -XX:+UseG1GC -XX:MaxGCPauseMillis=2000 -XX:+UseStringDeduplication
g1_parallel	-Xmx63G/256G -XX:+UseG1GC -XX:MaxGCPauseMillis=2000 -XX:+ParallelRefProcEnabled
g1_period	-Xmx63G/256G -XX:+UseG1GC -XX:G1PeriodicGCInterval=1k
g1_resere	-Xmx63G/256G -XX:+UseG1GC -XX:G1ReservePercent=15
g1_current	-Xmx63G/256G -XX:+UseG1GC -XX:MaxGCPauseMillis=2000 -XX:NewSize=1G/12G/16G -XX:+UseStringDeduplication -XX:+ParallelRefProcEnabled -XX:+UnlockDiagnosticVMOptions -XX:ObjectAlignmentInBytes=16 -XX:+UseCompressedOops -XX:SurvivorRatio=1 -XX:MaxTenuringThreshold=15 -XX:+ExplicitGCInvokesConcurrent -XX:G1HeapWastePercent=5 -XX:MetaspaceSize=256M -XX:InitialBootClassLoaderMetaspaceSize=32M -XX:MinMetaspaceFreeRatio=50 -XX:MaxMetaspaceFreeRatio=80 -XX:MinMetaspaceExpansion=4M -XX:MaxMetaspaceExpansion=16M
g1_default	-Xmx63G/256G -XX:+UseG1GC
g1_throughput	-Xmx63G/256G -XX:+UseG1GC -XX:MaxGCPauseMillis=2000 -XX:InitiatingHeapOccupancyPercent=50 -XX:G1RSetUpdatingPauseTimePercent=15
g1_latency	-Xmx63G/256G -XX:+UseG1GC -XX:MaxGCPauseMillis=1000 -XX:G1RSetUpdatingPauseTimePercent=7 -XX:+ParallelRefProcEnabled

Table 4.1: Configuration setups for Garbage First GC used in the tests

Garbage First GC - II	Options
g1_xms	-Xmx63G/256G -XX:+UseG1GC -Xms32G/128G
g1_compressed_heapwaste	-Xmx63G/256G -XX:+UseG1GC -XX:+UseCompressedOops -XX:G1HeapWastePercent=5 -XX:+UnlockDiagnosticVMOptions -XX:ObjectAlignmentInBytes=16

Table 4.2: Configuration setups for Garbage First GC used in the tests

Shenandoah GC	Options
sh_alloc_spike	-XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC -Xmx63G/256G -XX:ShenandoahAllocSpikeFactor=7
sh_compact4/10	-XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC -Xmx63G/256G -XX:ShenandoahGCHeuristics=compact -XX:ConcGCThreads=4/10
sh_static_freeT	-XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC -Xmx63G/256G -XX:ShenandoahGCHeuristics=static -XX:ShenandoahFreeThreshold=30
sh_string	-XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC -Xmx63G/256G -XX:+UseStringDeduplication
sh_parallel	-XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC -Xmx63G/256G -XX:+ParallelRefProcEnabled
sh_default	-XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC -Xmx63G/256G
sh_compact3/20	-XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC -Xmx63G/256G -XX:ShenandoahGCHeuristics=compact -XX:ConcGCThreads=3/20
sh_static	-XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC -Xmx63G/256G -XX:ShenandoahGCHeuristics=static
sh_static_parallel	-XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC -Xmx256G -XX:ShenandoahGCHeuristics=static -XX:+ParallelRefProcEnabled
sh_meta	-XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC -Xmx63G/256G -XX:MetaspaceSize=256M -XX:InitialBootClassLoaderMetaspaceSize=32M -XX:MinMetaspaceFreeRatio=50 -XX:MaxMetaspaceFreeRatio=80 -XX:MinMetaspaceExpansion=4M -XX:MaxMetaspaceExpansion=16M
sh_xms	-XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC -Xmx256G -Xms128G
sh_moreConc	-XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC -Xmx256G -XX:ConcGCThreads=20

Table 4.3: Configuration setups for Shenandoah GC used in the tests

Z GC - I	Options
z_throughput	-XX:+UnlockExperimentalVMOptions -XX:+UseZGC -Xmx63G/256G -XX:ConcGCThreads=4 -XX:MetaspaceSize=256M -XX:InitialBootClassLoaderMetaspaceSize=32M -XX:MinMetaspaceFreeRatio=50 -XX:MaxMetaspaceFreeRatio=80 -XX:MinMetaspaceExpansion=4M -XX:MaxMetaspaceExpansion=16M
z_parallel	-XX:+UnlockExperimentalVMOptions -XX:+UseZGC -Xmx63G/256G -XX:+ParallelRefProcEnabled -XX:+ExplicitGCInvokesConcurrent -XX:MetaspaceSize=256M -XX:InitialBootClassLoaderMetaspaceSize=32M -XX:MinMetaspaceFreeRatio=50 -XX:MaxMetaspaceFreeRatio=80 -XX:MinMetaspaceExpansion=4M -XX:MaxMetaspaceExpansion=16M
z_defaultTrans	-XX:+UnlockExperimentalVMOptions -XX:+UseZGC -Xmx63G/256G -XX:+UseTransparentHugePage
z_spike	-XX:+UnlockExperimentalVMOptions -XX:+UseZGC -Xmx63G/256G -XX:ZAllocationSpikeTolerance=4
z_uncommit	-XX:+UnlockExperimentalVMOptions -XX:+UseZGC -Xmx63G/256G -XX:+ZUncommit
z_proactive	-XX:+UnlockExperimentalVMOptions -XX:+UseZGC -Xmx63G/256G -XX:+ZProactive
z_interval	-XX:+UnlockExperimentalVMOptions -XX:+UseZGC -Xmx63G/256G -XX:ZCollectionInterval=30

Table 4.4: Configuration setups for Z GC used in the tests

Z GC - II	Options
z_default	-XX:+UnlockExperimentalVMOptions -XX:+UseZGC -Xmx63G/256G
z_string	-XX:+UnlockExperimentalVMOptions -XX:+UseZGC -Xmx63G/256G -XX:+UseStringDeduplication -XX:+ParallelRefProcEnabled -XX:+ExplicitGCInvokesConcurrent -XX:MetaspaceSize=256M -XX:InitialBootClassLoaderMetaspaceSize=32M -XX:MinMetaspaceFreeRatio=50 -XX:MaxMetaspaceFreeRatio=80 -XX:MinMetaspaceExpansion=4M -XX:MaxMetaspaceExpansion=16M
z_meta	-XX:+UnlockExperimentalVMOptions -XX:+UseZGC -Xmx63G/256G -XX:MetaspaceSize=256M -XX:InitialBootClassLoaderMetaspaceSize=32M -XX:MinMetaspaceFreeRatio=50 -XX:MaxMetaspaceFreeRatio=80 -XX:MinMetaspaceExpansion=4M -XX:MaxMetaspaceExpansion=16M
z_throughput4/3	-XX:+UnlockExperimentalVMOptions -XX:+UseZGC -Xmx63G/256G -XX:ConcGCThreads=4/3 -XX:MetaspaceSize=256M -XX:InitialBootClassLoaderMetaspaceSize=32M -XX:MinMetaspaceFreeRatio=50 -XX:MaxMetaspaceFreeRatio=80 -XX:MinMetaspaceExpansion=4M -XX:MaxMetaspaceExpansion=16M
z_xms	-XX:+UnlockExperimentalVMOptions -XX:+UseZGC -Xmx63G/256G -Xms32G/128G
z_uncommit_meta	-XX:+UnlockExperimentalVMOptions -XX:+UseZGC -Xmx63G -XX:+ZUncommit -XX:MetaspaceSize=256M -XX:InitialBootClassLoaderMetaspaceSize=32M -XX:MinMetaspaceFreeRatio=50 -XX:MaxMetaspaceFreeRatio=80 -XX:MinMetaspaceExpansion=4M -XX:MaxMetaspaceExpansion=16M
z_moreConc	-XX:+UnlockExperimentalVMOptions -XX:+UseZGC -Xmx63G/256G -XX:ConcGCThreads=7/20

Table 4.5: Configuration setups for Z GC used in the tests

4.3 Execution

4.3.1 DaCapo benchmarks

The benchmarks chosen from the DaCapo suite are avrora, jython, h2, lusearch-fix and tradebeans. Tradebeans and h2 have the longest execution times and tend to allocate the largest heaps in the set. They are therefore of interest as they produce more log data than other benchmarks in the set. Avrora and jython are of medium size relative to the benchmarks in the set, while lusearch-fix represents the smaller applications.

Each benchmark is executed once for each configuration setup. All benchmark programs run until termination (task completion) each time. They run with the extension of their respective size configurations mentioned in section 4.2.1 Configurations. Each run generates a log file of the garbage collector behavior which is sent to GCEasy, via REST API, for analysis. The responses are individually stored in new files. Certain values, such as total STW pause time and total concurrent time, is calculated by GCEasy although not included in the response body. These values are manually looked up, from the web-report generated by GCEasy linked within the rest response.

The data from the generated response files are then categorized by benchmark and assembled into one .csv file per benchmark for manual analysis, comparison, and visualization.

4.3.2 Processing node

The processing node is run once for each configurations setup of G1GC, Shenandoah GC, and ZGC. The system limit on the number of maximum memory mappings per process is set to 120000 (originally 65530) in the environment. ZGC requests at least 116121 allowed memory mappings for a Java maximum heap size of 63GB to avoid risks of fatal errors such as failure to map memory. The execution is terminated when a threshold of 37700000 processed messages has been reached, to ensure an equal amount of processed data. The execution is run on the historical date 2020/03/03 from days start. Each run generates a log file of the garbage collector behavior which is sent to GCEasy, via REST API, for analysis. The responses are individually stored in new files. Values of total STW pause time and total concurrent time are manually looked up from GCEasy. The data from the generated response files are then assembled into a single .csv file for manual analysis.

Additionally, a subset of the gc-log data files are uploaded to jClarity Cen-

sum for further inspection.

4.3.3 Query node

The query node is run once for each configurations setup of G1GC, Shenandoah GC, and ZGC. The system limit on the number of maximum memory mappings per process is set to 500000 (originally 65530) in the test environment. ZGC requests at least 471859 allowed memory mappings for a Java maximum heap size of 256GB to avoid risks of fatal errors such as failure to map memory. After startup, a query is called with the parameter value 300. This specific query investigates time windows of 300s throughout the day of the working date. The execution of the query node is shut down upon the completion of the triggered query. The execution is run on the historical date 2020/03/04 which triggers the query to analyze data of 2020/03/03. Each run generates a log file of the garbage collector behavior which is sent to GCEasy, via REST API, for analysis. The responses are individually stored in new files. Values of total STW pause time and total concurrent time are manually looked up from GCEasy. The data from the generated response files are then assembled into a single .csv file for manual analysis.

Additionally, a subset of the gc-log data files are uploaded to jClarity Censum for further inspection.

4.4 Metrics definitions

Throughput percentage: Percentage of time spent in application workload compared to time spent in GC activity of the total time. A higher percentage is a good indication that GC overhead is low. Concurrent GC work is not regarded as GC time but included in application time.
Measurement duration: Total time duration of the GC logging.
JVM heap size allocation: The allocated size of heap memory.
JVM heap size peak: The memory peak utilization size.
Average allocation rate: The average memory allocation rate representing the object creation rate by the application.
Average pause time: The average amount of time taken by one STW GC pause.
Maximum pause time: The maximum amount of time taken by one STW GC pause.
Total STW time: Total accumulated time all application threads have been stopped for GC activity (Stop-the-world time).

Total concurrent time: Total accumulated time of GC activity concurrent with application threads.

5 Results

5.1 DaCapo

Figure 5.1 displays GC-Easy data analysis results of the DaCapo benchmark applications. The figure shows the tests conducted with the default configurations for each garbage collector. Blank cells in the table of allocation rates are due to no such data being provided by GC-Easy. All applications ran until termination which makes measurement duration (M-Duration) an indicator of throughput. For applications H2 and Jython the results show lower measured duration time for G1GC compared to Shenandoah and ZGC, > 9% lower for H2 and >7% Jython. Simultaneously Jython and H2 show contradictory higher throughput percentage values for Shenandoah and ZGC compared to G1GC. Shenandoah and ZGC produced at least 87% lower average pause time values (AvgPT) and at least 83% lower maximum pause time values (MaxPT) than G1GC for all DaCapo applications in default mode. The throughput percentage (Throughput%) value is also slightly higher for Shenandoah and ZGC compared to G1GC. Shenandoah and ZGC perform very similarly in that metric. G1GC answered to the highest total accumulated stop-the-world (STW) time for all applications. Shenandoah and ZGC produced > 87% lower accumulated STW times. Shenandoah proved slightly lower total STW values than ZGC for the longer running applications H2 and Tradebeans, and slightly higher for Avrora, Jython and Lusearch-fix. An increase of total time spent in concurrent cycles (Total conc) can also be matched to a lower total STW value for the three longer running applications H2, Tradebeans, and Jython. ZGC produced the highest average allocation rate (AvgAllocRate) for all 5 of the applications. The table in figure 5.1 shows that G1GC has allocated and used (peak usage) the least JVM heap size while ZGC always allocated and used the most.

For the default configurations, Shenandoah allocated 18 threads in parallel

Label	M-duration	jvmHeap alloc	jvmHeap peak	AvgAllocRate	Throughput%	avgPT (ms)	maxPT (ms)	Total STW	Total conc
Avrora									
g1_default	31 sec 18 ms	40 mb	25 mb	7.77 mb/sec	99.705	8.3180895	10.0	91.5ms	12.8ms
sh_default	28 sec 726 ms	2 gb	24 mb		99.993	0.51975	0.992	2.08ms	3.03ms
z_default	38 sec 855 ms	2 gb	16 mb	421 kb/sec	99.999	0.176	0.246	0.528ms	12.1ms
H2									
g1_default	5 min 44 sec 821 ms	3.49 gb	2.47 gb	56.47 mb/sec	99.643	35.19223	230.0	1s 232ms	550ms
sh_default	6 min 19 sec 262 ms	3.15 gb	2.51 gb	25.76 mb/sec	99.996	0.4817188	1.976	15.4ms	2s 35ms
z_default	6 min 21 sec 855 ms	5.83 gb	5.76 gb	67.24 mb/sec	99.996	0.3795854	1.398	15.6ms	7s 842ms
Tradebeans									
g1_default	2 min 15 sec 624 ms	2 gb	884 mb	250.06 mb/sec	99.253	14.467321	70.0	1s 13ms	59ms
sh_default	2 min 11 sec 223 ms	9.02 gb	8.52 gb	265.15 mb/sec	99.958	1.1428332	8.039	54.9ms	816ms
z_default	2 min 4 sec 883 ms	9.71 gb	9.36 gb	371.05 mb/sec	99.948	1.5868778	7.657	65.1ms	2s 488ms
Jython									
g1_default	17 sec 438 ms	1.09 gb	685 mb	552.53 mb/sec	99.197	11.666667	20.0	140ms	0
sh_default	18 sec 755 ms	2.53 gb	2.51 gb	531.91 mb/sec	99.936	0.7493125	3.278	12ms	71.6ms
z_default	18 sec 993 ms	3.51 gb	3.42 gb	665.4 mb/sec	99.956	0.46400002	2.124	8.35ms	188ms
Lusearch-fix									
g1_default	1 sec 452 ms	1.22 gb	572 mb	1.1 gb/sec	98.537	5.3107495	10.0	21.2ms	20.5ms
sh_default	2 sec 218 ms	2 gb	20 mb		99.885	0.63824993	1.265	2.55ms	4.45ms
z_default	1 sec 441 ms	2.42 gb	2.26 gb	1.55 gb/sec	99.898	0.24499997	0.532	1.47ms	45ms

Figure 5.1: DaCapo benchmark suite results of out-of-the-box configurations.

and 9 threads to be concurrent workers. ZGC allocated 22 threads in parallel and 5 threads concurrent.

Further tuned configurations tested on the DaCapo benchmarks yielded no major differences. The results may be found in the appendix section A.

5.2 Processing node

5.2.1 Default configurations

Figure 5.2 show the result data from GC-Easy of the process node run with the out-of-the-box configurations for each collector with the additional option of *-Xmx63G*. The ZGC default configuration experienced a non-fatal problem detected by GC-Easy; the Metadata occupancy is reaching its limit quite often and triggering garbage collection. This suggests the allocated metadata space is too small. ZGC allocated 115mb for metadata, peak usage 113mb. The pause time values of average pause time (AvgPT), maximum pause time (MaxPT), and total STW time were significantly higher for G1GC compared to

Shenandoah and ZGC. Between ZGC and Shenandoah, ZGC performs notably better in pause times. ZGC produces the highest throughput percentage value, followed by Shenandoah, and lastly G1GC. However G1GC was the fastest collector in terms of measured duration time of the GC logging (M-duration). Allocation and usage of JVM heap size were similar between the three collectors but with ranking order 1. Shenandoah GC, 2. G1GC, 3. ZGC, ranked low usage to high. The average allocation rate greatly varies between the collectors with G1GC on 516.76mb/sec, Shenandoah GC on 463.65mb/sec, and ZGC with the lowest allocation rate of 348.02mb/sec.

Default configurations

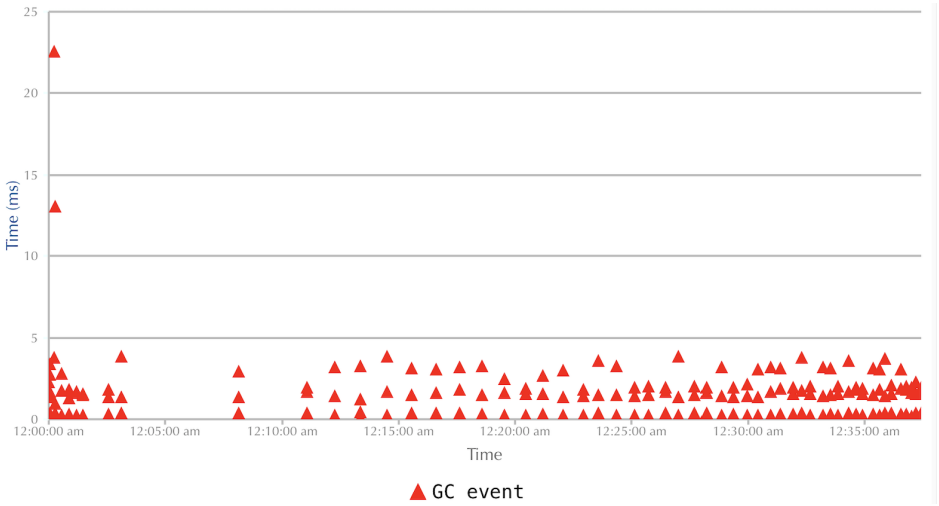
Label	Problem	M-duration	jvmHeap alloc	jvmHeap peak	AvgAllocRate	Throughput%	avgPT (ms)	maxPT (ms)	Total STW	Total conc
g1_default	False	36 min 34 sec 297 ms	63 gb	55.68 gb	516.76 mb/sec	99.487	41.70864	340.0	11s 261ms	5m 28s 625ms
sh_default	False	37 min 42 sec 381 ms	63 gb	53.53 gb	463.65 mb/sec	99.987	1.3052895	22.551	298ms	3m 12s 871ms
z_default	True	37 min 16 sec 8 ms	60.84 gb	58.86 gb	348.02 mb/sec	99.993	0.8283775	3.664	162ms	10m 34s 732ms

Figure 5.2: Process node GC-Easy results of out-of-the-box configurations.

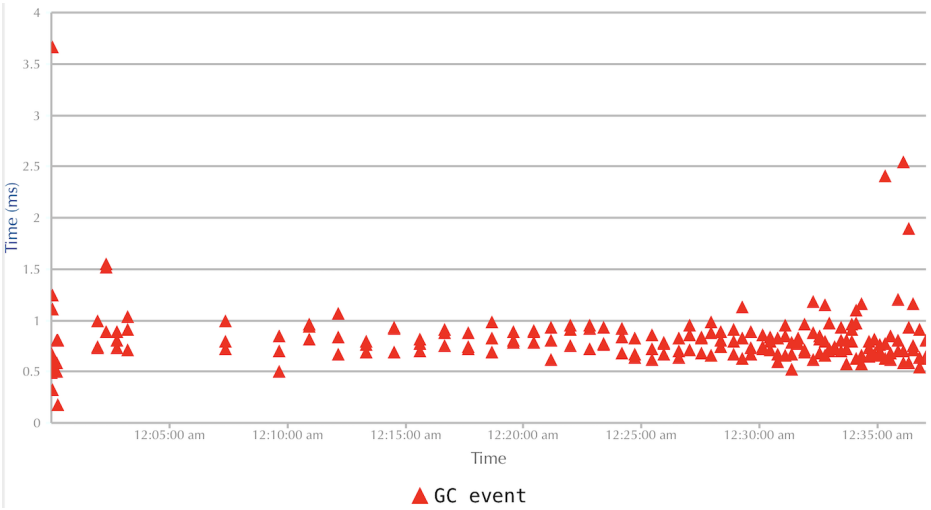
For the default configurations, Shenandoah allocated 18 threads in parallel and 9 threads to be concurrent workers. ZGC allocated 22 threads in parallel and 5 threads concurrent.

Figure 5.3 (a), (b) and (c) provide further information regarding the occurred pause times and their duration variation. Note that the scale varies on the y-axis. For Shenandoah and ZGC the pause times increased in frequency toward the end of the execution. They both had very low duration variation except for a few stray events. G1GC default conducted one full GC early on, thereafter only young generation GC events occurred. G1GC undergo almost a 100 pause time events of 50-150ms. These are however countered by even more events in the range 0-50ms, causing the average pause time of 41.7ms (figure 5.3 (c)).

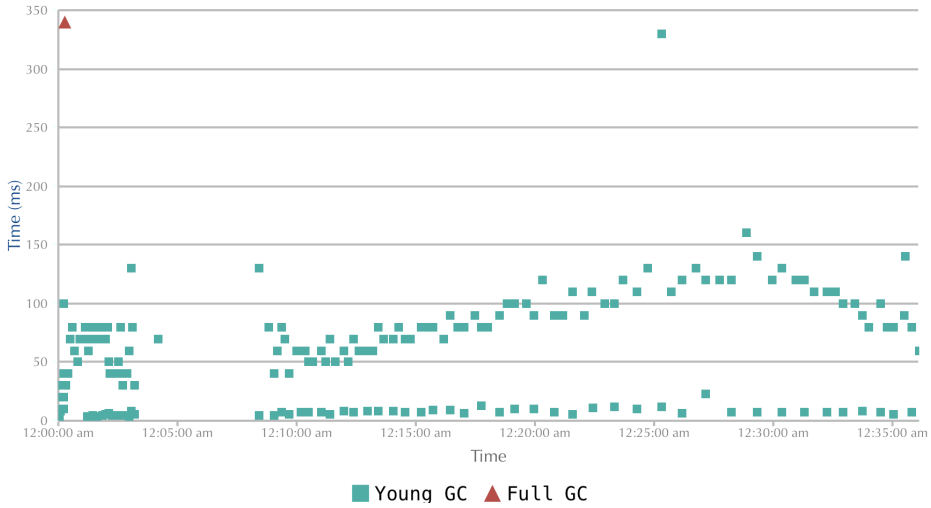
The allocation rates of the runs are showed in graphs in figure 5.4 (a), (b) and (c). Note the variations of scale in the y-axis. The g1_default configuration maintains a spread out allocation rate, although showing continuous ups and downs within the bounds of 2000-20000mb. ZGC and Shenandoah instead allocate heavily and fast to begin with (peaks of 58000mb and 47000mb respectively) and dramatically decrease the allocation rate almost linearly as the execution moves on. ZGC reaches allocation rates below 1000mb in the last minutes of the execution. Shenandoah never allocates under the rate of 5000mb.



(a) Sh_default.

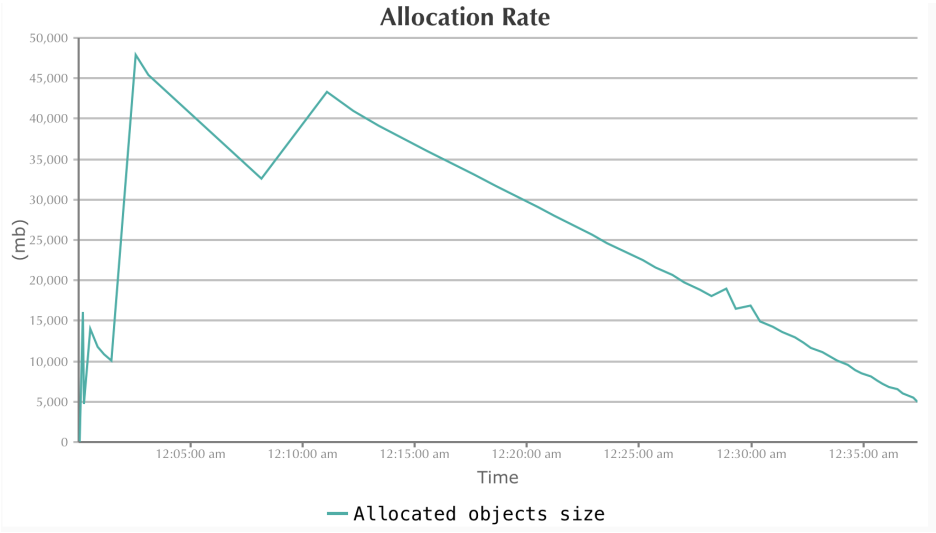


(b) Z_default..

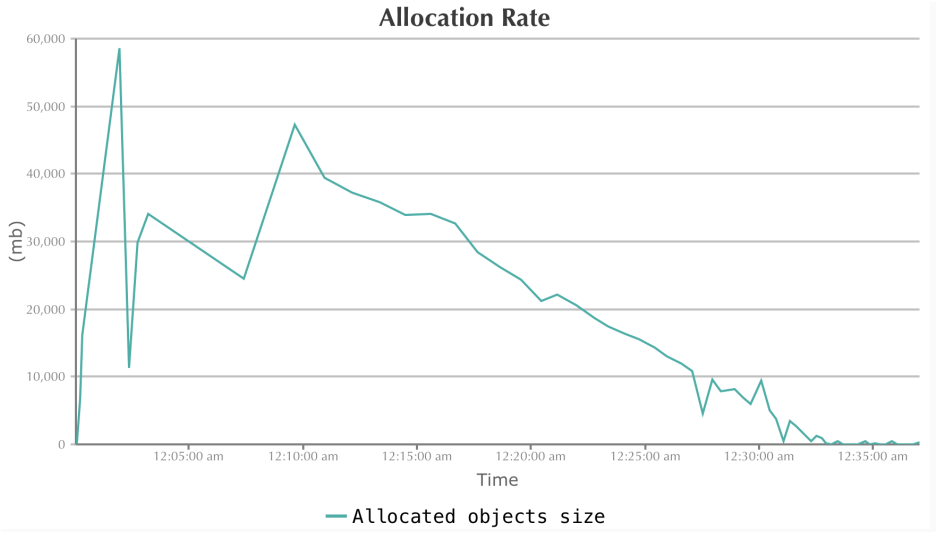


(c) G1_default.

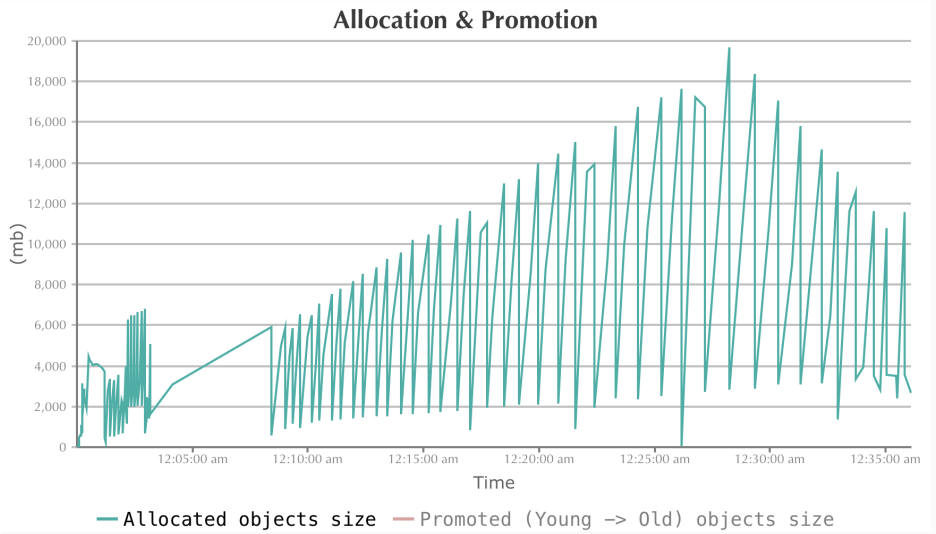
Figure 5.3: GC Pause Duration time, generated by GC-Easy.



(a) Sh_default.



(b) Z_default..

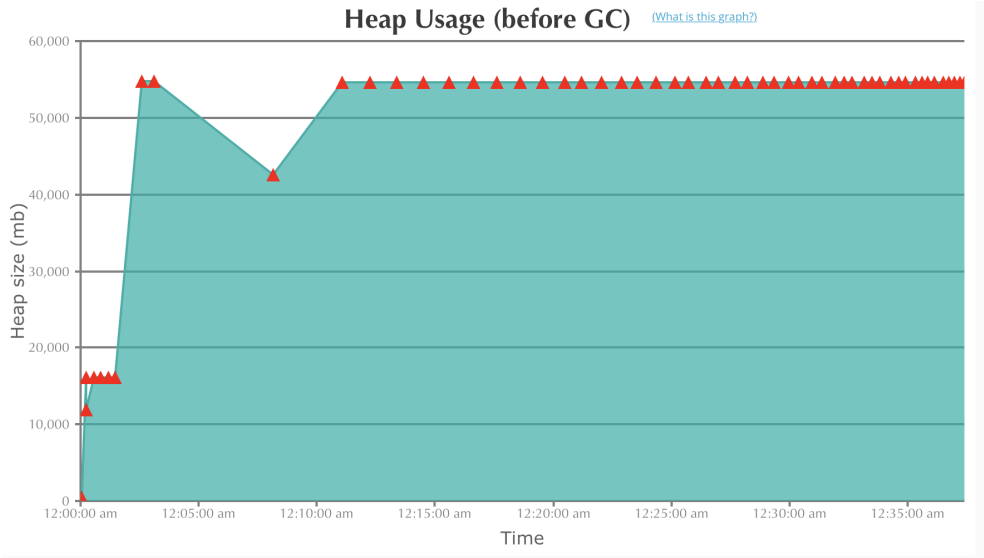


(c) G1_default.

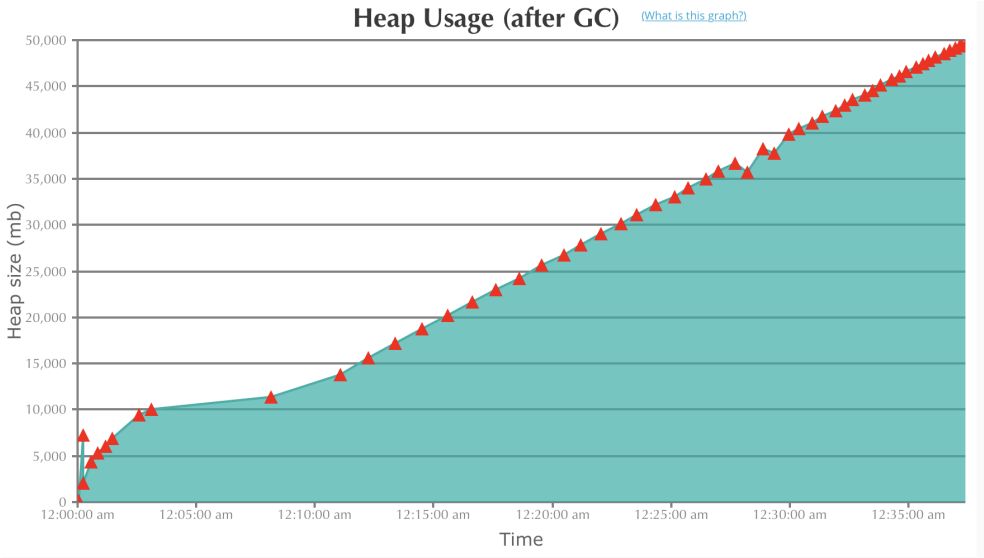
Figure 5.4: GC allocation rates, generated by GC-Easy.

The collectors heap usage during execution are illustrated in figure 5.5 - 5.7. Note that the y-axis varies in scale. Shenandoah and ZGC show a rather stable curve of heap usage before GC events, triggering the GC at similar thresholds. 55000mb and 50000-59000mb respectively. Their heap usage curves after a GC event conforms to linearity which means that less and less memory could be reclaimed by each triggered GC. The G1GC, on the other hand, allows for a higher and higher threshold before triggering a GC event, starting at a lower initial threshold and planning out around 58000mb. The heap usage curve before GC is steeper than the heap usage curve after the GC which reveals a growing reclamation of memory during the majority of execution until the "usage before GC" plans out (around 55000mb heap size). ZGC initially reaches extremely close to the heap maximum limit of 63000mb.

The JClarity Censum analytics tool alarms of high CPU usage for the G1GC default configuration. The kernel time exceeded the user time 7 times, and GC threads collected an abnormal amount of kernel time 57times. High kernel CPU utilization indicates that the reason for long pauses may lie outside of the JVM. Other measurements provided by Censum align with the data from GC-Easy. For Shenandoah and ZGC default configurations no additional information was provided by Censum. The tool did not analyze the Shenandoah and ZGC logs in great depth.

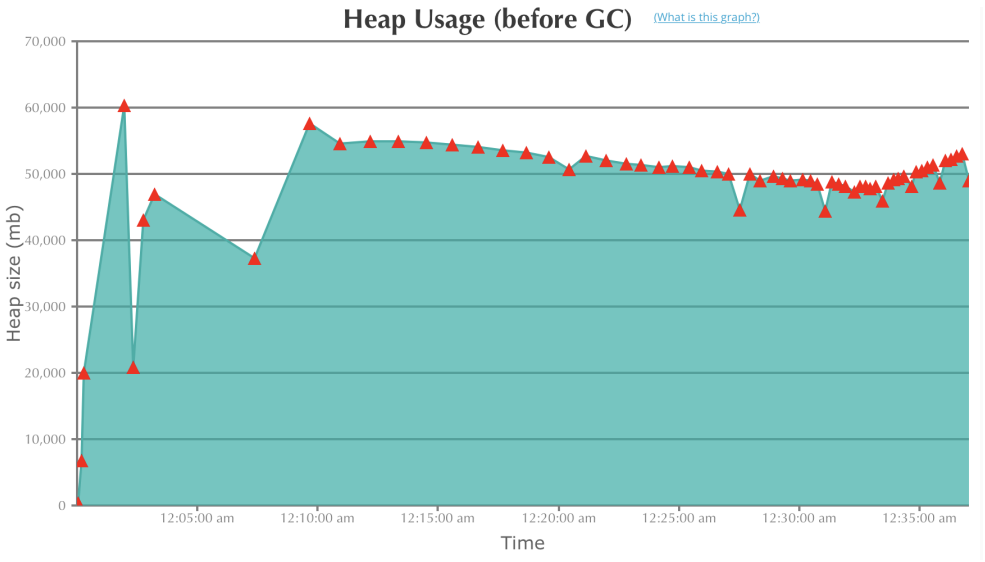


(a) Shenandoah before.

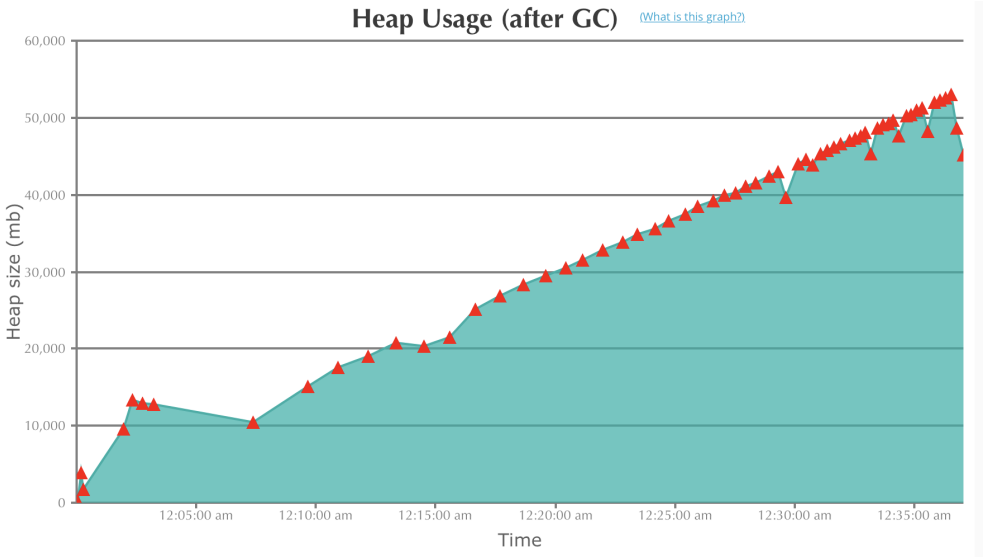


(b) Shenandoah after.

Figure 5.5: Shenandoah GC - Heap Usage Before/After GC event, generated by GC-Easy.

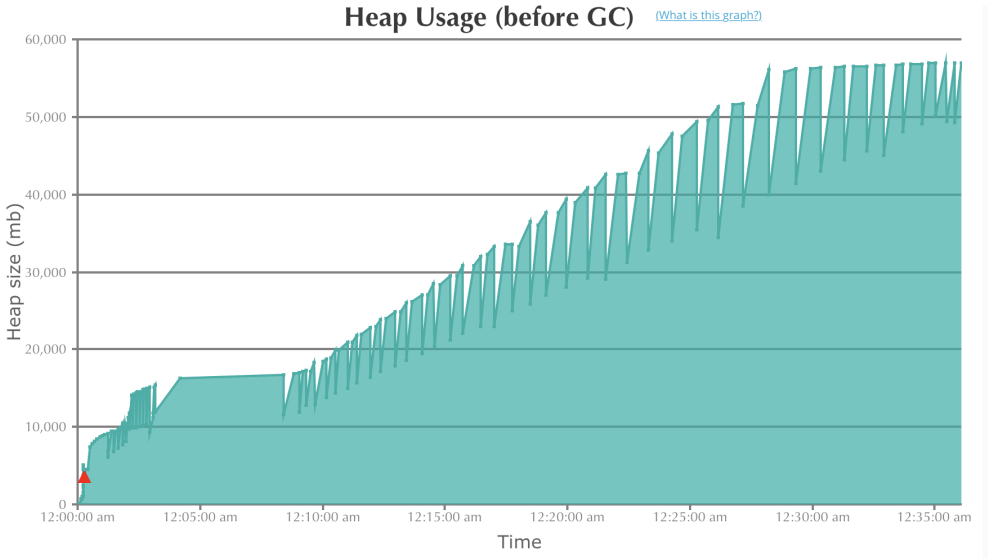


(a) Z before.

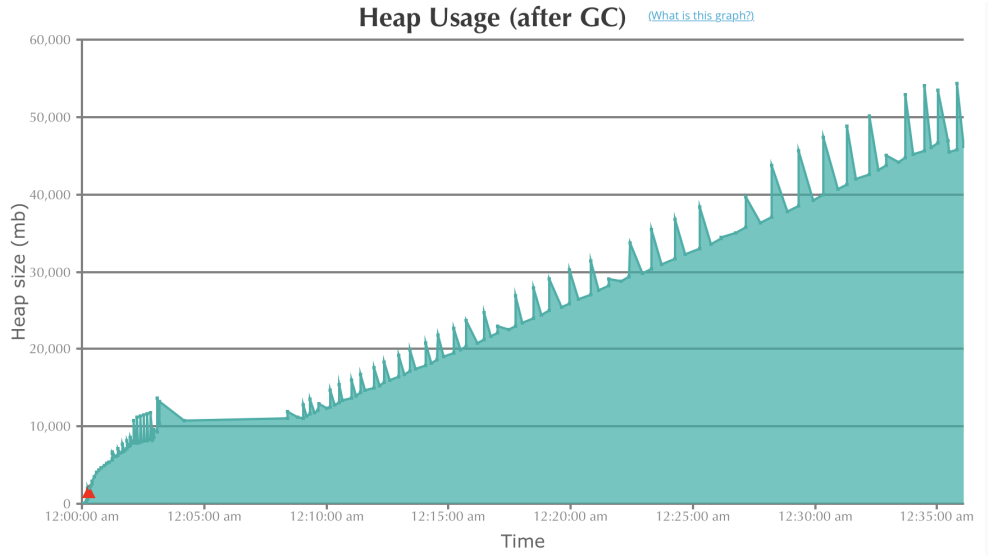


(b) Z after.

Figure 5.6: ZGC - Heap Usage Before/After GC event, generated by GC-Easy.



(a) G1 before.



(b) G1 after.

Figure 5.7: G1GC - Heap Usage Before/After GC event, generated by GC-Easy.

5.2.2 Modified configurations

Shenandoah GC

Label	Problem	M-duration	jvmHeap alloc	jvmHeap peak	AvgAllocRate	Throughput%	avgPT (ms)	maxPT (ms)	Total STW	Total conc
sh_alloc_spike	False	37 min 27 sec 364 ms	63 gb	53.54 gb	457.52 mb/sec	99.987	1.2160301	12.917	282ms	3m 20s 91ms
sh_compact4	False	44 min 35 sec 770 ms	55.05 gb	49.85 gb	212.15 mb/sec	99.958	1.4543287	74.196	1s 123ms	20min27s 421ms
sh_static_freeT	False	38 min 44 sec 457 ms	53.67 gb	50.24 gb	347.24 mb/sec	99.972	1.0140258	12.347	656ms	11m 5s 193ms
sh_string	False	38 min 20 sec 513 ms	63 gb	53.53 gb	450.44 mb/sec	99.341	67.67501	704.575	15s 159ms	3m 17s 991ms
sh_parallel	False	38 min 2 sec 431 ms	63 gb	53.51 gb	454.92 mb/sec	99.988	1.2395747	13.069	283ms	3m 14s 167ms
sh_default	False	37 min 42 sec 381 ms	63 gb	53.53 gb	463.65 mb/sec	99.987	1.3052895	22.551	298ms	3m 12s 871ms
sh_compact3	False	45 min 25 sec 815 ms	56.47 gb	49.94 gb	150.27 mb/sec	99.962	1.3484429	21.954	1s 41ms	27m 1s 614ms
sh_static	False	37 min 49 sec 374 ms	63 gb	53.57 gb	467.08 mb/sec	99.99	1.1076081	12.575	235ms	3m 6s 969ms
sh_meta	False	38 min 15 sec 410 ms	63 gb	53.51 gb	447.26 mb/sec	99.987	1.4193493	29.829	301ms	3m 8s 329ms
sh_xms	False	38 min 15 sec 400 ms	63 gb	53.52 gb	452.93 mb/sec	99.987	1.3565719	24.448	304ms	3m 9s 15ms
sh_static_parallel	False	38 min 10 sec 378 ms	63 gb	53.58 gb	457.08 mb/sec	99.99	1.1061883	11.963	235ms	3m 7s 508ms
sh_compressed	False	38 min 24 sec 432 ms	63 gb	53.54 gb	451.64 mb/sec	99.987	1.3339907	24.385	299ms	3m 8s 867ms

Figure 5.8: Process node GC-Easy results of Shenandoah GC configurations.

Figure 5.8 show a set of performance values produced by different configurations of Shenandoah GC on the processing node application. Memory usage is similar for all and lies in the range 49-54gb. The average allocation rate is significantly lower for the sh_compact4 and sh_compact3 configurations which used the compact heuristic in combination with fewer concurrent threads than the default (5), assigned to the GC. Sh_compact4 and sh_compact3 thus experienced longer measured duration times and spent longer periods of time in an STW state and concurrent work alongside the application. The sh_static configuration, which utilized the static heuristic, performed the highest throughput percentage value (along with sh_static_parallel) and amongst the lower measured duration times while decreasing the pause times, average, max and total STW compared to the default configuration. Sh_string, which tested the *-XX:+UseStringDeduplication* option executed the application with immensely higher pause times than the other Shenandoah configurations along with a decreased throughput percentage.

The usage of parallel reference processing (sh_parallel, sh_static_parallel) slightly improved the total STW time and average pause time and almost halved the maximum pause time compared to the default Shenandoah configuration. Sh_alloc_spike ran the shortest measured duration time and performed amongst the top configurations in throughput and latency.

Figure 5.9 display the processing node test results of various ZGC configurations. GC-Easy detected problems for several of the configurations. For z_throughput allocation stalls of a total of 47sec and 456ms occurred during execution. An allocation stall occurs when the allocation rate of the applica-

ZGC

Label	Problem	M-duration	jvmHeap alloc	jvmHeap peak	AvgAllocRate	Throughput%	avgPT (ms)	maxPT (ms)	Allocation stall	Total STW	Total conc
z_throughput4	False	37 min 17 sec 833 ms	62.46 gb	59.78 gb	314.01 mb/sec	99.992	0.93210995	10.402	-	186ms	12m 58s 390ms
z_parallel	False	36 min 29 sec 647 ms	60.66 gb	58.78 gb	349.57 mb/sec	99.992	0.91823316	10.871	-	177ms	10m 49s 80ms
z_defaultTrans	True	37 min 23 sec 865 ms	61.44 gb	59.36 gb	349.33 mb/sec	99.993	0.8505073	7.8389997	-	168ms	10m 46s 385ms
z_spike	True	37 min 54 sec 398 ms	59.39 gb	56.37 gb	267.91 mb/sec	99.985	0.87348956	8.213	-	335ms	15m 33s 306ms
z_uncommit	True	37 min 39 sec 588 ms	61.28 gb	59.29 gb	351.74 mb/sec	99.993	0.82908255	7.075	-	161ms	10m 23s 985ms
z_proactive	True	37 min 43 sec 59 ms	61.67 gb	59.67 gb	354.57 mb/sec	99.992	0.8914101	7.151	-	174ms	10m 33s 743ms
z_interval	True	36 min 50 sec 202 ms	60.47 gb	52.12 gb	341.26 mb/sec	99.987	0.8735181	8.17	-	292ms	12m 20s 252ms
z_default	True	37 min 16 sec 8 ms	60.84 gb	58.86 gb	348.02 mb/sec	99.993	0.8283775	3.664	-	162ms	10m 34s 732ms
z_string	False	37 min 4 sec 722 ms	60.64 gb	56.34 gb	341.79 mb/sec	99.991	0.78663975	8.708	-	205ms	11m 10s 971ms
z_meta	False	36 min 29 sec 867 ms	60.31 gb	56.49 gb	345.44 mb/sec	99.992	0.8518965	8.305	-	164ms	10m 46s 49ms
z_throughput3	True	37 min 14 sec 297 ms	63 gb	59.75 gb	242.43 mb/sec	99.992	0.8889997	8.495	47 sec 456 ms	183ms	17m 51s 148ms
z_xms	True	36 min 20 sec 951 ms	61.7 gb	59.74 gb	359.95 mb/sec	99.993	0.73562485	3.322	-	147ms	10m 42s 129ms
z_uncommit_meta	False	36 min 16 sec 178 ms	60.48 gb	56.14 gb	357.29 mb/sec	99.992	0.94243115	8.948	-	179ms	10m 33s 505ms
z_moreConc	True	36 min 35 sec 982 ms	61.27 gb	59.82 gb	421.7 mb/sec	99.994	0.7891813	6.607	-	135ms	6m 32s 577ms

Figure 5.9: Process node GC-Easy results of ZGC configurations.

tion is higher than what the ZGC garbage collector can handle, and the heap runs out of available space. The ZGC then slows down the application threads to keep up. The low z_throughput average allocation rate serves as further evidence of this serious memory issue. The remaining detected problems by GC-Easy are rooted in metadata space occupancy. The allocated space for metadata is too small which leads the occupancy to reach its limit often and triggering GC events. This is no fatal issue but should be avoided by allocating larger memory space for metadata and thus minimizing this type of GC activity. The allocated metadata space for these configurations was 115mb (default). The z_meta configuration addresses the metadata occupancy issue by assigning 256mb space and configuring the expansion and free ratio of the metadata space. Z_moreConc configuration produced the highest average allocation rate amongst the ZGC configurations by assigning more concurrent threads, 7, to GC work. As opposed to the default 5 concurrent GC threads. A vaguely higher throughput percentage value and a reduced measured duration time, when measured against the default ZGC configuration, can also be observed. Compared to Shenandoah GC and G1GC results the average allocation rate of z_moreConc still seems to be within bounds of reason for the processing node application. Z_xms produced the shortest maximum pause time and average pause time while z_moreConc responds to the shortest accumulated STW time. The pause time values for most ZGC configurations are however very closely knitted.

Figure 5.10 showcase the results of a set of configurations run with the Garbage First GC on the processing node. The g1_throughput configuration, a combination of g1_init, and g1_updatePT, delivered a decreased mea-

G1GC

Label	Problem	M-duration	jvmHeap alloc	jvmHeap peak	AvgAllocRate	Throughput%	avgPT (ms)	maxPT (ms)	Total STW	Total Conc
g1_newsize	False	36 min 8 sec 389 ms	63 gb	56.85 gb	527.42 mb/sec	99.534	59.786118	250.0	10s 104ms	4m 24s 261ms
g1_compressed	False	36 min 46 sec 357 ms	63 gb	55.69 gb	516.64 mb/sec	99.503	40.629723	490.0	10s 970ms	5m 31s 985ms
g1_expconc	False	37 min 26 sec 378 ms	63 gb	55.87 gb	507.5 mb/sec	99.537	44.482018	203.364	10s 409ms	5m 24s 493ms
g1_lowHeapWaste	False	36 min 31 sec 317 ms	63 gb	55.71 gb	519.91 mb/sec	99.504	41.971054	440.0	10s 871ms	5m 32s 206ms
g1_init	False	35 min 57 sec 282 ms	63 gb	55.88 gb	529.59 mb/sec	99.493	48.14704	190.0	10s 929ms	5m 4s 928ms
g1_updatePT	False	36 min 22 sec 298 ms	63 gb	55.65 gb	523.57 mb/sec	99.457	45.26802	450.0	11s 860ms	5m 33s 477ms
g1_meta_pure	False	36 min 17 sec 307 ms	63 gb	55.81 gb	521.39 mb/sec	99.483	51.63398	310.0	11s 256ms	5m 16s 6ms
g1_string	False	37 min 45 sec 35 ms	63 gb	55.62 gb	506.89 mb/sec	98.811	109.03887	640.0	26s 933ms	5m 34s 968ms
g1_parallel	False	36 min 30 sec 294 ms	63 gb	55.67 gb	519.43 mb/sec	99.459	45.432377	550.0	11s 858ms	5m 30s 943ms
g1_period	False	38 min 26 sec 437 ms	63 gb	55.66 gb	500.4 mb/sec	98.824	20.76629	400.0	27s 121ms	34m 20s 390ms
g1_resere	False	37 min 14 sec 344 ms	63 gb	52.81 gb	510.4 mb/sec	99.501	37.785763	330.0	11s 147ms	6m 35s 965ms
g1_current	True	38 min 38 sec 470 ms	53.97 gb	42.63 gb	424.41 mb/sec	98.338	247.03992	1060.0	38s 538ms	3m 28s 967ms
g1_default	False	36 min 34 sec 297 ms	63 gb	55.68 gb	516.76 mb/sec	99.487	41.70864	340.0	11s 261ms	5m 28s 625ms
g1_throughput	False	35 min 30 sec 330 ms	63 gb	55.68 gb	534.93 mb/sec	99.428	48.76063	550.0	12s 190ms	5m 12s 960ms
g1_latency	False	36 min 5 sec 286 ms	63 gb	55.64 gb	528.93 mb/sec	99.449	45.166157	310.0	11s 924ms	5m 26s 892ms
g1_xms	True	36 min 9 sec 276 ms	63 gb	55.7 gb	522.99 mb/sec	99.518	60.03539	600.0	10 446ms	4m 15s 516ms
g1_compressed_heapwaste	False	38 min 41 sec 580 ms	62.95 gb	52.67 gb	428.07 mb/sec	99.512	46.974335	230.0	11s 321ms	4m 56s 280ms

Figure 5.10: Process node GC-Easy results of Garbage First GC configurations.

sured duration time by 1min compared to the default G1GC configuration, and 3min compared to g1_current. However the trade-off expresses itself in increased latency values (average and max pause time, total STW). G1_current and g1_compressed_heapwaste achieve circa 100mb/sec lower average allocation rates than the rest of the G1GC configurations. The three configurations all utilize *-XX:+UseCompressedOops, -XX:G1HeapWastePercent=5* and *-XX:ObjectAlignmentInBytes=16*. G1_expconc, g1_compressed_heapwaste, g1_init and g1_newsize had lower maximum pause times, but moderately higher average pause times compared to g1_default. G1_meta and g1_string yielded a doubled average pause time and an increase of maximum pause time and total accumulated STW time.

G1_current and g1_xms both experience problems detected by GC-Easy. The issue was occurrences of GC events where system time was greater than user time. This is not a healthy sign. In normal GC events, most time should be spent within the JVM for GC activity. If the application has multiple occurrences of this problem it could indicate issues that lie outside of the VM. G1_current and g1_xms experienced it once each.

Figure 5.11 displays 2 of the top performers of each collector in the same table. Comparing these, ZGCs z_moreConc and z_uncommit_meta processed the application data in the shortest measured duration times and had the highest throughput percentages. Additionally the two ZGC configurations delivered the lowest average pause times, maximum pause times and STW values.

Label	Problem	M-duration	jvmHeap alloc	jvmHeap peak	AvgAllocRate	Throughput%	avgPT (ms)	maxPT (ms)	Total STW	Total conc
sh_alloc_spike	False	37 min 27 sec 364 ms	63 gb	53.54 gb	457.52 mb/sec	99.987	1.2160301	12.917	282ms	3m 20s 91ms
sh_static	False	37 min 49 sec 374 ms	63 gb	53.57 gb	467.08 mb/sec	99.99	1.1076081	12.575	235ms	3m 6s 969ms
z_moreConc	True	36 min 35 sec 982 ms	61.27 gb	59.82 gb	421.7 mb/sec	99.994	0.7891813	6.607	135ms	6m 32s 577ms
z_uncommit_meta	False	36 min 16 sec 178 ms	60.48 gb	56.14 gb	357.29 mb/sec	99.992	0.94243115	8.948	179ms	10m 33s 505ms
g1_expconc	False	37 min 26 sec 378 ms	63 gb	55.87 gb	507.5 mb/sec	99.537	44.482018	203.364	10s 409ms	5m 24s 493ms
g1_compressed_heapwaste	False	38 min 41 sec 580 ms	62.95 gb	52.67 gb	428.07 mb/sec	99.512	46.974335	230.0	11s 321ms	4m 56s 280ms

Figure 5.11: A selection of 2 possible configurations from each collector. Processing node.

Z_uncommit_meta had the lowest average allocation rate of 357.29mb/sec. The z_moreConc execution did experience metadata memory space issues. The z_meta configuration from figure 5.9, compared to z_default tells of reduced measured duration time and a small cost of maximum pause time to fix the metadata occupancy problem of z_moreConc. In terms of latency, average pause time, maximum pause time and total STW, Shenandoah produced more than 93% better than G1GC in all three categories, and ZGC produced more than 95% better values than G1GC in all three categories. The calculations are based on the configuration with the shortest M-duration for each garbage collector (i.e z_uncommit_meta, sh_alloc_spike, and g1_expconc).

Putting all collectors against each other for the process node results, it can be observed that the ZGC spent about twice as much time in a concurrent GC working state compared to its opponents. G1GC reached the highest allocation rates, followed by Shenandoah GC and lastly ZGC. ZGC utilizes more metadata memory space and takes more hits of GC events triggered by the metadata occupancy limit being reached. The throughput percentage values between the collectors lie within similar bounds. The average pause time, maximum pause time, and total STW time was significantly higher for G1GC configurations than for Shenandoah GC and ZGC. Doubled or worse. Between ZGC and Shenandoah GC, ZGZ produced moderately lower values.

5.3 Query node

5.3.1 Default configurations

The results of the default configurations of G1GC, Shenandoah and ZGC run on the query node application are showed in figure 5.12. The default configurations are extended with *-Xmx256G*, allowing larger heap sizes. ZGCs z_default was not the exact out-of-the-box configuration, but was run with the

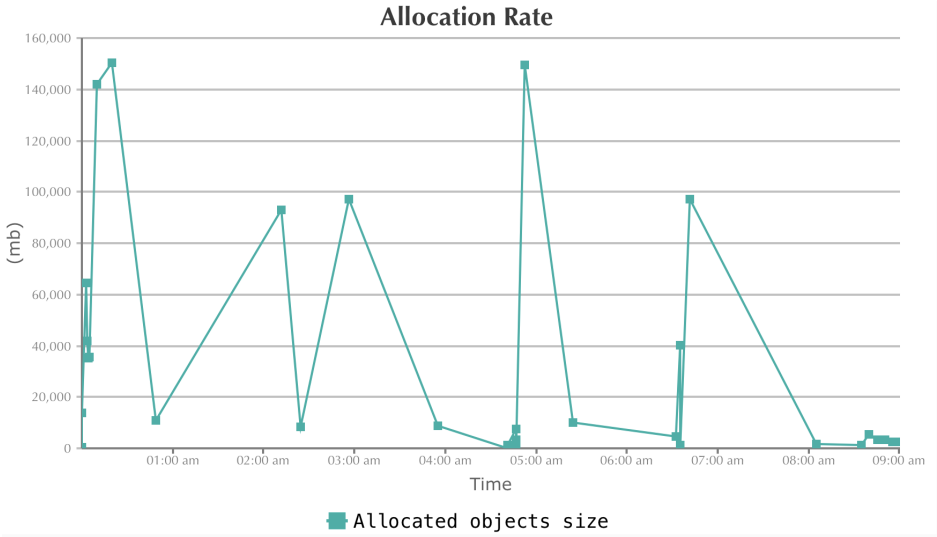
additional JVM option of *-XX:ConcGCThreads=10*, increasing number of GC threads from 5 to 10, as mentioned in the method chapter.

G1GC ran through the query in a measured duration time of 49min 27sec 943ms, Shenandoah ran for 9hours 1min 23 seconds, and ZGC ran for 4hours 34min 18sec. None of the configurations experienced allocation stalls, however sh_default and z_default suffered form long-running GC events (and pause times) which make a heavy footprint in the execution time of the query (measured duration). Z_default additionally suffered form metadata occupancy inadequacy. These are the problems referred to in the problem column in figure 5.12, detected by GC-Easy. The long execution time is closely connected to the extremely low average allocation rates of Shenandoah and ZGC, 31.98mb/sec and 63.34mb/sec respectively, compared to G1GC with an average rate of 462.34mb/sec. The allocation rate of G1GC is more reasonable given the results of the processing node, although that is an entirely different application. Another observation is that g1_default barely makes use of the allowed jvm heap memory of 256G.

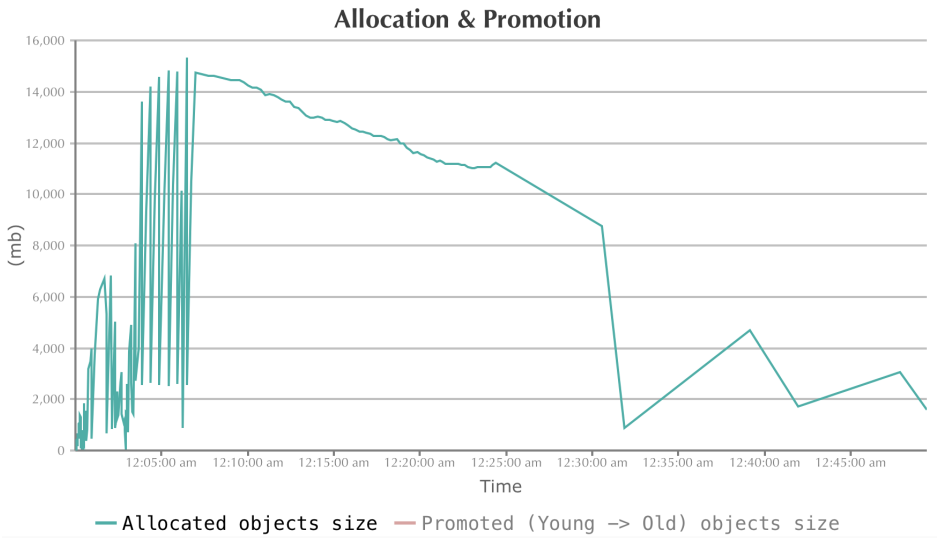
Label	Problem	M-duration	jvmHeap alloc	jvmHeap peak	AvgAllocRate	Throughput%	avgPT (ms)	maxPT (ms)	Total STW	Total Conc
g1_default	False	49 min 27 sec 943 ms	39.56 gb	34.4 gb	462.34 mb/sec	99.549	59.50272	300.0	13s 388ms	1m 16s 458ms
sh_default	True	9 hrs 1 min 23 sec	246.41 gb	245.09 gb	31.98 mb/sec	96.268	12371.154	686110.94	20m 12s 373ms	7h 4m 50s
z_default	True	4 hrs 34 min 18 sec	232.96 gb	181.55 gb	63.34 mb/sec	99.844	215.19756	14413.588	25s 609ms	2h 24m 37s

Figure 5.12: Query node GC-Easy results of default configurations.

Graphs in figures 5.13 - 5.17, further showcase the GC behaviour throughout the execution of the query. G1_default utilized the heap with consistency whilst sh_default show a lot of ups and downs, filling the heap to the max and then cleaning up with long-running GCs. Z_default behave similarly to sh_default with a swinging heap utilization. Note the difference in scale in the y-axis (length and sec/ms).



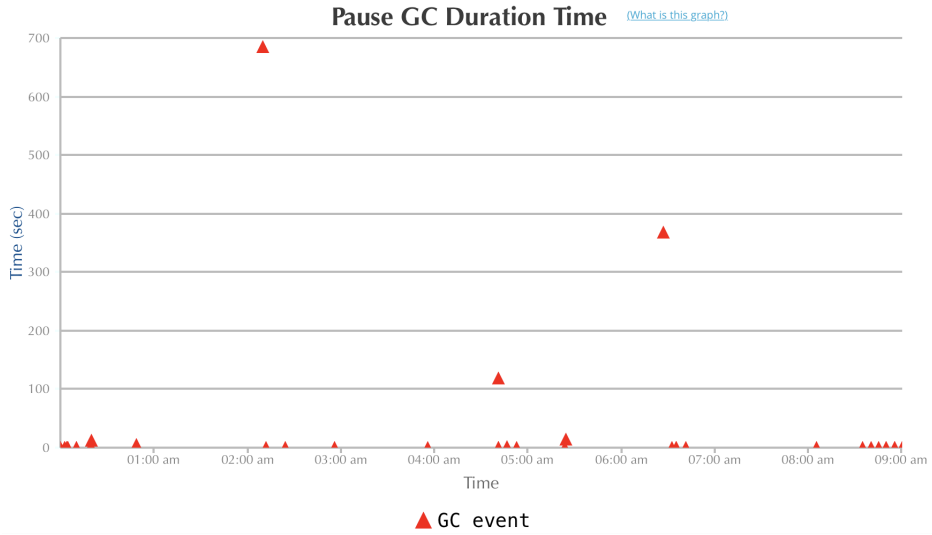
(a) Shenandoah



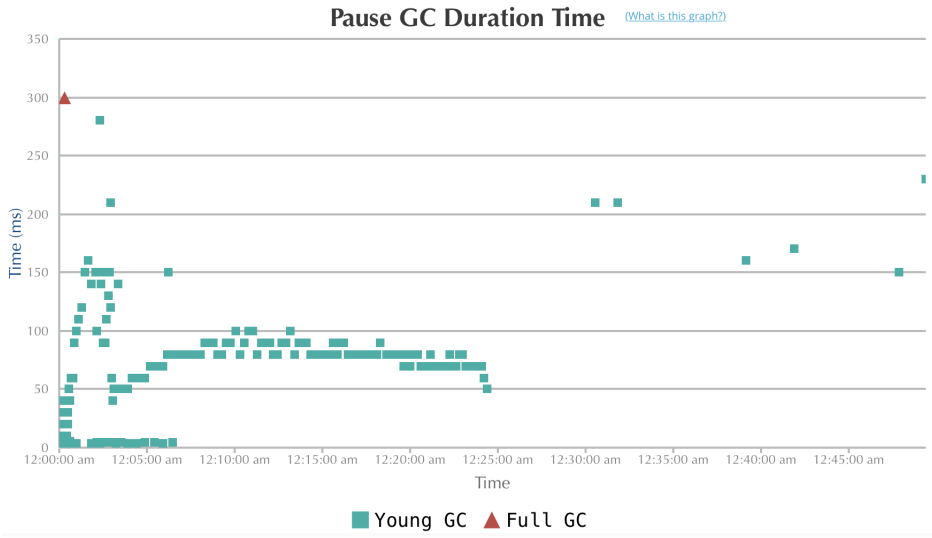
(b) G1

(c) ZGC

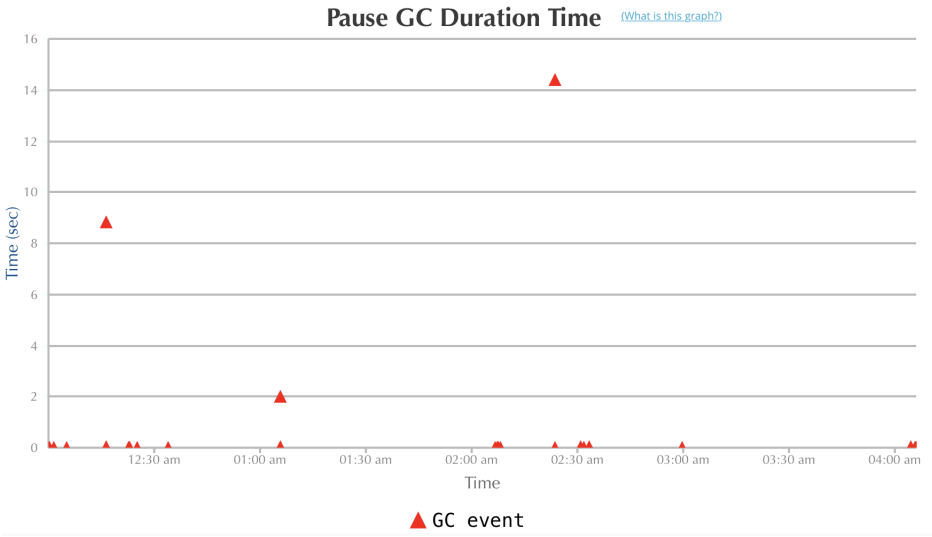
Figure 5.13: Query node default configurations allocation rates, generated by GC-Easy.



(a) Shenandoah

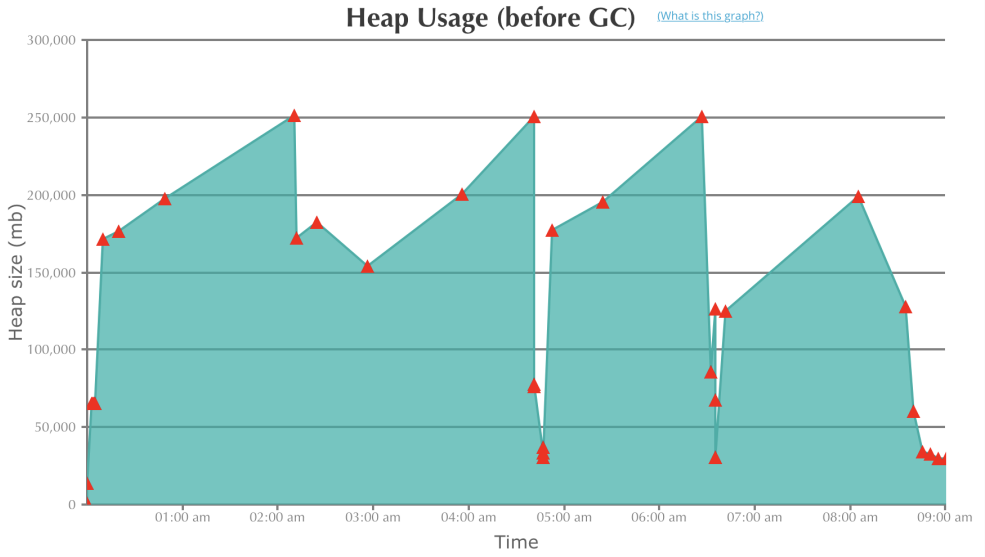


(b) G1

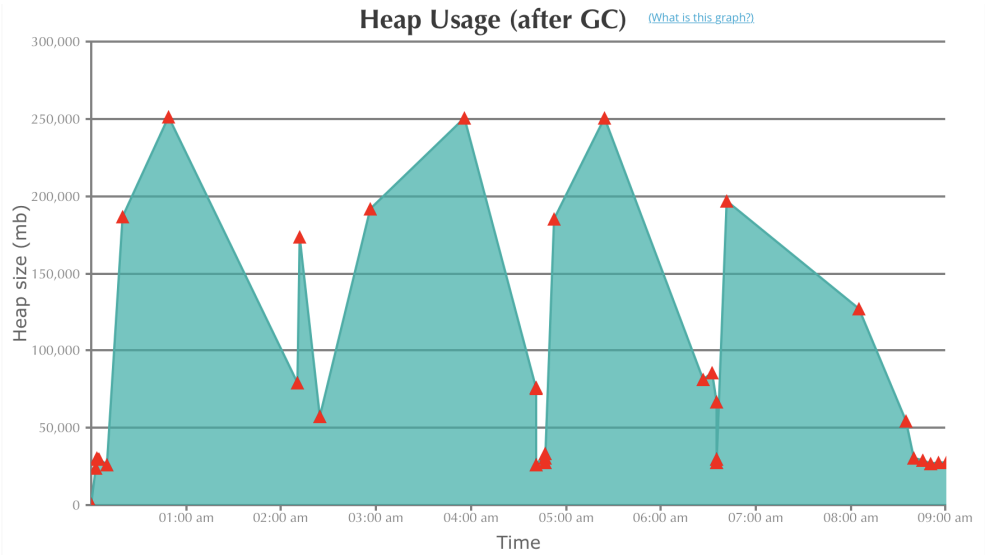


(c) ZGC

Figure 5.14: Query node default configurations GC pause duration times, generated by GC-Easy.

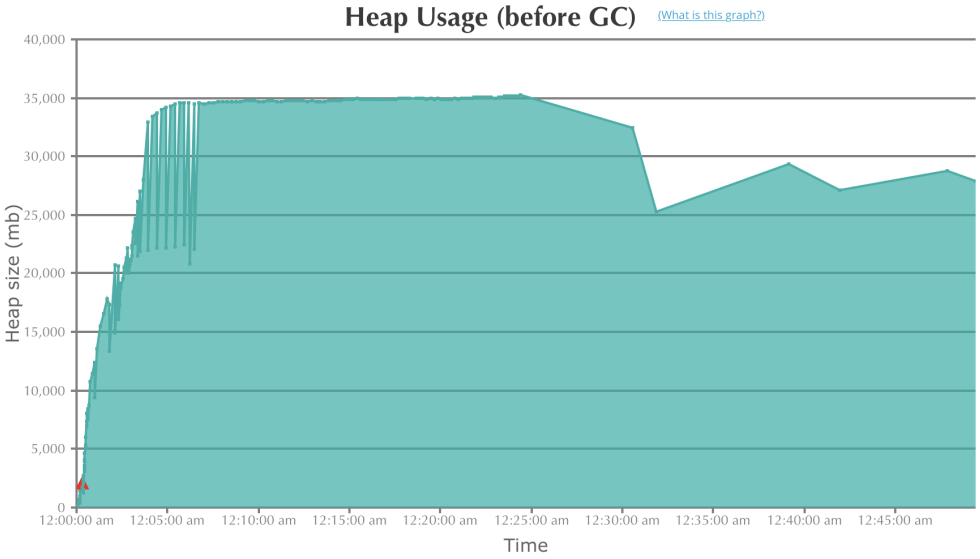


(a) Shenandoah before.

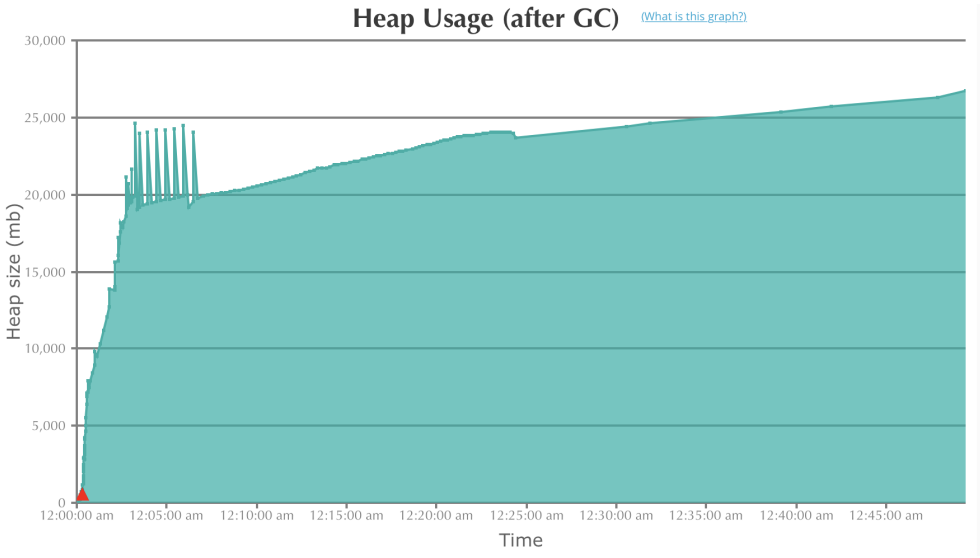


(b) Shenandoah after.

Figure 5.15: Shenandoah GC - Query node Heap Usage Before/After GC event, generated by GC-Easy.

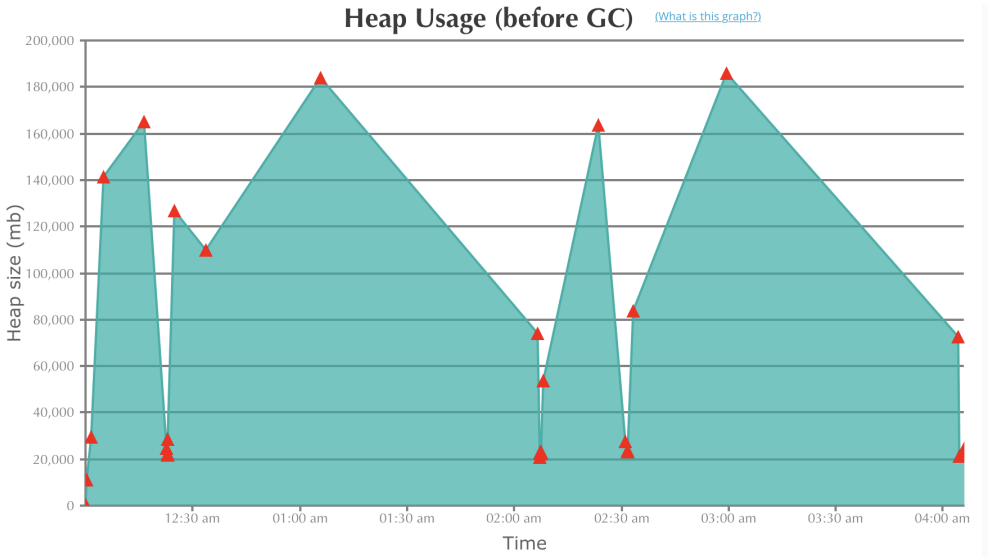


(a) G1 before.

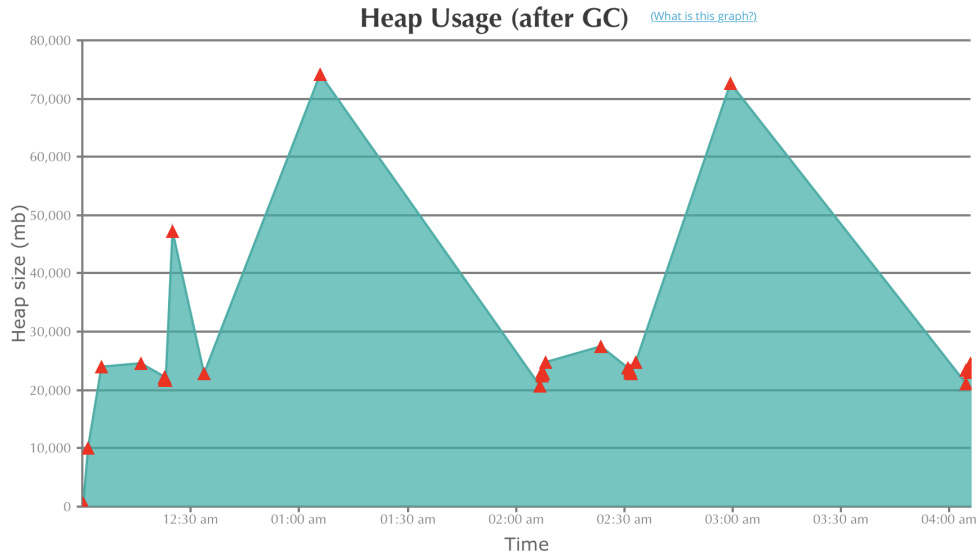


(b) G1 after.

Figure 5.16: G1GC - Query node Heap Usage Before/After GC event, generated by GC-Easy.



(a) ZGC before.



(b) ZGC after.

Figure 5.17: ZGC - Query node Heap Usage Before/After GC event, generated by GC-Easy.

JClarity Censum provided no additional information regarding the performance of sh_default or z_default. The tool did not conduct any deep analysis of the Shenandoah and ZGC logs. For g1_default high kernel times were detected. The kernel time exceeded the user time once, and the GC threads collected an abnormal amount of kernel time 67times.

For the default configurations, Shenandoah allocated 18 threads in parallel and 9 threads to be concurrent workers. ZGC allocated 22 threads in parallel and 5 threads concurrent.

5.3.2 Modified configurations

The result of the modified configuration tests of G1GC run on the query node are shown in figure 5.18. There is a large variation of allocated heap memory and peak usage between the configurations and none utilized the full allowed capacity of 256GB, specified with -Xmx256G. G1_current utilized the most memory, peak 60.24gb. The data of g1_current does however not show any benefits in throughput nor latency. On the contrary, the throughput percentage was reduced compared to g1_default (99.358 vs 99.549), along with the highest average and maximum pause times in the table.

Most allocation rates are within the interval of 460mb/sec to 480mb/sec. Anomalies are g1_throughput with 400.4mb/sec and g1_compressed_heapwaste with 496.75mb/sec. G1_throughput additionally had the lowest peak of utilized JVM heap memory and ran for the longest measured duration time of 56min 52sec 361ms. The throughput percentage of g1_period is comparably low and GC-Easy also detected a memory leak issue of the g1_period run. Memory leaks are serious problems that may cause out-of-memory errors, the JVM to freeze, poor application responsiveness, and/or high CPU consumption. The other configuration which experienced problems was the g1_newsize configuration. GC-Easy spots that sys-time time is greater than user-time. Sys-time is the amount of CPU time spent in the kernel within a CPU event, and user-time is the CPU time spent in user-mode outside the kernel, within a CPU event. Multiple occurrences of sys-time being greater than user-time may indicate OS problems, VM related problems, memory constraint issues, or disk I/O pressure.

In terms of throughput, the highest percentage values were delivered by g1_init and g1_throughput (99.612, 99.611). Although the measured execution time duration was shorter for more than half of the other configurations, compared to the two. G1_compressed, g1_expconc, g1_compressed_heapwaste, and g1_resere are able to better balance a reduced measure duration time with a

moderate increase of throughput percentage. These configurations pay for the speed with extended pause times. G1_compressed and g1_expconc both include the *-XX:UseCompressedOops* option. G1_resere executed in the shortest measured duration time of 48min 19sec 708ms.

G1GC

Label	Problem	M-duration	jvmHeap alloc	jvmHeap peak	AvgAllocRate	Throughput%	avgPT (ms)	maxPT (ms)	Total STW	Total Conc
g1_newsize	True	48 min 43 sec 406 ms	39.75 gb	35.6 gb	477.75 mb/sec	99.594	90.50072	1030.0	11s 856ms	1m 15s 719ms
g1_compressed	False	48 min 29 sec 135 ms	45.66 gb	39.49 gb	473.9 mb/sec	99.566	68.631836	270.0	12s 628ms	47s 140ms
g1_expconc	False	48 min 41 sec 459 ms	45.34 gb	39.23 gb	470.76 mb/sec	99.586	66.41548	220.0	12s 88ms	46s 361ms
g1_lowHeapWaste	False	48 min 47 sec 445 ms	38.91 gb	33.86 gb	467.5 mb/sec	99.56	55.963924	270.0	12s 872ms	1m 26s 64ms
g1_init	False	49 min 7 sec 420 ms	43.97 gb	38.26 gb	467.54 mb/sec	99.612	73.70085	460.0	11s 424ms	21s 559ms
g1_updatePT	False	48 min 57 sec 327 ms	38.41 gb	33.41 gb	466.13 mb/sec	99.575	57.02001	480.0	12s 487ms	1m 33s 338ms
g1_meta_pure	False	49 min 3 sec 653 ms	43.22 gb	37.56 gb	468.01 mb/sec	99.594	79.63	470.0	11s 994ms	33s 912ms
g1_string	False	51 min 33 sec 814 ms	40.91 gb	35.46 gb	443.15 mb/sec	99.543	98.18086	530.0	14s 138ms	14s 348ms
g1_parallel	False	49 min 16 sec 450 ms	43.78 gb	37.95 gb	464.67 mb/sec	99.607	71.24765	410.0	11s 613ms	34s 504ms
g1_period	True	49 min 55 sec 204 ms	61.97 gb	26.89 gb	468.44 mb/sec	98.672	25.232857	240.0	39s 767ms	43m 2s 575ms
g1_resere	False	48 min 19 sec 708 ms	47.81 gb	39.08 gb	472.34 mb/sec	99.554	67.41087	310.0	12s 943ms	43s 865ms
g1_current	False	49 min 37 sec 572 ms	70.84 gb	60.24 gb	487.38 mb/sec	99.358	360.84802	1090.0	19s 125ms	80.5ms
g1_default	False	49 min 27 sec 943 ms	39.56 gb	34.4 gb	462.34 mb/sec	99.549	59.50272	300.0	13s 388ms	1m 16s 458ms
g1_throughput	False	56 min 52 sec 361 ms	36.69 gb	32.12 gb	400.4 mb/sec	99.611	61.8078	290.0	13s 289ms	30s 88ms
g1_latency	False	49 min 3 sec 651 ms	43.97 gb	38.06 gb	469.14 mb/sec	99.609	69.29054	370.0	11s 502ms	34s 526ms
g1_xms	True	49 min 46 sec 411 ms	128.0 gb	95.74 gb	461.35 mb/sec	99.692	135.10156	470.0	9s 187ms	365ms
g1_compressed_heapwaste	False	48 min 46 sec 952 ms	41.03 gb	35.67 gb	496.75 mb/sec	99.532	60.399937	400.0	13s 711ms	1m 24s 230ms

Figure 5.18: Query node GC-Easy results of the modified G1GC configurations.

Figure 5.19 display data results of the Shenandoah modified configurations run on the query node. It is clear that the out-of-the-box configuration sh_default measured the longest running duration of over 9 hours. Sh_compact10 and sh_compact20, which both ran with the compact heuristic, performed notably better than its competitors both in terms of throughput and latency. They both had a measured duration time of under 1 hour, an average time under 1.2sec, maximum pause time under 20ms and throughout above 99.98%. This compared to measured duration times above 3 hours, throughput below 98.2% and average pause times above 3sec. For the compact heuristics, the option of setting concurrent GC threads was used. Sh_compact10 was allowed 10 concurrent GC threads and sh_compact20 20. The default allocated GC threads by Shenandoah were 5 for the query node. In terms of memory, most configurations allocated and used around 245gb of the allowed 256gb, except for the configurations with the compact heuristic who utilized less than 20% of the allowed space, much like the behavior of the G1GC query configurations.

GC-Easy detected possible problems for all configuration runs except sh_compact10, see figure 5.1. All but compact and sh_compact10 were suffering from long-running GC events which is clear when looking at the measured duration data in figure 5.19. Sh_static_freeT additionally experienced

Shenandoah

Label	Problem	M-duration	jvmHeap alloc	jvmHeap peak	AvgAllocRate	Throughput%	avgPT (ms)	maxPT (ms)	Total STW	Total Conc
sh_alloc_spike	True	4 hrs 25 min 24 sec	245 gb	244.88 gb	73.46 mb/sec	97.103	4118.4683	301638.8	7m 41s 268ms	2h 52m 23s
sh_compact10	False	52 min 13 sec 431 ms	50.94 gb	43.55 gb	416.05 mb/sec	99.984	1.1926584	18.62	496ms	4m 14s 483ms
sh_static_freeT	True	5 hrs 28 min 41 sec	245.38 gb	244 gb	59.94 mb/sec	98.141	4215.006	363169.16	6m 6s 706ms	3h 59m 22s
sh_string	True	5 hrs 49 min 52 sec	244.91 gb	244.91 gb	53.83 mb/sec	95.47	9906.32	612115.9	15m 51s 7ms	3h 46m 10s
sh_parallel	True	4 hrs 44 min 11 sec	244.94 gb	244.94 gb	69.65 mb/sec	93.936	10770.584	566475.4	17m 13s 976ms	3h 7m 21s
sh_default	True	9 hrs 1 min 23 sec	246.41 gb	245.09 gb	31.98 mb/sec	96.268	12371.154	686110.94	20m 12s 373ms	7h 4m 50s
sh_compact20	True	52 min 17 sec 918 ms	47.62 gb	43.72 gb	427.79 mb/sec	99.986	1.046493	17.594	440ms	2m 26s 110ms
sh_static	True	9 hrs 51 min 9 sec	246.81 gb	245.56 gb	28.95 mb/sec	95.446	20710.355	618574.56	26m 55s 408ms	7h 50m 25s
sh_meta	True	7 hrs 17 min 18 sec	244.84 gb	244.84 gb	41.88 mb/sec	91.848	28142.574	832559.4	35m 38s 836ms	4h 55m 18s
sh_xms	True	5 hrs 20 min 19 sec	244.88 gb	244.75 gb	62.97 mb/sec	97.678	4056.8904	411061.78	7m 26s 258ms	3h 24m 48s
sh_static_parallel	True	9 hrs 25 min 9 sec	256 gb	245.16 gb	32.69 mb/sec	85.825	73949.74	2320930.5	1h 20m 6s	6h 51m 47s
sh_compressed	True	3 hrs 23 min 56 sec	244.81 gb	244.81 gb	104 mb/sec	96.9	3870.7012	188196.14	6m 19s 320ms	1h 55m 54s
sh_moreConc	True	3 hrs 46 min 55 sec	245.03 gb	245.03 gb	93.83 mb/sec	97.709	3058.6719	156417.38	5m 11s 985ms	2h 12m 25s

Figure 5.19: Query node GC-Easy results of the modified Shenandoah GC configurations.

metadata space occupancy issues as the metadata threshold was reached quite often and triggering GC events. Sh_parallel spent more time on GC than what GC-Easy considers feasible. Namely more than 5% of the total time, which is displayed in the throughput percentage value of sh_parallel. Too much time spent on GC activity degrades response time and consumes CPU. For the sh_compact run, configured with 20 concurrent GC threads, GC-Easy detected a memory leak, which is a severy problem.

Sh_alloc_spike, sh_static_freeT, sh_string, sh_parallel, sh_default, sh_static, sh_meta, sh_xms, sh_static_parallel, sh_compressed and sh_moreConc do not meet the query node performance requirement of maximum pause times under 10-20sec.

Graphs in figures 5.20, 5.21, 5.22, and 5.23 display further information about the sh_compact10 execution on the query node (the quickest run). The graphs show that sh_compact10 was more stable than the default Shenandoah configuration sh_default. A larger number of pause times occurred, although with shorter time durations. The heap usage before and after GC, along with the allocation rate, is a lot smoother than in the default mode (Sh_defaul), without the ups and downs except for a dramatic drop at half-time.

Configuration	Long running GC	Metadata occupancy	Memory leak	Too much time in GC
sh_alloc_spike	X			
sh_compact10				
sh_static_freeT	X	X		
sh_string	X			
sh_parallel	X			X
sh_default	X			
sh_compact20			X	
sh_static	X	X		
sh_meta	X			X
sh_xms	X		X	
sh_static_parallel	X			X
sh_compressed	X	X		
sh_moreConc	X			

Table 5.1: Problems detected by GC-Easy for Shenandoah query node configurations.

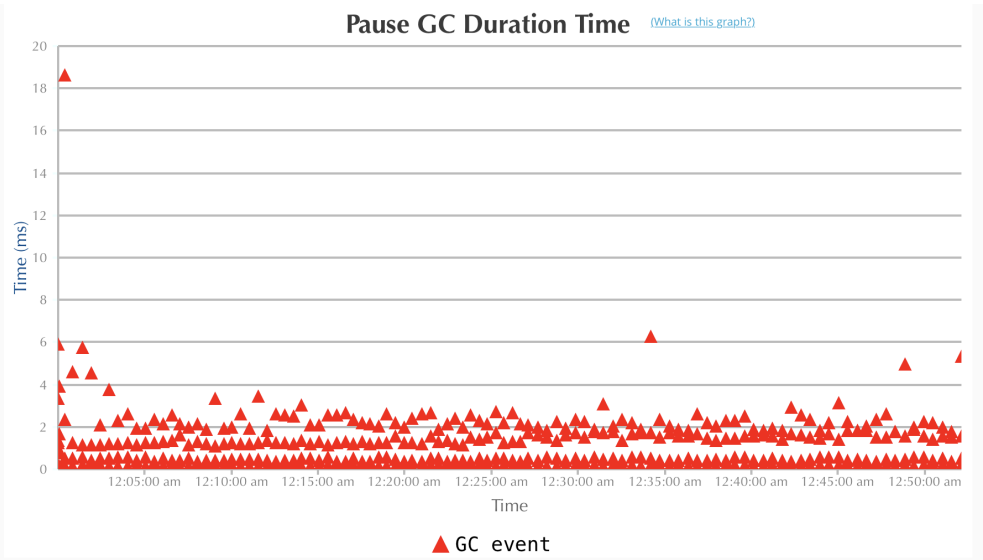


Figure 5.20: Sh_compact10 GC pause duration times, generated by GC-Easy

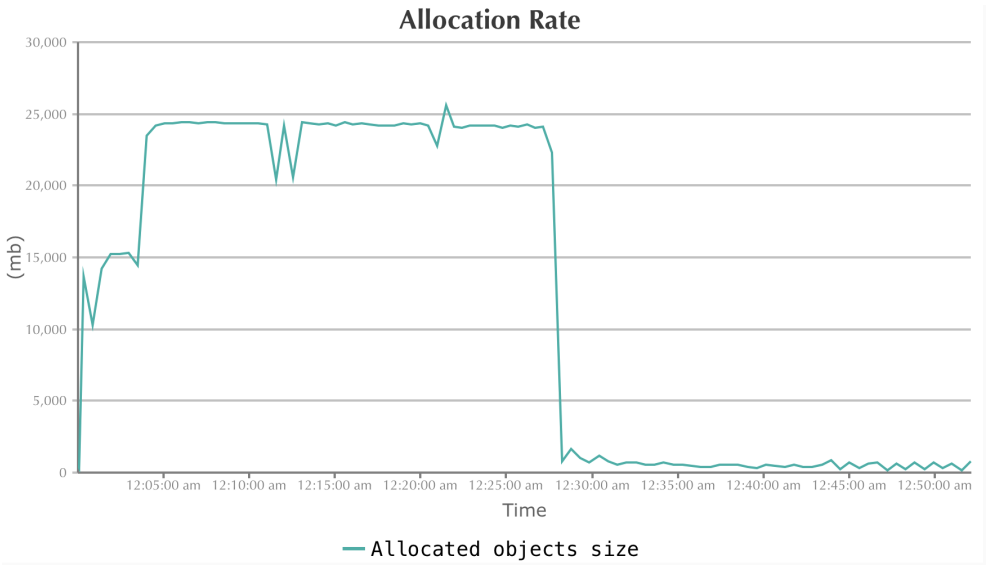


Figure 5.21: Sh_compact10 allocation rates, generated by GC-Easy

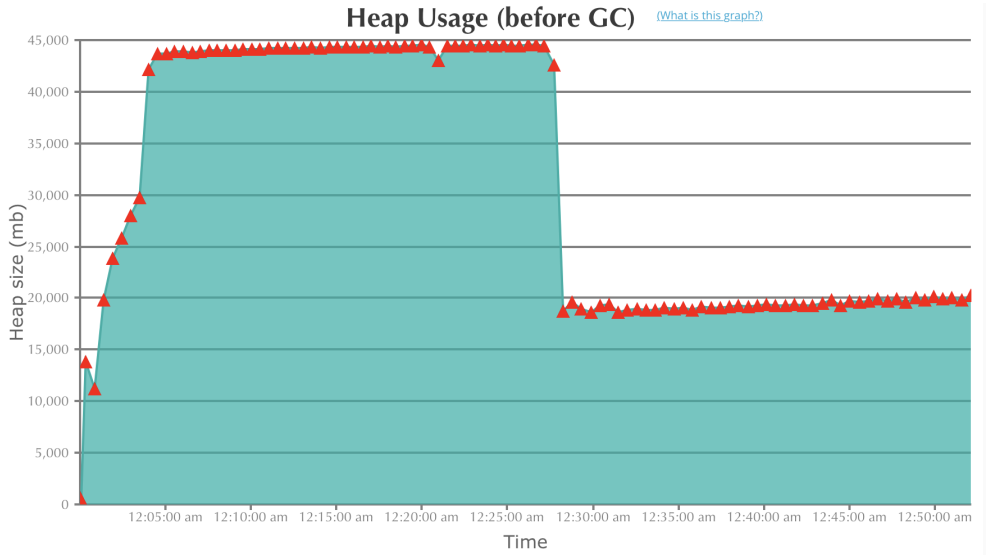


Figure 5.22: Sh_compact10 heap usage before GC, generated by GC-Easy

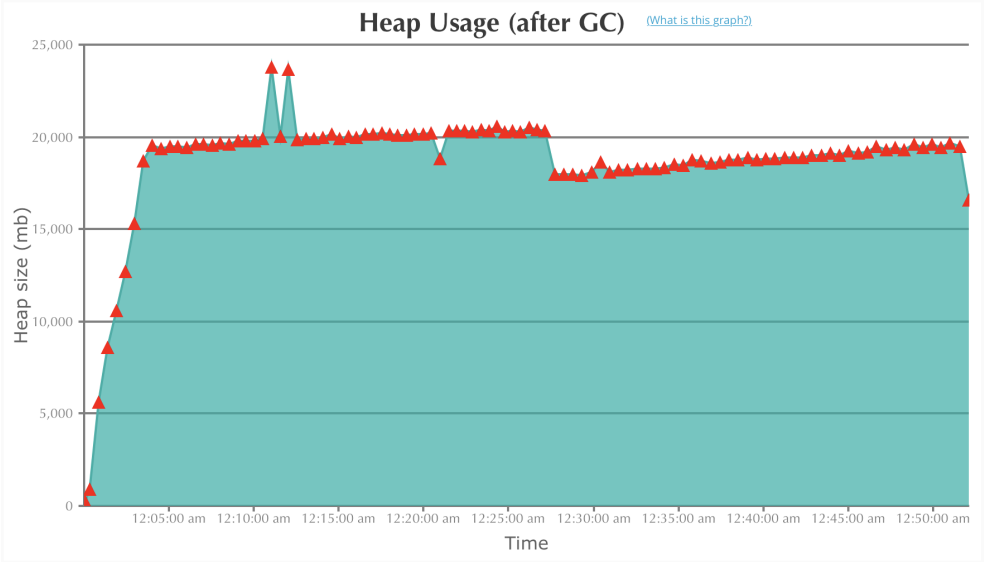


Figure 5.23: Sh_compact10 heap usage after GC, generated by GC-Easy

ZGC

Label	Problem	M-duration	jvmHeap alloc	jvmHeap peak	AvgAllocRate	Throughput%	avgPT (ms)	maxPT (ms)	Total STW	Total Conc
z_parallel	True	3 hrs 24 min 33 sec	225.08 gb	177.23 gb	87.66 mb/sec	99.716	288.28494	33725.41	34s 882ms	1h 48m 41s
z_defaultTrans	True	5 hrs 26 min 27 sec	238.24 gb	184.07 gb	55.84 mb/sec	99.994	9.17359	937.779	1s 165ms	3h 31m 16s
z_spike	True	2 hrs 57 min 15 sec	207.79 gb	172.3 gb	112.4 mb/sec	99.998	1.8049259	32.703	220ms	1h 25m 37s
z_uncommit	True	3 hrs 34 min 11 sec	214.71 gb	179.65 gb	88.75 mb/sec	99.588	448.32178	52666.598	52s 902ms	1h 49m 53s
z_proactive	True	4 hrs 10 min 44 sec	223.99 gb	183.89 gb	75.81 mb/sec	99.6	466.93393	59310.723	1m 234s	1h 51m 10s
z_interval	True	49 min 5 sec 725 ms	58.71 gb	55.3 gb	421.63 mb/sec	99.99	0.8681516	11.174	302ms	5m 33s 312ms
z_default	True	4 hrs 34 min 18 sec	232.96 gb	181.55 gb	63.34 mb/sec	99.844	215.19756	14413.588	25s 609ms	2h 24m 37s
z_string	True	3 hrs 59 min 1 sec	219.55 gb	182.68 gb	72.91 mb/sec	99.774	299.98773	30319.445	32s 399ms	2h 1m 44s
z_meta	True	3 hrs 31 min 37 sec	227.46 gb	185.93 gb	97.29 mb/sec	99.899	133.97224	6060.167	12s 861ms	1h 38m 42s
z_xms	True	4 hrs 32 min 28 sec	241.19 gb	198.63 gb	67.44 mb/sec	99.984	21.208643	2393.333	2s 609ms	1h 37m 14s
z_moreConc	True	3 hrs 23 min 26 sec	194.29 gb	175.6 gb	98.78 mb/sec	99.582	486.38126	48541.82	51s 70ms	1h 22m 54s

Figure 5.24: Query node GC-Easy results of the modified ZGC configurations.

In figure 5.24 the results of ZGC tuned configurations are displayed with key metric values. All configurations suffered from problems detected by the GC-Easy analytic tool, see table 5.2. The problems were long-running GC events which are unfavorable for GC performance and/or metadata occupancy issues where the threshold is reached too often. The configurations which suffered from metadata occupancy problems all allocated metadata space of 115mb (default) while the configurations which did not experience these problems were all specifically configured with a 256mb metadata space allocation and additional expansion and free ratio instructions.

The configuration with the shortest measured duration time was z_interval which lasted for 49min 5sec 725ms, followed by z_spike that ran for 2hours 57min and 15sec. This compared to z_default execution of 4hours 34min 18sec. Z_spike ran with an allocation spike tolerance factor of 4 (estimated rate correction factor). The z_defaultTrans which allowed for use of transparent huge pages (*-XX:UseTransparentHugePages*) took the longest to finish the query execution with an M-duration of 5hours 26min 27sec. All configurations except for z_interval and z_spike had an average allocation rate below 100mb/sec. Z_interval had the highest average allocation rate of 421.63mb/sec and Z_spike allocated about 112.4mb/sec.

Z_spike performed the highest throughput percentage value of 99.998%. Z_moreConc achieved the bottom throughput percent of 99.582%. The peak JVM heap usage for the configurations lies within the range 170-200gb, with

Configuration	**Long running GC**	**Metadata occupancy**
z_parallel	X	
z_defaultTrans		X
z_spike		X
z_uncommit	X	X
z_proactive	X	X
z_interval		X
z_default	X	X
z_string	X	
z_meta		
z_xms		X
z_moreConc	X	X

Table 5.2: Problems detected by GC-Easy for ZGC query node configurations.

the exception of z_interval, which is far more than the utilization range of the G1GC configurations but notably less than most of the Shenandoah configurations range. Z_spike utilized the least memory amongst the ZGC configurations. Z_interval ran immensely faster than the other configurations, had a higher allocation rate of 421.63mb/sec, used significantly less memory (55.3gb peak) and executed with great values in terms of latency. However, the configuration suffered from metadata occupancy irritations. Looking at latency, average and maximum pause time and total STW, z_interval ranks highest (0.87ms, 11.174ms, 302ms) followed by z_spike (1.8ms, 32.7ms, 220ms), z_defaultTrans (9.2ms, 937.8ms, 1s 165ms) and z_xms (21.2ms, 2393ms, 2s 609ms). All of which performed significantly better than the rest which produced average pause times in the range of 130-500ms, maximum pause times above 6sec, and total accumulated STW time of 12s-1min.

Z_defaultTrans put down notably more time in concurrent phases compared to the other ZGC configurations. Configurations z_parallel, z_uncommit, z_proactive, z_string, and z_moreConc do not achieve the query node requirement of maximum pause times below 10-20sec.

Graphs in figures 5.25, 5.26, 5.27, and 5.28 display further information about the z_interval configuration run (the shortest M-duration execution). Compared to ZGC default (z_default), z_interval had a lot more pause times, as they were initiated by time interval, although the length of the pause times were notably shorter. The allocation rate and heap usage were also more balanced for z_interval than z_default. Two levels rather than spikes up and down.

In figure 5.29 a selection of two possible configurations for each collector has been put together to better visualize their compared performances. In terms of measured duration time to execute the test query, the z_spike option is far behind and simply cannot compete. Shenandoah configuration sh_compact20 had memory issues regarding sys time being greater than user time, which is undesirable. Shenandoah sh_compact10 had a moderately higher peak heap size and longer measured duration compared to the G1GC configurations but produced much better throughput percentage and prominently lower pause time values. Z_interval produced even lower pause times than sh_compact10, slightly higher throughput percentage, and ran for about 49min which puts it between sh_compact10 and the G1GC configurations in terms of measured execution time. However, GC-Easy acknowledged metadata memory space insufficiencies for z_interval.

In terms of latency, average pause time, maximum pause time and total STW, Shenandoah GC and ZGC produced more than 90% better values than

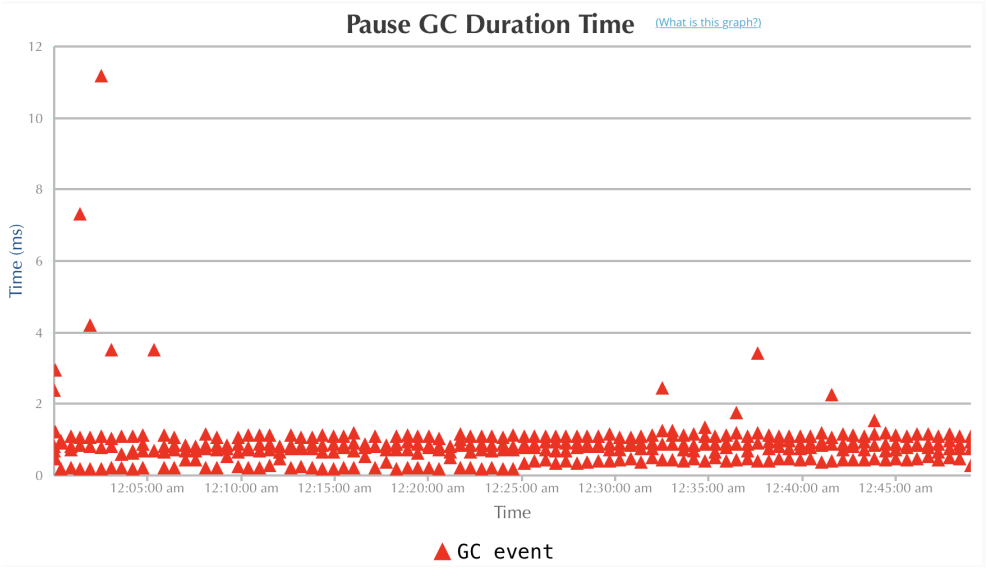


Figure 5.25: Z_interval GC pause duration times, generated by GC-Easy

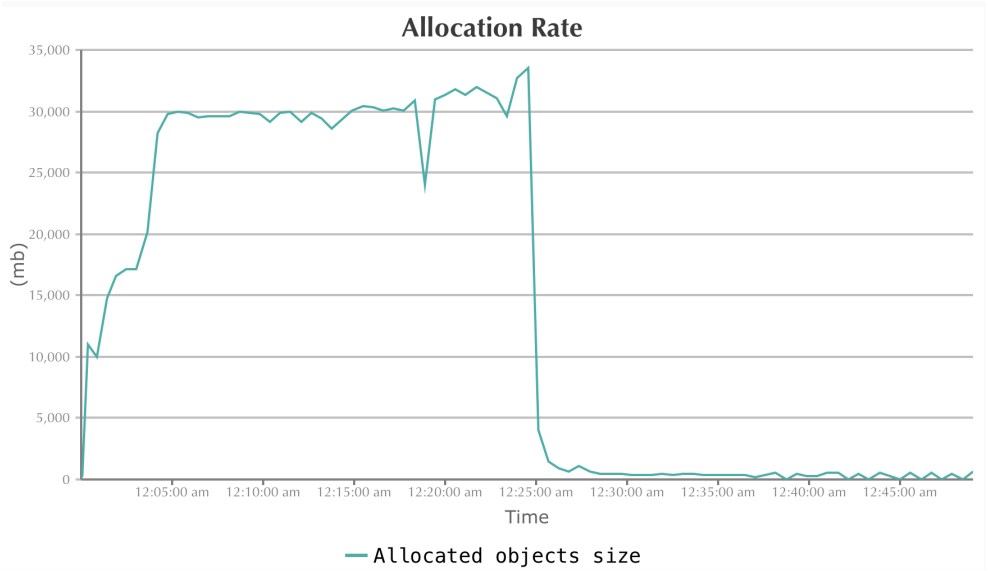


Figure 5.26: Z_interval allocation rates, generated by GC-Easy

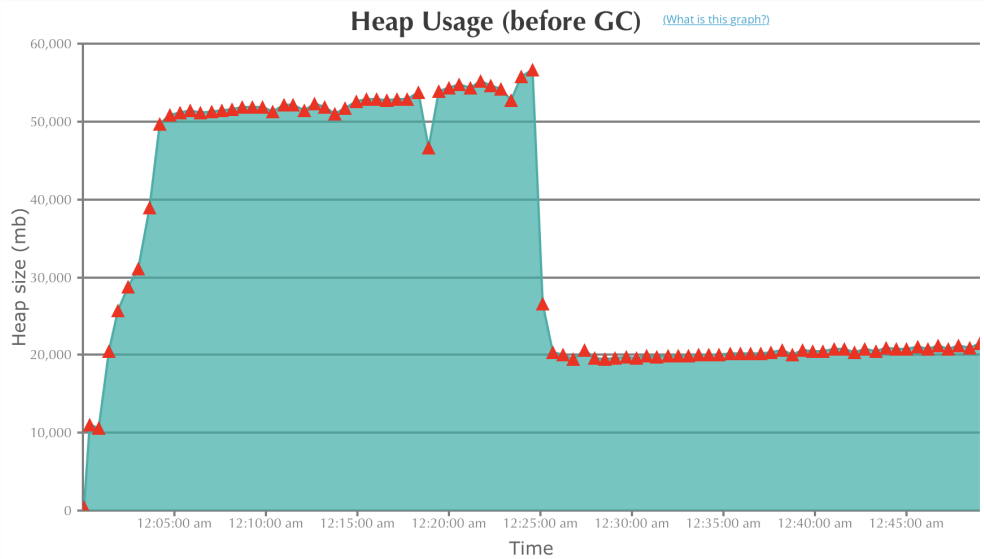


Figure 5.27: Z_interval heap usage before GC, generated by GC-Easy

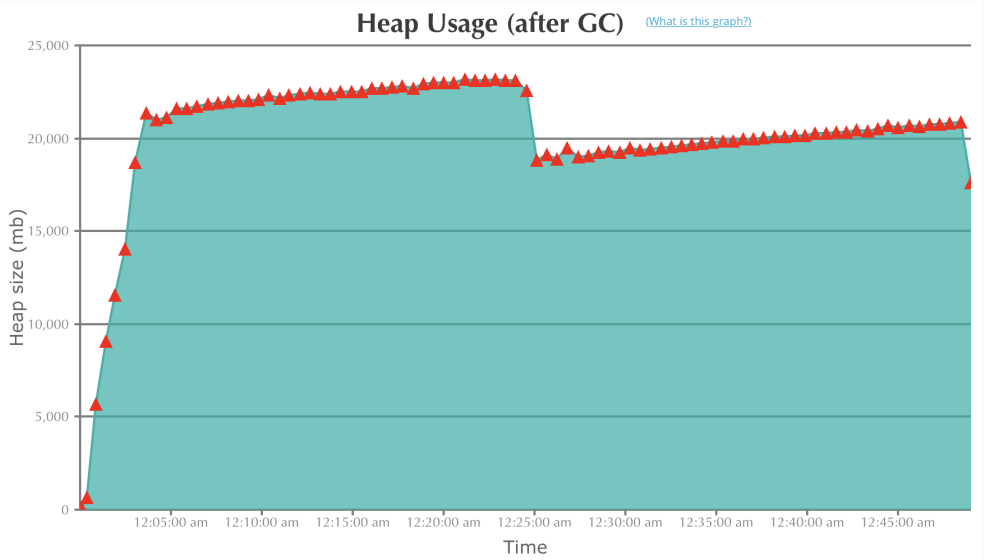


Figure 5.28: Z_interval heap usage after GC, generated by GC-Easy

G1GC in all three categories (Shenandoah >91%, ZGC >94%). The calculations are based on the configuration with the shortest M-duration for each garbage collector (i.e z_interval, sh_compact10, and g1_expconc).

Label	Problem	M-duration	jvmHeap alloc	jvmHeap peak	AvgAllocRate	Throughput%	avgPT (ms)	maxPT (ms)	Total STW	Total Conc
z_interval	True	49 min 5 sec 725 ms	58.71 gb	55.3 gb	421.63 mb/sec	99.99	0.8681516	11.174	302ms	5m 33s 312ms
z_spike	True	2 hrs 57 min 15 sec	207.79 gb	172.3 gb	112.4 mb/sec	99.998	1.8049259	32.703	220ms	1h 25m 37s
sh_compact10	False	52 min 13 sec 431 ms	50.94 gb	43.55 gb	416.05 mb/sec	99.984	1.1926584	18.62	496ms	4m 14s 483ms
sh_compact20	True	52 min 17 sec 918 ms	47.62 gb	43.72 gb	427.79 mb/sec	99.986	1.046493	17.594	440ms	2m 26s 110ms
g1_lowHeapWaste	False	48 min 47 sec 445 ms	38.91 gb	33.86 gb	467.5 mb/sec	99.56	55.963924	270.0	12s 872ms	1m 26s 64ms
g1_expconc	False	48 min 41 sec 459 ms	45.34 gb	39.23 gb	470.76 mb/sec	99.586	66.41548	220.0	12s 88ms	46s 361ms

Figure 5.29: A selection of 2 possible configurations from each collector. Query node.

In an overlooking perspective comparing the performance of the three collectors on the query node application, the G1GC collector could perform the query in significantly less time than the average ZGC and Shenandoah configuration. ZGC and Shenandoah generally made use of the available heap space when G1GC did not. ZGC ca 200gb (with one exception of ca 55gb), Shenandoah ca 245gb (with two exceptions of ca 45gb) and G1GC ca 35gb. Shenandoah and ZGC were more prone to memory issues detected by GC-Easy some of which fatal and some minor fixable ones. None of the collector configurations fell into allocation stalls. ZGC produced the lowest pause time values, followed by Shenandoah and lastly G1GC.

6 Discussion

An interesting factor to mention when discussing the results is the ZGC usage of multiple memory mappings which required the maximum memory mappings per process in the environment to be increased. Otherwise, it was at risk for fatal errors of failing to map memory.

In a presentation by Per Lidén and Stefan Karlsson, "The Z Garbage Collector", ZGC is presented to perform equal or better throughput than G1GC on the SPECJbb2015 benchmarks [3]. The results of this study, processing node and query node, confirm this with throughput percentages of ZGC configurations being higher than for G1GC. However, G1GC and ZGC seem to perform equally (in general) in terms of measured duration time (M-duration), which is another throughput metric. Referring to the metrics validity discussion in section 6.4, the actuality of it is that G1GC is more throughput favorable. This would then contradict the SPECJbb2015 results in the presentation. The DaCapo results of this study conform to the same throughput percentage pattern as the process and query node, but not necessarily in regards to the M-duration pattern. This could be due to the much smaller scale of the DaCapo applications.

The same study, "The Z Garbage Collector" [3], also presented latency measurements on SPECJbb2015 which showed that ZGC performed significantly better pause times. This is what we can see in this report as well. For the processing node, ZGC performed tremendously better in terms of latency than G1GC. The same goes for the DaCapo benchmarks. For the query node, this is true only for some of the ZGC configurations and not all. The ones that do perform better pause times do so vastly, but the ones that don't, all of which suffer from long-running GC events, perform notably worse.

The developers behind Shenandoah GC also made a comparison against the SPECjbb2015 [17], as mentioned in the related works chapter 3. They looked into Shenandoah GC versus G1GC. Their results showed higher throughput and lower pause time values (average and maximum) for Shenandoah compared to G1GC. However, they saw performance degradation of other metrics.

This is similar to what this study found, Shenandoah delivering better latency values than G1GC for DaCapo and the processing node. For the query node, however, the pause time values are only better for the configurations using the compact heuristic with an increased number of GC threads. The mentioned study also ran tests on Elasticsearch for which they saw notable slowdowns in runtime which aligns with the results of the query node; Shenandoah GC configurations compared to the Garbage First GC configurations.

6.1 DaCapo applications

Even though the DaCapo benchmark programs are a lot smaller than the two main application nodes of this study, they are able to give us some performance indications that apply to the processing node and query node. For one the latency metrics of average pause time, maximum pause time, and total stop-the-world time were notably lower for Shenandoah GC and ZGC compared to G1GC (about 80% for default configurations) which is true for most configurations in the other applications as well. Another factor is the moderate changes in measured duration time between the collectors. This is similar to the processing node results but not the query node results.

The DaCapo applications were over-provisioned with heap memory as they did only use fractions of their allowed space defined by -Xmx. This means that not much garbage collection work had to be conducted since there was always free space available.

6.2 Processing node

The ZGC collector tended to more often experience metadata occupancy problems although all three collectors allocated the same amount of memory for metadata if not otherwise specified. Shenandoah is the collector that uses Brooks pointers which should entail a larger memory footprint however this is not something the results portray which means the memory overhead can be considered negligible.

As promised the ZGC and Shenandoah GC both deliver significantly lower pause times than G1GC for the processing node (about 90% with applied tuning). All collectors do however stay within the requirement bounds of the processing node, maximum 1s pause times. It is also clear when looking at the GC pause duration time graphs in figure 5.3 that Shenandoah and ZGC

also produce uniform pauses within tighter bounds than G1GC. It is also plain to see that ZGC conducts the majority of its GC activity concurrently with mutator threads.

For the processing node, the results showed higher allocation rates for G1GC configurations compared to Shenandoah GC and ZGC. Low allocation rates can be bad as we want to utilize the heap efficiently and also keep up with the object allocation rate of the application. However, when allocation rates are too high we instead run at risk for failure to allocate memory and other problems caused by the GC being unable to properly handle the allocation of the application.

Looking at the singled out configurations in figure 5.11 z_moreConc and z_uncommit_meta executed in the shortest measured duration time, gave the highest throughput percentage values and lowest pause time values while having a reasonable average allocation rate. These make them best suitable to work along with the processing node server. However, z_moreConc had a non-fatal memory issue regarding metadata space. This problem is fixable by allocating more metadata space, much like z_meta. This may come at a cost but z_meta tells us it is likely cheap.

For z_uncommit_meta, all requirements for the processing node application are met with excellence and executes in a fast measured duration time. The average allocation rate is a bit low compared to G1GC and Shenandoah GC but does not cause any memory issues nor affect execution time. Given the above discussion, this configuration is therefore best suitable for the processing node. This in turn means that the Z Garbage Collector is the best suited collector for the job of garbage collection of the processing node application. Z_uncommit_meta is a configuration of ZGC with option
-XX:ZUncommit and metadata modifications *-XX:MetaspaceSize -XX:InitialBootClassLoaderMetaspaceSize, -XX:MinMetaspaceFreeRatio, -XX:MaxMetaspaceFreeRatio, -XX:MinMetaspaceExpansion, -XX:MaxMetaspaceExpansion*.

The processing node represents applications processing data in near real-time. If optimizing for low latency rather than high throughput, the results demonstrate that ZGC is still the garbage collector of choice. Although a different tuning approach should be taken. Z_xms (larger initial heap) performed the lowest latency values but suffered from frequent full metadata space. The metadata issue can be resolved by expanding the metadata space although it comes at a small cost in latency.

6.3 Query node

For the query node, ZGC configurations ran with an additional specification of concurrent GC threads, doubling the number from the default 5 to 10. Even so, the collector could not keep up with the allocation rates of the application and suffered from long-running GG events for all but one configuration, causing high execution times. These long execution times deem them unfit for the query node. At least when throughput is of value (and thus M-duration), which the requirements of the query node state.

Only one configuration executed the query in a measured duration time that could compete with Shenandoah and G1GC, namely z_interval. Z_interval conducted fixed interval garbage collection and did thus not listen as much to application behavior.

Additionally, several ZGC configurations did not meet the 10-20s maximum pause time requirement.

Shenandoah GC also struggled to keep up with the application pace. The default number of concurrent threads for Shenandoah is 9 and the collector managed to execute the query in default mode. Notable is that 9 is greater than ZGCs default of 5, and almost equal to the 10 with which the ZGC configurations ran. Increasing the number of concurrent GC threads only affected Shenandoah when using the compact heuristic. The compact heuristic was recommended as an option for reduced latency but in this case, it quite dramatically increased throughput as well as gave Shenandoah more "time" for GC activity.

ZGC and Shenandoah conduct more GC phases concurrently with mutator threads which means that increased high allocation rates would at some point push the need for more concurrent GC threads. Expectation could have been that ZGC would try to allocate more such threads perhaps by relation to the maximum set heap size (for default configurations). It is however abundantly clear that the configuration argument of concurrent GC threads is of the utmost importance when using Shenandoah and ZGC.

G1GC utilized much less of the allowed JVM heap size than ZGC and Shenandoah did in general. The few successful configurations of Shenandoah and ZGC also conformed to the pattern of only using a fraction of the allowed JVM heap space. A smaller heap size could contain the allocation rates thus keep the balance between application and GC work. A possibility could then be to reduce the ZGC and Shenandoah configurations maximum allowed heap, which is now over-provisioned, and by doing so control allocation rates. If

allocation rates are restrained then the default number of threads for ZGC and Shenandoah, 5 and 9 threads respectively, might perform better.

However, if the query node were to be more heavily used, e.g larger more demanding queries, the heap space allowed might come to use and a lower allowance would be insufficient. The key is likely to try to better match the heap size with the hunger of the application.

Another perspective is that the workload pressure of the query node may greatly vary over time. High heap utilization variation. The query that was chosen for the implementation of the performance evaluation of the query node in this thesis consist of one single query. The results tell us that 256G is an over-provision for this single query which results in long M-duration values. For production, however, we might need the 256G heap, when running multiple queries simultaneously (high heap utilization points), although we might not need it all of the time. There may still be time ranges where a single query is active, like in the test performed (low heap utilization points). When this is the case, the 256G heap maximum should be kept, and the recommendation would be to run ZGC with the z_interval configuration (*-XX:ZCollectionInterval*) plus additional concurrent GC threads, and run Shenandoah GC with the compact heuristic with a feasible increase of the number of concurrent GC threads (not to over-provision).

For applications with high heap utilization variation, optimizing for low latency, the ZGC collector is the recommended choice when used in combination with the collection interval option. Possibly with an increase of metadata space in order to avoid metadata occupancy threshold being reached too often triggering unnecessary GC events.

With the data results that we have for the query node and the requirements that are, G1GCs g1_lowHeapWaste and g1_expconc are both suitable configurations that balance throughput and latency. Shenandoah GCs sh_compact10, however, performs notably better pause time values and a higher throughput percentage, although the measured duration time is 3.5min longer in comparison to G1GC. This is also a suitable option for the query node server. ZGCs z_interval is also applicable, having the best latency values and a measured execution duration between G1GC and Shenandoah. Although it should be extended with a larger metadata space to solve the metadata occupancy problem.

Which configuration, and thus garbage collector, best suites the query node application depends on which throughput parameter that weighs heavier; M-duration or throughput%. The requirements of the query node tell us that

throughput is to be prioritized above latency, given that the latency needs are still met (maximum pause times under 10-20s). Furthermore, the next section 6.4, points to M-duration being a more valuable throughput metric to consider in this scenario. This leads the g1_expconc configuration to be the best choice for the query node application, amongst the tested configurations of this study. G1_expconc utilizes G1GC with additional options -*XX:+UseCompressedOops* and *XX:+ExplicitGCInvokesConcurrent*. Thus Garbage First GC is the way to go for the query node server (executing the tested query). Although for future application it would be wise to further consider combinations of the configurations that showed performance benefits.

The results of the query node, clearly show the great importance of tuning a collector. There are huge differences in both throughput and latency between the configurations within a collector. Especially for ZGC and Shenandoah.

6.4 Validity

Interesting regarding the throughput percentage metric is that GC-Easy do not regard concurrent GC work as time spent in GC but instead includes it in application time. This is a simplification. Ideally, time spent in a concurrent state would be weighed or split somehow between the two categories. Since this is not the case, the throughput percentage metric greatly favors ZGC and Shenandoah GC above G1GC as they spend a lot of time in a concurrent state and all of which is overlooked in the throughput% metric. This can explain the high throughput percentage values for configurations with long measured duration times and a detected problem of suffering from long-running GC events. Measured duration time (M-duration) on the other hand states the time duration for which logging was produced in the gc-log of a run. This thus measures the time from the startup of the application until shutdown. Ideally, for the query node, an additional metric would be added which measures only the time from query invocation until the query completes. The startup and shutdown may be affected by factors that lie outside of the actual query execution and thus affect the M-duration value. With that said, M-duration should perhaps still be considered a more accurate metric of throughput between the collectors, despite a possible fault margin, while throughput% is more useful for comparisons of configurations within a collector.

JClarity Censum discovered high CPU usage for the G1GC default configuration in both the processing node and the query node. High amounts of

kernel CPU utilization may indicate that the root cause for long-running GC pauses may lie outside of the JVM. If the cause is outside of the JVM it would likely have an equal effect on the ZGC and Shenandoah runs as well, however this remains unknown as Censum provided no deeper analysis of the gc-logs of ZGC or Shenandoah.

Although the tests ran on the same server and with the same environment configurations, this study was not the only project to utilize the server. Several other projects were active within that environment and could potentially have caused minor side effects. Referring back to the Censum detection of high CPU usage, other outside variables could have been affected too. The tests were run at all hours of the day, during a time span of 2 months. During this time the outside environment on the test server could have possibly changed as it was accessible to other projects in-between the test executions.

Another threat against the validity of the results is that a GC configuration may not perform uniformly on every run, even with the exact same environment and application. There may be smaller inconsistencies that affect performance. The method of this study involved executing each configuration of a collector *once*, instead of for example taking average result values of multiple executions.

6.5 Limitations

The benchmark suite used in this thesis is DaCapo 9.12. Ideally, the larger SPECjbb2015 benchmark suite would have been used in addition to DaCapo, however, for this study, there was not enough capital to purchase this suite. The SPECjbb2015 benchmarks are specifically developed for measuring Java server performance.

Another limitation of the study is that evaluation of other nodes/servers of the large heap transaction system could not be performed due to problems of isolation. Those could perhaps have been more interesting from a latency point of view as they have harder constraints regarding such matters.

All JVM options and variations of configurations could not be evaluated and exhausted in this study because of limitations in terms of project time and the sheer number of possible combinations.

Unfortunately, during the course of this study, the 2020 Covid-19 pandemic broke out. This further limited the scope of the study by affecting the

workflow and time schedule, but should however not have had any impact on the presented results.

7 Conclusions

The main objective of this study is to shed light on the performances of Garbage First GC, Shenandoah GC, and the Z Garbage Collector. This is achieved by looking at their respective performances on several applications within the same environment. This study was able to demonstrate the importance of garbage collector choice, and the impact of GC parameter tuning on application performance. Furthermore, the experiments of this study confirm the general latency advantage given by ZGC and Shenandoah GC compared to the JDK13 default GC, Garbage First GC. With the right tuning applied, a possibility of 90% improvement of average and maximum pause time values as well as accumulated STW time, for the processing node and query node. And about 80% latency improvement for the DaCapo benchmark programs.

ZGC uses more memory mappings per process than other collectors due to its multi-mapping feature. Thus ZGC should always be run in an environment with adjusted maximum mappings per process, fitting the application load, to avoid memory mapping failures.

Applications with high heap utilization variation, working on large heaps, optimizing for low latency - should consider the Z Garbage Collector firstly and Shenandoah GC secondly. It is recommended to increase number of concurrent GC threads (*-XX:ConcGCThreads*), and use the *-XX:ZCollectionInterval* and *-XX:ShenandoahGCHeuristics=compact* options respectively, in order to achieve high performance. The number of concurrent GC threads should be chosen as to increase GC parallelism without over-provisioning. ZGC, the top choice, may also be combined with metadata space options to avoid metadata occupancy issues (e.g *-XX:MetaspaceSize, -XX:Min/MaxMetaspaceFreeRatio*).

Applications processing data in near real-time (normalizing, indexing, analyzing, etc.), and optimizing for low latency, are recommended to use the Z Garbage Collector. ZGC significantly improve latency performance compared to G1GC, while maintaining the same throughput strength. It also performs slightly better than Shenandoah GC. Specifically, ZGC should be considered with additional options to increase the initial heap size to fit the application

and potentially larger metadata space as ZGC tend to quickly fill the metadata space with default metadata space allocation.

One of the principal research questions was which collector best suited the processing node and the query node, respectively, when obliging to their heap size limits, latency requirements, and high throughput aim while acknowledging a subset of available GC tuning options. Although several configurations of each collector could get the job done, the Z Garbage Collector was the top choice for the processing node. Specifically, ZGC in combination with the *-XX:ZUncommit* option and metadata space extensions. For the query node, the default configurations of ZGC and Shenandoah GC would be unfit for production, however, with applied tuning they managed to perform significantly better. Despite that, the best suitable collector for the query node is the Garbage First GC. It met the requirements with stability and performed the tasks with the highest throughput. Specifically G1GC with additional options *-XX:+UseCompressedOops* and *XX:+ExplicitGCInvokesConcurrent* performed well for the query node.

7.1 Future work

This study lays the groundwork for future work into the performance of ZGC, Shenandoah GC, and G1GC and which collector best suites which type of application requirements.

The amount of research on garbage collectors G1GC, Shenandoah and ZGC is still rather limited and thus there is much more to learn and investigate within the area. Future work could further investigate the performance effects of tuning options in singularity and in combination for all collectors. For instance, it would be interesting to evaluate the performance of combinations of some of the modified configurations with the "best" results within a collector and see if their benefits are added on top of each other or if one counteracts the other. It could also be interesting to run the ZGC and Shenandoah configurations in the same query node environment but with a smaller heap, testing the theory of over-provisioned heap space mentioned in the discussion section. This study also offers limited insight into the memory footprint and CPU usage of the garbage collectors. These aspects of performance would be interesting to further investigate. Another path could be to evaluate an application with other characteristics. This study further confirms that garbage collection performance is application dependent, thus it is always interesting to look into the performance of these same collectors in another application species. Both the processing node and query node of this study had rather re-

laxed latency requirements. It could therefore be of interest to look at a larger node within a system that has higher pause time demands. The impact of the hardware in the environment is yet another aspect of the GC performance that could be looked into and compared to other hardware setups.

Bibliography

[1] Maria Carpen-Amarie et al. “A Performance Study of Java Garbage Collectors on Multicore Architectures”. In: *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*. PMAM ’15. San Francisco, California: Association for Computing Machinery, 2015, pp. 20–29. ISBN: 9781450334044. DOI: 10.1145/2712386.2712404. URL: https://doi-org.focus.lib.kth.se/10.1145/2712386.2712404.

[2] Philipp Lengauer and Hanspeter Mössenböck. “The Taming of the Shrew: Increasing Performance by Automatic Parameter Tuning for Java Garbage Collectors”. In: *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*. ICPE ’14. Dublin, Ireland: Association for Computing Machinery, 2014, pp. 111–122. ISBN: 9781450327336. DOI: 10.1145/2568088.2568091. URL: https://doi-org.focus.lib.kth.se/10.1145/2568088.2568091.

[3] Oracle Per Lidén Stefan Karlsson - HotSpot garbage collection team. *The Z Garbage Collector*. 1999. URL: https://archive.fosdem.org/2018/schedule/event/zgc/attachments/slides/2211/export/events/attachments/zgc/slides/2211/ZGC_FOSDEM_2018.pdf (visited on 02/04/2020).

[4] Oracle. *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide, Release 9*. 2017. URL: https://docs.oracle.com/javase/9/gctuning/JSGCT.pdf (visited on 03/02/2020).

[5] Nicholas Walliman. *Research Methods: The Basics*. eng. 2nd ed. Routledge, 2018. ISBN: 9781138693982.

[6] H Grgic, B Mihaljevic, and A Radovan. “Comparison of garbage collectors in Java programming language”. eng. In: *2018 41st International Convention on Information and Communication Technology, Electron-*

ics and Microelectronics (MIPRO). Croatian Society MIPRO, 2018, pp. 1539–1544. ISBN: 9789532330953.

[7] P. Pufek, H. Grgić, and B. Mihaljević. "Analysis of Garbage Collection Algorithms and Memory Management in Java". In: *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. May 2019, pp. 1677–1682. DOI: 10.23919/MIPRO.2019.8756844.

[8] Lijie Xu et al. "An experimental evaluation of garbage collectors on big data applications". eng. In: *Proceedings of the VLDB Endowment* 12.5 (2019), pp. 570–583. ISSN: 21508097.

[9] Georgios Gousios, Vassilios Karakoidas, and Diomidis Spinellis. "Tuning Java's Memory Manager for High Performance Server Applications". In: (July 2008).

[10] Oracle. *Java SE HotSpot at a Glance*. URL: https://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html (visited on 02/26/2020).

[11] David Detlefs et al. "Garbage-First Garbage Collection". In: *Proceedings of the 4th International Symposium on Memory Management*. ISMM '04. Vancouver, BC, Canada: Association for Computing Machinery, 2004, pp. 37–48. ISBN: 1581139454. DOI: 10.1145/1029873.1029879. URL: https://doi-org.focus.lib.kth.se/10.1145/1029873.1029879.

[12] Oracle help center. *Garbage-First Garbage Collector*. URL: https://docs.oracle.com/javase/9/gctuning/garbage-first-garbage-collector.htm%7B%5C#%7DJSGCT-GUID-ED3AB6D3-FD9B-4447-9EDF-983ED2F7A573 (visited on 02/27/2020).

[13] Oracle. *HotSpot Virtual Machine Garbage Collection Tuning Guide*. URL: https://docs.oracle.com/en/java/javase/13/gctuning/garbage-first-garbage-collector.html#GUID-ED3AB6D3-FD9B-4447-9EDF-983ED2F7A573 (visited on 02/26/2020).

[14] Raoul-Gabriel Urma and Richard Warburton. *Understanding the JDK's New Superfast Garbage Collectors*. 2019. URL: https://blogs.oracle.com/javamagazine/understanding-the-jdks-new-superfast-garbage-collectors#anchor_4 (visited on 03/02/2020).

[15] Roman Kennke Iris Clark Aleksey Shipilev. *Shenandoah GC*. 2020. URL: https://wiki.openjdk.java.net/display/shenandoah/Main (visited on 02/26/2020).

[16] Aleksey Shipilev. *ShenandoahGC... and how it looks like in February 2018*. 2018. URL: https://www.jfokus.se/jfokus18/preso/Shenandoah-GC--What-We-Know-In-2018.pdf (visited on 02/25/2020).

[17] Christine H. Flood et al. "Shenandoah: An Open-Source Concurrent Compacting Garbage Collector for OpenJDK". In: *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. PPPJ '16. Lugano, Switzerland: Association for Computing Machinery, 2016. ISBN: 9781450341356. DOI: 10.1145/2972206.2972210. URL: https://doi.org/10.1145/2972206.2972210.

[18] Sadiq Jaffer Richard Warburton. *Java's new Z Garbage Collector (ZGC) is very exciting*. 2018. URL: https://www.opsian.com/blog/javas-new-zgc-is-very-exciting/ (visited on 02/26/2020).

[19] S. M. Blackburn et al. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". In: *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. Portland, OR, USA: ACM Press, Oct. 2006, pp. 169–190. DOI: http://doi.acm.org/10.1145/1167473.1167488.

[20] S. Jayasena et al. "Auto-Tuning the Java Virtual Machine". In: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. May 2015, pp. 1261–1270. DOI: 10.1109/IPDPSW.2015.84.

[21] Haoyu Li, Mingyu Wu, and Haibo Chen. "Analysis and Optimizations of Java Full Garbage Collection". In: *Proceedings of the 9th Asia-Pacific Workshop on Systems*. APSys '18. Jeju Island, Republic of Korea: Association for Computing Machinery, 2018. ISBN: 9781450360067. DOI: 10.1145/3265723.3265735. URL: https://doi-org.focus.lib.kth.se/10.1145/3265723.3265735.

[22] Kun Suo et al. "Characterizing and Optimizing Hotspot Parallel Garbage Collection on Multicore Systems". In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys '18. Porto, Portugal: Association for Computing Machinery, 2018. ISBN: 9781450355841. DOI: 10.1145/3190508.

3190512. URL: https://doi-org.focus.lib.kth.se/10.1145/3190508.3190512.

[23] Mart Mägi. “Visualising the logs of Shenandoah Garbage Collection algorithm Bachelor”. In:

A Appendix

DaCapo modified configuration test runs.

TRADEBEANS

Label	M-duration	jvmHeap alloc	jvmHeap peak	AvgAllocRate	Throughput%	avgPT (ms)	maxPT (ms)	Total STW	Total conc
g1_newsize1G6	2 min 15 sec 805 ms	2 gb	1.2 gb	252.49 mb/sec	99.462	16.986183	30.0	730ms	46.2ms
compressed63	2 min 15 sec 628 ms	2 gb	909 mb	250.85 mb/sec	99.261	14.52608	70.0	1s 2ms	52.2ms
expconc63	2 min 19 sec 846 ms	2 gb	895 mb	240.89 mb/sec	99.367	11.200553	20.0	885ms	112ms
lowHeapWaste63	2 min 16 sec 458 ms	2 gb	905 mb	247.94 mb/sec	99.289	15.164807	50.0	971ms	35.5ms
init63	2 min 15 sec 560 ms	2 gb	912 mb	249.62 mb/sec	99.336	14.292787	70.0	900ms	63.7ms
updatePT63	2 min 13 sec 927 ms	2 gb	902 mb	252.4 mb/sec	99.299	14.447748	50.0	939ms	92.9ms
meta_pure63	2 min 15 sec 561 ms	2 gb	906 mb	252.21 mb/sec	99.396	12.592457	20.0	819ms	86.2ms
string63	2 min 15 sec 258 ms	2 gb	899 mb	250.47 mb/sec	99.203	15.396636	70.0	1s 78ms	101ms
parallel63	2 min 15 sec 214 ms	2 gb	912 mb	250.26 mb/sec	99.258	14.745434	70.0	1s 3ms	80.9ms
period63	2 min 14 sec 590 ms	2 gb	728 mb	252.24 mb/sec	98.763	7.56514	60.0	1s 664ms	5s 819ms
g1_resere63	2 min 16 sec 180 ms	2 gb	892 mb	247.34 mb/sec	99.273	14.566608	80.0	991ms	49.7ms
g1_default_scila	2 min 18 sec 981 ms	2 gb	1.21 gb	273.48 mb/sec	99.286	20.264011	30.0	993ms	50.5ms
g1_default	2 min 15 sec 624 ms	2 gb	884 mb	250.06 mb/sec	99.253	14.467321	70.0	1s 13ms	59ms
throughput	2 min 14 sec 287 ms	2 gb	912 mb	252.11 mb/sec	99.244	14.291514	50.0	1s 15ms	149ms
latency	2 min 15 sec 434 ms	2 gb	910 mb	249.32 mb/sec	99.297	14.430508	60.0	952ms	28.6ms
meta-g1gc	2 min 16 sec 80 ms	2 gb	878 mb	251.28 mb/sec	99.349	12.66151	20.0	886ms	51.8ms
compressed_heapwaste	2 min 21 sec 861 ms	2 gb	943 mb	266.17 mb/sec	99.161	12.265239	70.0	1a 190ms	121ms
compressed_heapwaste_meta	2 min 22 sec 810 ms	2 gb	941 mb	263.12 mb/sec	99.258	16.060598	110.0	1s 60ms	0
sh_alloc_spike63	2 min 10 sec 303 ms	9.01 gb	8.52 gb	266.12 mb/sec	99.957	1.1576041	7.765	55.6ms	834ms
compact63	2 min 8 sec 819 ms	1.62 gb	1.27 gb	325.2 mb/sec	99.768	1.144348	14.122	299ms	10s 371ms
static63_freeT	2 min 8 sec 590 ms	7.08 gb	6.51 gb	254.64 mb/sec	99.963	1.1842002	8.124	47.4ms	786ms
sh_string63	2 min 14 sec 421 ms	9.02 gb	8.52 gb	261.64 mb/sec	99.952	1.3370835	7.667	64.2ms	763ms
sh_parallel63	2 min 7 sec 973 ms	9 gb	8.52 gb	270.92 mb/sec	99.957	1.1443126	7.546	54.9ms	816ms
sh_default_scila	2 min 9 sec 130 ms	9.02 gb	8.51 gb	263.05 mb/sec	99.95	1.6043501	7.9580007	64.2ms	693ms
sh_default	2 min 11 sec 223 ms	9.02 gb	8.52 gb	265.15 mb/sec	99.958	1.1428332	8.039	54.9ms	816ms
compact	2 min 10 sec 687 ms	1.61 gb	1.27 gb	315.26 mb/sec	99.772	1.127898	11.884	298ms	13s 129ms
static	2 min 11 sec 621 ms	9.1 gb	8.51 gb	263.63 mb/sec	99.967	1.1990557	7.67	43.2ms	719ms
unload	2 min 11 sec 202 ms	9.02 gb	8.51 gb	267.95 mb/sec	99.958	1.1477083	8.397	55.1ms	818ms
sh_meta	2 min 14 sec 457 ms	9.02 gb	8.52 gb	253.4 mb/sec	99.962	1.285725	7.505	51.4ms	806ms
static_parallel	2 min 22 sec 220 ms	2.38 gb	2.31 gb	48.15 mb/sec	99.98	1.4574503	9.246	29.1ms	109ms
sh_compressed	2 min 9 sec 620 ms	16.8 gb	15.79 gb	386.48 mb/sec	99.968	1.3136874	8.648	42ms	669ms
z_throughput63	2 min 5 sec 941 ms	9.69 gb	9.31 gb	337.62 mb/sec	99.956	1.9248967	8.784	55.8ms	2s 374ms
z_parallel63	2 min 5 sec 405 ms	9.77 gb	9.44 gb	358.23 mb/sec	99.95	1.598	7.855	62.3ms	2s 371ms
z_default63trans	2 min 1 sec 991 ms	9.24 gb	8.89 gb	327.47 mb/sec	99.955	1.4594738	8.002	55.5ms	2s 124ms
z_spike63	2 min 5 sec 320 ms	9.08 gb	8.78 gb	375.38 mb/sec	99.946	1.6860499	8.469	67.4ms	2s 487ms
z_uncommit63	2 min 4 sec 620 ms	8.97 gb	8.62 gb	325.07 mb/sec	99.949	1.7729166	8.065	63.8ms	2s 165ms
z_proactive63	2 min 5 sec 633 ms	9.77 gb	9.44 gb	365.29 mb/sec	99.939	1.6421062	8.225	77.2ms	2s 448ms
z_interval63	2 min 5 sec 724 ms	9.13 gb	8.78 gb	323.82 mb/sec	99.944	1.8603424	7.9970007	70.7ms	2s 191ms
z_default_scila	2 min 5 sec 363 ms	9.8 gb	9.44 gb	347.2 mb/sec	99.946	1.6580247	7.459	68ms	2s 202ms
z_default	2 min 4 sec 883 ms	9.71 gb	9.36 gb	371.05 mb/sec	99.948	1.5868778	7.657	65.1ms	2s 488ms
z_string	2 min 5 sec 320 ms	9.76 gb	9.41 gb	360.12 mb/sec	99.949	1.5443661	8.066	63.3ms	2s 373ms
z_meta	2 min 41 sec 380 ms	9.81 gb	9.47 gb	266.54 mb/sec	99.964	1.7583029	7.503	58ms	2s 283ms
z_throughputz_uncommit_meta	2 min 6 sec 144 ms	9.87 gb	9.43 gb	332.52 mb/sec	99.973	1.3373599	7.9979997	33.4ms	2s 832ms
z_moreConc	2 min 34 sec 449 ms	9.87 gb	9.52 gb	278.3 mb/sec	99.971	1.6517408	7.37	44.6ms	2s 73ms
z_compressed	2 min 4 sec 573 ms	7.19 gb	6.89 gb	318.91 mb/sec	99.958	1.4413333	7.606	51.9ms	1s 598ms

Figure A.1: DaCapo tradebeans results.

H2

Label	M-duration	jvmHeap a	jvmHeap p	AvgAllocRate	Throughput%	avgPT (ms)	maxPT (ms)	Total STW	Total conc
g1_newsize1G6	5 min 47 sec 514 ms	2.32 gb	1.97 gb	54.79 mb/sec	99.68	48.320175	160.0	1s 111ms	605ms
compressed63	5 min 50 sec 930 ms	3.51 gb	2.78 gb	55.82 mb/sec	99.606	33.736435	250.0	1s 383ms	1s 131ms
expconc63	5 min 45 sec 818 ms	3.54 gb	2.78 gb	53.56 mb/sec	99.725	29.728373	90.0	951ms	406ms
lowHeapWaste63	5 min 49 sec 449 ms	3.65 gb	2.86 gb	56.21 mb/sec	99.625	37.482025	240.0	1s 312ms	593ms
init63	5 min 47 sec 745 ms	2.82 gb	2.03 gb	56.99 mb/sec	99.689	41.614693	210.0	1s 82ms	548ms
updatePT63	5 min 48 sec 321 ms	2.7 gb	2.08 gb	56.14 mb/sec	99.68	37.123367	220.0	1s 114ms	1s 66ms
meta_pure63	5 min 50 sec 696 ms	2.68 gb	2.25 gb	55.5 mb/sec	99.686	35.578323	80.0	1s 103ms	1s 67ms
string63	5 min 51 sec 19 ms	2.38 gb	1.86 gb	55.39 mb/sec	99.625	37.651142	180.0	1s 318ms	1s 16ms
parallel63	5 min 57 sec 98 ms	2.38 gb	1.94 gb	55.13 mb/sec	99.685	37.451096	210.0	1s 124ms	1s 1ms
period63	5 min 24 sec 433 ms	3.81 gb	1.78 gb	69.2 mb/sec	97.99	8.434835	170.0	6s 520ms	2m 16s 377ms
g1_resere63	5 min 48 sec 902 ms	3.57 gb	2.5 gb	55.61 mb/sec	99.61	37.819275	220.0	1s 361ms	556ms
g1_default_scila	5 min 49 sec 275 ms	2.85 gb	2.27 gb	60.83 mb/sec	99.476	79.64902	120.0	1s 832ms	211ms
g1_default	5 min 44 sec 821 ms	3.49 gb	2.47 gb	56.47 mb/sec	99.643	35.19223	230.0	1s 232ms	550ms
throughput	5 min 46 sec 668 ms	2.36 gb	2 gb	57.3 mb/sec	99.662	43.404816	180.0	1s 172ms	564ms
latency	5 min 46 sec 153 ms	3.08 gb	2.3 gb	56.57 mb/sec	99.679	39.70504	230.0	1s 112ms	572ms
meta-g1gc	5 min 54 sec 846 ms	2.88 gb	2.51 gb	53.71 mb/sec	99.709	41.347042	110.0	1s 34ms	496ms
compressed_heapwaste	6 min 13 sec 110 ms	12.47 gb	2.33 gb	57.5 mb/sec	99.7	37.33333	290.0	1s 120ms	0
compressed_heapwaste_meta	6 min 12 sec 669 ms	11.83 gb	2.27 gb	57.32 mb/sec	99.691	38.333336	300.0	1s 150ms	0
sh_alloc_spike63	6 min 21 sec 158 ms	3.17 gb	2.5 gb	25.46 mb/sec	99.996	0.42059377	1.549	13.5ms	2s 29ms
compact63	6 min 10 sec 189 ms	2.5 gb	1.87 gb	45.01 mb/sec	99.988	0.49650005	2.254	43.7ms	12s 206ms
static63_freeT	6 min 11 sec 808 ms	7.25 gb	6.5 gb	30.35 mb/sec	99.998	0.47750005	1.646	7.64ms	1s 453ms
sh_string63	6 min 27 sec 355 ms	3.2 gb	2.5 gb	25.35 mb/sec	99.97	3.6830626	38.082	118ms	1s 978ms
sh_parallel63	6 min 22 sec 434 ms	3.15 gb	2.5 gb	25.59 mb/sec	99.996	0.46318746	1.609	14.8ms	2s 20ms
sh_default_scila	6 min 35 sec 796 ms	8.74 gb	8.47 gb	45.55 mb/sec	99.954	4.513925	39.129	181ms	2s 562ms
sh_default	6 min 19 sec 262 ms	3.15 gb	2.51 gb	25.76 mb/sec	99.996	0.4817188	1.976	15.4ms	2s 35ms
compact	6 min 28 sec 44 ms	2.24 gb	1.87 gb	43.91 mb/sec	99.99	0.414764	2.29	36.9ms	15s 716ms
static	6 min 24 sec 69 ms	9.16 gb	8.5 gb	20.49 mb/sec	99.999	0.66475004	1.796	5.32ms	866ms
unload	6 min 21 sec 237 ms	3.16 gb	2.5 gb	25.67 mb/sec	99.996	0.43887505	1.682	14ms	2s 52ms
sh_meta	6 min 9 sec 660 ms	3.16 gb	2.5 gb	26.14 mb/sec	99.996	0.4465	2.208	14.3ms	2s 89ms
static_parallel	6 min 35 sec 360 ms	17.73 gb	15.56 gb	36.83 mb/sec	99.999	0.6495	1.705	5.20ms	963ms
sh_compressed	6 min 28 sec 493 ms	17.69 gb	15.55 gb	37.48 mb/sec	99.999	0.6165	1.646	4.93ms	975ms
z_throughput63	6 min 21 sec 379 ms	5.81 gb	5.74 gb	63.86 mb/sec	99.997	0.42906672	2.489	12.9ms	7s 348ms
z_parallel63	6 min 22 sec 746 ms	5.8 gb	5.73 gb	66.05 mb/sec	99.998	0.23758331	0.771	8.55ms	7s 760ms
z_default63trans	6 min 15 sec 614 ms	5.9 gb	5.83 gb	68.62 mb/sec	99.997	0.2780833	2.475	10ms	7s 859ms
z_spike63	6 min 24 sec 42 ms	5.81 gb	5.73 gb	66.56 mb/sec	99.997	0.3069722	2.736	11.1ms	7s 963ms
z_uncommit63	6 min 20 sec 32 ms	5.89 gb	5.81 gb	67.88 mb/sec	99.997	0.34350002	1.286	12.4ms	7s 914ms
z_proactive63	6 min 16 sec 760 ms	5.86 gb	5.79 gb	67.93 mb/sec	99.997	0.33691892	2.58	12.5ms	7s 923ms
z_interval63	6 min 18 sec 101 ms	5.77 gb	5.69 gb	67.48 mb/sec	99.997	0.2651556	1.131	11.9ms	9s 795ms
z_default_scila	6 min 26 sec 914 ms	6.64 gb	6.57 gb	70.25 mb/sec	99.997	0.314	1.398	11.6ms	7s 731ms
z_default	6 min 21 sec 855 ms	5.83 gb	5.76 gb	67.24 mb/sec	99.996	0.3795854	1.398	15.6ms	7s 842ms
z_string	6 min 16 sec 443 ms	5.77 gb	5.7 gb	68.2 mb/sec	99.997	0.26254055	1.072	9.71ms	7s 921ms
z_meta	6 min 22 sec 633 ms	5.76 gb	5.68 gb	66.29 mb/sec	99.997	0.3504242	3.742	11.6ms	7s 254ms
z_throughputz_uncommit_meta	6 min 20 sec 377 ms	5.75 gb	5.67 gb	63.01 mb/sec	99.999	0.2125833	0.765	5.10ms	7s 936ms
z_moreConc	6 min 51 sec 292 ms	19.35 gb	18.94 gb	68.57 mb/sec	99.999	0.34288886	1.353	3.09ms	1s 909ms
z_compressed	6 min 53 sec 522 ms	19.36 gb	18.97 gb	68.38 mb/sec	99.999	0.29266667	1.075	2.63ms	1s 570ms

Figure A.2: DaCapo h2 results.

JYTHON

Label	M-duration	jvmHeap alloc	jvmHeap peak	AvgAllocRate	Throughput%	avgPT (ms)	maxPT (ms)	Total STW	Total conc
g1_newsize1G6	17 sec 465 ms	1.07 gb	1 gb	576.7 mb/sec	99.542	11.428571	20.0	80ms	0
compressed63	17 sec 679 ms	1.09 gb	685 mb	549.18 mb/sec	99.095	11.428572	20.0	160ms	0
expconc63	17 sec 197 ms	2 gb	632 mb	551.67 mb/sec	99.259	9.103428	10.0	127ms	10.1ms
lowHeapWaste63	17 sec 429 ms	1.09 gb	685 mb	563.83 mb/sec	99.311	10.909089	20.0	120ms	0
init63	17 sec 452 ms	1.09 gb	683 mb	568.7 mb/sec	99.026	11.333333	20.0	170ms	0
updatePT63	17 sec 238 ms	1.09 gb	685 mb	583.71 mb/sec	99.246	14.444445	20.0	130ms	0
meta_pure63	17 sec 313 ms	2 gb	632 mb	576.04 mb/sec	99.148	9.219312	10.0	148ms	12.9ms
string63	17 sec 418 ms	1.09 gb	682 mb	569.58 mb/sec	98.967	12.0	20.0	180ms	0
parallel63	17 sec 471 ms	1.09 gb	683 mb	565.51 mb/sec	99.256	11.81818	20.0	130ms	0
period63	18 sec 50 ms	2 gb	674 mb	557.12 mb/sec	98.361	11.8336	32.715	296ms	66.1ms
g1_resere63	17 sec 430 ms	1.09 gb	685 mb	578.08 mb/sec	99.139	13.636365	20.0	150ms	0
g1_default_scila	17 sec 718 ms	2 gb	1 gb	571.23 mb/sec	99.556	8.741221	10.0	78.7ms	8.95ms
g1_default	17 sec 438 ms	1.09 gb	685 mb	552.53 mb/sec	99.197	11.666667	20.0	140ms	0
throughput	17 sec 282 ms	1.09 gb	683 mb	574.88 mb/sec	99.016	11.333333	20.0	170ms	0
latency	17 sec 422 ms	1.09 gb	684 mb	565.89 mb/sec	99.139	12.5	20.0	150ms	0
meta-g1gc	17 sec 425 ms	2 gb	631 mb	567.92 mb/sec	99.32	9.110153	10.0	118ms	12.5ms
compressed_heapwaste	18 sec 92 ms	2 gb	760 mb	591.86 mb/sec	99.032	10.949314	40.0	175ms	8.69ms
compressed_heapwaste_meta	17 sec 945 ms	1.16 gb	712 mb	596.43 mb/sec	99.331	11.999998	20.0	120ms	0
sh_alloc_spike63	18 sec 572 ms	2.54 gb	2.51 gb	535.43 mb/sec	99.925	0.8723125	4.052	14ms	69ms
compact63	18 sec 686 ms	1.08 gb	1.03 gb	460.72 mb/sec	99.838	0.757375	3.91	30.3ms	326ms
static63_freeT	19 sec 604 ms	6.53 gb	6.51 gb	338.91 mb/sec	99.952	1.173625	3.816	9.39ms	37.3ms
sh_string63	18 sec 544 ms	2.54 gb	2.51 gb	535.81 mb/sec	99.907	1.0808749	4.053	17.3ms	73.9ms
sh_parallel63	18 sec 521 ms	2.54 gb	2.51 gb	538.2 mb/sec	99.931	0.80112505	3.674	12.8ms	71.7ms
sh_default_scila	18 sec 365 ms	2.55 gb	2.51 gb	541.03 mb/sec	99.911	1.0267501	4.161	16.4ms	72.1ms
sh_default	18 sec 755 ms	2.53 gb	2.51 gb	531.91 mb/sec	99.936	0.7493125	3.278	12ms	71.6ms
compact	18 sec 896 ms	1.09 gb	1.03 gb	452.05 mb/sec	99.825	0.8268749	3.9989998	33.1ms	382ms
static	20 sec 467 ms	8.54 gb	8.51 gb	424.68 mb/sec	99.952	1.2312502	3.568	9.85ms	38.8ms
unload	18 sec 709 ms	2.54 gb	2.51 gb	531.29 mb/sec	99.912	1.0281249	3.609	16.4ms	74ms
sh_meta	18 sec 427 ms	2.54 gb	2.51 gb	540.78 mb/sec	99.928	0.8310625	4.39	13.3ms	75.3ms
static_parallel	20 sec 729 ms	2.03 gb	183 mb	4.63 mb/sec	99.965	0.9051251	2.819	7.24ms	14.3ms
sh_compressed	20 sec 546 ms	2.03 gb	168 mb	3.89 mb/sec	99.958	1.0712501	3.311	8.57ms	15.4ms
z_throughput63	19 sec 33 ms	3.99 gb	3.89 gb	640.26 mb/sec	99.959	0.4825	1.867	7.72ms	198ms
z_parallel63	18 sec 945 ms	3.45 gb	3.35 gb	667.51 mb/sec	99.966	0.3552778	0.756	6.39ms	200ms
z_default63trans	19 sec 51 ms	3.71 gb	3.62 gb	665.37 mb/sec	99.954	0.43390003	2.021	8.68ms	190ms
z_spike63	18 sec 906 ms	3.65 gb	3.53 gb	665.4 mb/sec	99.95	0.47169998	1.353	9.43ms	183ms
z_uncommit63	19 sec 356 ms	4.07 gb	3.98 gb	694.98 mb/sec	99.955	0.48149997	2.088	8.67ms	210ms
z_proactive63	18 sec 837 ms	3.82 gb	3.73 gb	684.18 mb/sec	99.954	0.4786667	2.189	8.62ms	206ms
z_interval63	19 sec 142 ms	3.64 gb	3.51 gb	656.98 mb/sec	99.953	0.49805555	2.028	8.96ms	188ms
z_default_scila	19 sec 179 ms	3.76 gb	3.66 gb	705.35 mb/sec	99.952	0.48389477	1.779	9.19ms	205ms
z_default	18 sec 993 ms	3.51 gb	3.42 gb	665.4 mb/sec	99.956	0.46400002	2.124	8.35ms	188ms
z_string	19 sec 89 ms	3.24 gb	3.12 gb	657.13 mb/sec	99.961	0.4104444	1.872	7.39ms	195ms
z_meta	18 sec 929 ms	3.3 gb	3.2 gb	666.7 mb/sec	99.968	0.33511114	0.898	6.03ms	200ms
z_throughputz_uncommit_meta	19 sec 311 ms	4.54 gb	4.44 gb	689.24 mb/sec	99.965	0.417	1.114	6.67ms	238ms
z_moreConc	21 sec 803 ms	12.71 gb	12.61 gb	594.87 mb/sec	99.987	0.46733335	1.741	2.80ms	80.1ms
z_compressed	21 sec 999 ms	2 gb	122 mb	6.82 mb/sec	99.991	0.3145	1.021	1.89ms	26.6ms

Figure A.3: DaCapo jython results.

AVRORA

Label	M-duration	jvmHeap alloc	jvmHeap peak	AvgAllocRate	Throughput%	avgPT (ms)	maxPT (ms)	Total STW	Total conc
g1_newsize1G6	34 sec 485 ms	1.01 gb	9 mb	267 kb/sec	99.971	10.0	10.0	10ms	0
compressed63	33 sec 640 ms	40 mb	25 mb	7.16 mb/sec	99.847	7.3324285	10.0	51.3	15.4
expconc63	30 sec 270 ms	2 gb	16 mb	9.02 mb/sec	99.546	2.590566	10.0	137	283
lowHeapWaste63	33 sec 842 ms	40 mb	25 mb	7.12 mb/sec	99.789	7.9308877	10.0	71.4	14.9
init63	30 sec 588 ms	1.02 gb	15 mb	1.6 mb/sec	99.869	10.0	10.0	40.0	0
updatePT63	33 sec 234 ms	40 mb	25 mb	6.47 mb/sec	99.785	7.936556	10.0	71.4	16.3
meta_pure63	32 sec 670 ms	2 gb	17 mb	7.96 mb/sec	99.768	5.0548663	10.0	75.8	54.4
string63	34 sec 130 ms	40 mb	25 mb	7 mb/sec	99.79	7.9461102	10.0	71.5	13.1
parallel63	35 sec 639 ms	40 mb	27 mb	6.65 mb/sec	99.827	7.688625	10.0	61.5	14.4
period63	30 sec 148 ms	40 mb	20 mb	8.29 mb/sec	99.593	3.9551933	10.0	123	140
g1_resere63	34 sec 839 ms	40 mb	23 mb	6.4 mb/sec	99.91	6.28	10.0	31.4	16.3
g1_default_scila	34 sec 570 ms	2 gb	8 mb	236 kb/sec	99.965	4.049	10.0	12.1	5.63
g1_default	31 sec 18 ms	40 mb	25 mb	7.77 mb/sec	99.705	8.3180895	10.0	91.5	12.8
throughput	37 sec 915 ms	44 mb	25 mb	5.83 mb/sec	99.653	8.7655325	10.0	131	16
latency	30 sec 300 ms	40 mb	25 mb	7.89 mb/sec	99.864	6.8845005	10.0	41.3	14.5
meta-g1gc	33 sec 309 ms	2 gb	17 mb	7.11 mb/sec	99.623	6.9801106	10.0	126	43.6
compressed_heapwaste	33 sec 700 ms	80 mb	24 mb	8.04 mb/sec	99.644	9.999999	10.0	120	0
compressed_heapwaste_meta	34 sec 18 ms	80 mb	24 mb	8 mb/sec	99.824	9.999999	10.0	60	0
sh_alloc_spike63	31 sec 909 ms	2 gb	24 mb	n/a	99.991	0.719	1.465	2.88	3.79
compact63	31 sec 864 ms	2 gb	329 mb	10.17 mb/sec	99.981	0.76675	1.792	6.13	24.5
static63_freeT	29 sec 457 ms	2 gb	24 mb	n/a	99.99	0.71949995	1.374	2.88	3.15
sh_string63	32 sec 965 ms	2 gb	24 mb	n/a	99.991	0.74724996	1.607	2.99	3.26
sh_parallel63	31 sec 129 ms	2 gb	24 mb	n/a	99.993	0.58075	1.296	2.32	3.78
sh_default_scila	30 sec 247 ms	2 gb	28 mb	n/a	99.991	0.68025	1.405	2.72	3.43
sh_default	28 sec 726 ms	2 gb	24 mb	n/a	99.993	0.51975	0.992	2.08	3.03
compact	37 sec 786 ms	2 gb	241 mb	6.25 mb/sec	99.984	0.74675	1.617	5.97	26.7
static	38 sec 443 ms	2 gb	24 mb	n/a	99.994	0.619	1.379	2.48	3.13
unload	32 sec 769 ms	2 gb	24 mb	n/a	99.993	0.6100001	1.239	2.44	3.48
sh_meta	31 sec 704 ms	2 gb	28 mb	n/a	99.991	0.7175	1.388	2.87	4.15
static_parallel	34 sec 330 ms	2.02 gb	96 mb	n/a	99.991	0.77674997	1.62	3.11	4.43
sh_compressed	27 sec 475 ms	2.02 gb	64 mb	n/a	99.989	0.73775005	1.548	2.95	3.15
z_throughput63	34 sec 332 ms	2 gb	16 mb	477 kb/sec	99.998	0.22200002	0.351	0.666	12.2
z_parallel63	30 sec 618 ms	2 gb	14 mb	468 kb/sec	99.998	0.16533333	0.22	0.496	12.6
z_default63trans	31 sec 209 ms	2 gb	16 mb	524 kb/sec	99.998	0.24000001	0.388	0.720	13.1
z_spike63	30 sec 852 ms	2 gb	16 mb	531 kb/sec	99.998	0.20566669	0.293	0.617	12.8
z_uncommit63	34 sec 50 ms	2 gb	18 mb	541 kb/sec	99.998	0.261	0.373	0.783	11.5
z_proactive63	31 sec 585 ms	2 gb	16 mb	518 kb/sec	99.998	0.19500001	0.257	0.585	12.6
z_interval63	34 sec 461 ms	2 gb	312 mb	8.71 mb/sec	99.997	0.193	0.298	1.16ms	31.3
z_default_scila	34 sec 676 ms	2 gb	14 mb	413 kb/sec	99.998	0.26666668	0.474	0.800	12.8
z_default	38 sec 855 ms	2 gb	16 mb	421 kb/sec	99.999	0.176	0.246	0.528	12.1
z_string	32 sec 821 ms	2 gb	16 mb	499 kb/sec	99.998	0.19699998	0.267	0.591	10.7
z_meta	35 sec 210 ms	2 gb	18 mb	523 kb/sec	99.998	0.18566667	0.251	0.557	11.4
z_throughputz_uncommit_meta	33 sec 947 ms	2 gb	16 mb	482 kb/sec	99.998	0.20199998	0.266	0.606	12.8
z_moreConc	38 sec 596 ms	2 gb	16 mb	424 kb/sec	99.999	0.18533333	0.243	0.556	11.1
z_compressed	32 sec 50 ms	2 gb	16 mb	511 kb/sec	99.998	0.19466665	0.285	0.594	12.9

Figure A.4: DaCapo avrora results.

LUSEARCH-FIX

Label	M-duration	jvmHeap alloc	jvmHeap peak	AvgAllocRate	Throughput%	avgPT (ms)	maxPT (ms)	Total STW	Total conc
g1_newsize1G6	1 sec 383 ms	1.02 gb	1 gb	1.44 gb/sec	98.554	10.0	10.0	20ms	0
compressed63	1 sec 570 ms	1.02 gb	632 mb	1.41 gb/sec	98.641	5.3337502	10.0	21.3ms	5.18ms
expconc63	1 sec 328 ms	2 gb	630 mb	1.5 gb/sec	96.052	8.737832	20.0	52.4ms	10.9ms
lowHeapWaste63	1 sec 399 ms	1.26 gb	594 mb	1.1 gb/sec	99.194	3.7606666	10.0	11.3ms	26.9ms
init63	1 sec 417 ms	1.02 gb	630 mb	1.31 gb/sec	99.294	10.0	10.0	10ms	0
updatePT63	1 sec 418 ms	1.02 gb	630 mb	1.25 gb/sec	99.907	0.6625	1.228	1.33ms	18.3ms
meta_pure63	1 sec 529 ms	2 gb	630 mb	1.12 gb/sec	97.887	6.460799	10.0	32.3ms	12.6ms
string63	1 sec 470 ms	1.22 gb	632 mb	1.13 gb/sec	97.869	6.2646	10.0	31.3ms	49.7ms
parallel63	1 sec 235 ms	1.02 gb	630 mb	1.47 gb/sec	96.647	8.282999	20.0	41.4ms	17.4ms
period63	1 sec 249 ms	1.26 gb	633 mb	1.75 gb/sec	97.493	6.2619996	10.0	31.3ms	9.46ms
g1_resere63	1 sec 521 ms	1.02 gb	633 mb	1.34 gb/sec	98.59	5.36225	10.0	21.4ms	7.36ms
g1_default_scila	1 sec 583 ms	2 gb	1 gb	1.26 gb/sec	98.611	5.4954996	10.0	22ms	5.71ms
g1_default	1 sec 452 ms	1.22 gb	572 mb	1.1 gb/sec	98.537	5.3107495	10.0	21.2ms	20.5ms
throughput	1 sec 349 ms	1.02 gb	630 mb	1.3 gb/sec	99.259	10.0	10.0	10ms	0
latency	1 sec 487 ms	1.02 gb	630 mb	1.15 gb/sec	99.243	3.7500002	10.0	11.3ms	12.6ms
meta-g1gc	1 sec 450 ms	2 gb	630 mb	1.17 gb/sec	97.753	6.5170007	10.0	32.6ms	13.2ms
compressed_heapwaste	1 sec 375 ms	1.28 gb	617 mb	1.72 gb/sec	97.818	10.0	10.0	30ms	0
compressed_heapwaste_meta	1 sec 305 ms	1.28 gb	617 mb	1.68 gb/sec	98.467	10.0	10.0	20ms	0
sh_alloc_spike63	1 sec 907 ms	2 gb	24 mb	n/a	99.844	0.74475	1.473	2.98ms	4.17ms
compact63	1 sec 865 ms	2 gb	1.15 gb	941.55 mb/sec	99.602	0.61925	1.758	7.43ms	77.5ms
static63_freeT	2 sec 99 ms	2 gb	20 mb	n/a	99.839	0.84575	1.639	3.38ms	3.49ms
sh_string63	1 sec 870 ms	2 gb	20 mb	n/a	99.821	0.8365	1.702	3.35ms	4.53ms
sh_parallel63	2 sec 6 ms	2 gb	20 mb	n/a	99.846	0.77199996	1.518	3.09ms	4.61ms
sh_default_scila	1 sec 991 ms	2 gb	20 mb	n/a	99.855	0.72024995	1.556	2.88ms	3.49ms
sh_default	2 sec 218 ms	2 gb	20 mb	n/a	99.885	0.63824993	1.265	2.55ms	4.45ms
compact	1 sec 838 ms	2 gb	1.13 gb	948.86 mb/sec	99.646	0.5416667	1.613	6.50ms	125ms
static	1 sec 949 ms	2 gb	20 mb	n/a	99.848	0.7395	1.511	2.96ms	4.48ms
unload	2 sec 112 ms	2 gb	20 mb	n/a	99.871	0.68299997	1.397	2.73ms	4.11ms
sh_meta	1 sec 902 ms	2 gb	20 mb	n/a	99.857	0.67875	1.282	2.71ms	3.59ms
static_parallel	2 sec 12 ms	2.02 gb	96 mb	n/a	99.881	0.59875	1.233	2.39ms	4.06ms
sh_compressed	2 sec 84 ms	2.02 gb	80 mb	n/a	99.851	0.7775	1.647	3.11ms	3.90ms
z_throughput63	1 sec 548 ms	2.41 gb	2.22 gb	1.42 gb/sec	99.923	0.19766666	0.323	1.19ms	54ms
z_parallel63	1 sec 544 ms	2.39 gb	2.22 gb	1.43 gb/sec	99.928	0.18650001	0.387	1.12ms	45.2ms
z_default63trans	1 sec 654 ms	2.4 gb	2.2 gb	1.32 gb/sec	99.937	0.17333333	0.265	1.04ms	52.5ms
z_spike63	1 sec 512 ms	2.61 gb	2.38 gb	1.57 gb/sec	99.863	0.29557145	1.044	2.07ms	54.7ms
z_uncommit63	1 sec 650 ms	2.32 gb	2.15 gb	1.29 gb/sec	99.907	0.25466666	0.403	1.53ms	48.2ms
z_proactive63	1 sec 486 ms	2.41 gb	2.07 gb	1.38 gb/sec	99.925	0.18516666	0.382	1.11ms	76.7ms
z_interval63	1 sec 524 ms	2.5 gb	2.3 gb	1.49 gb/sec	99.904	0.24283335	0.382	1.46ms	48.3ms
z_default_scila	1 sec 672 ms	2.44 gb	2.2 gb	1.3 gb/sec	99.921	0.21966664	0.392	1.32ms	60ms
z_default	1 sec 441 ms	2.42 gb	2.26 gb	1.55 gb/sec	99.898	0.24499997	0.532	1.47ms	45ms
z_string	1 sec 427 ms	2.43 gb	2.27 gb	1.58 gb/sec	99.92	0.1903333	0.383	1.14ms	40.6ms
z_meta	1 sec 576 ms	2.37 gb	2.15 gb	1.36 gb/sec	99.924	0.20050001	0.371	1.20ms	52.8ms
z_throughputz_uncommit_meta	1 sec 613 ms	2.6 gb	2.36 gb	1.46 gb/sec	99.878	0.28028572	1.043	1.96ms	55.5ms
z_moreConc	1 sec 573 ms	2 gb	14 mb	8.9 mb/sec	99.958	0.22066668	0.305	0.662ms	9.95ms
z_compressed	1 sec 557 ms	2 gb	12 mb	7.71 mb/sec	99.951	0.25266668	0.381	0.758ms	12.6ms

Figure A.5: DaCapo lusearch-fix results.

TRITA-EECS-EX-2020:477