A Performance Comparison of Modern Garbage Collectors for Big Data Environments

Carlos Daniel Oliveira Gonçalves

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Paulo Jorge Pires Ferreira
Dr. Rodrigo Fraga Barcelos Paulus Bruno

Examination Committee

Chairperson: Prof. Miguel Nuno Dias Alves Pupo Correia
Supervisor: Prof. Paulo Jorge Pires Ferreira
Member of the Committee: Prof. Luís Manuel Antunes Veiga

January 2021

Acknowledgments

First and foremost, I would like to give a special thanks to my thesis supervisors, Professor Paulo Ferreira and Rodrigo Bruno, for all the patience, support, and availability since we started this project. They went above and beyond to always provide me with materials and opportunities to improve my work.

Then, I would like to thank my friends, who always kept me company and encouraged me during this journey—never forgetting Tagus4life for providing me fantastic moments of leisure and fun to overcome times of greater pressure.

Last but not least, a special thanks to my family for all their love, support and dedication during the good and bad times. This accomplishment would not have been possible without any of you. Thank you.

Abstract

The use of Java to develop Big Data platforms (e.g., Hadoop, Spark) has been a favoured choice among developers due to its fast development of large-scale systems, in part due to the automatic memory management. However, the impact of garbage collection on these platforms has been more and more of a concern, as platforms increasingly require lower pause times, higher throughput, and better memory usage by the garbage collector. In this project, we aim to understand how different garbage collectors scale in terms of throughput, latency, and memory usage in memory-hungry environments, so that given a platform with particular performance needs we may map the most suitable garbage collection (GC) algorithm. Previous works on this subject have used workloads that either failed to represent realistic use case scenarios of Big Data platforms or were run on top of academic implementations of the JVM, that are not meant to run Big Data applications. In this work, we use a combination of Big Data platforms (e.g., Cassandra, Lucene, and GraphChi) and real-world-based benchmarks (e.g., DaCapo) on top of an industrial JVM (OpenJDK HotSpot JVM) which provides a high degree of accuracy to the results. Additionally, we develop fine-grained benchmarks to study in more detail how particular techniques (e.g., barriers) employed by garbage collectors affect different performance metrics.

Keywords

Garbage collection, Big Data, scalability, Java, Big Data environments, Big Data platforms, Java Virtual Machine

Resumo

O uso de Java para desenvolver plataformas de Big Data (Hadoop, Spark) tem sido a escolha preferida entre desenvolvedores devido ao rápido desenvolvimento de sistemas de grande escala, em parte devido ao gerenciamento automático de memória. No entanto, o da reciclagem automática de memória nessas plataformas tem sido cada vez mais uma preocupação, pois estas plataformas cada vez mais exigem tempos de pausa menores, maior rendimento e melhor uso de memória pelo coletor de lixo. Neste projeto, pretendemos compreender como diferentes coletores de lixo escalam em termos de rendimento, latência e uso de memória em ambientes com grande consumo de memória, de modo que, dada uma plataforma com necessidades de desempenho específicas, possamos mapear o algoritmo de coleta de lixo (GC) mais adequado . Trabalhos anteriores sobre este assunto usaram cargas de trabalho que não conseguiram representar cenários de caso de uso realistas de plataformas de Big Data ou foram executados em cima de implementações acadêmicas da JVM, que não se destinam a executar aplicativos de Big Data. Neste trabalho, usamos uma combinação de plataformas de Big Data (Cassandra, Lucene e GraphChi) e benchmarks baseados no mundo real (DaCapo) em cima de uma JVM industrial (OpenJDK HotSpot JVM) que fornece um alto grau de precisão aos resultados. Além disso, desenvolvemos benchmarks de baixa granularidade para estudar em mais detalhes como técnicas específicas (por exemplo, barreiras de sincronização) empregadas por coletores de lixo afetam diferentes métricas de desempenho.

Palavras Chave

Reciclagem automática de memória, Big Data, Escalabilidade, Java, Ambientes de Big Data, Plataformas de Big Data, Maquina Virtual Java

Contents

List of Figures

List of Tables

Listings

1 Introduction

Contents

There is an increasing need to manage Big Data [11], whose term is often used to describe large data sets, rapidly growing, and in need to be processed rapidly in order extract value of large quantities of information. The use of Java to develop Big Data platforms (e.g., Hadoop [12], Spark [13]), on which Big Data applications that manage such data are executed, has been a favoured choice among developers. The use of Java results in faster development of large-scale systems, mainly due to the automatic memory management, and the large number of available resources made by the community. However, there is a cost associated with using automatic memory management. This cost is introduced mostly by the garbage collector (GC), which is responsible for reclaiming the memory space occupied by unreferenced objects and giving it back when the application needs. A possible solution to eliminate this cost would be to move back to an unmanaged language (e.g., C or C++), where the developer is responsible for memory managing. However, we would have to take into account the cost of a more extended development period, due to debugging memory problems and lack of reliability of such applications that would have many more errors forcing, in some cases, such applications to crash [14].

Applications such as credit card fraud detection, social networks management, financial analysis are examples of Big Data applications that need garbage collectors to scale in terms throughout and latency. However, currently used (classic) garbage collectors are not designed to be used with large-scale Big Data platforms, as they fail to scale regarding throughput and pause time. Currently used GCs require stop-the-world collections to free the garbage from the heap, which stops the application while the collection is in progress. As heap sizes grow with large-scale Big Data platforms, the time it takes to perform these collections also increases (since the time it takes is proportional to the heap's size), therefore, stopping the application for more extended periods, which significantly affects the application throughput negatively. Also, newly developed garbage collectors like ZGC[1] or Shenandoah[2], even though they were designed with such platforms in mind (GCs which collect the heap concurrently with the application running), we still do not entirely understand their impact on the performance of such platforms in real use case scenarios. This need to understand what implementation of a garbage collector is most suitable to fulfil the needs of an application, and understand where the strengths and weaknesses of such garbage collector lie upon, is what this thesis will attempt to address.

1.1 Problems

Every garbage collector has specific characteristics that makes it distinct from others. Understanding how these characteristics impact the different performance metrics of a particular program is a complex challenge. Not only does the performance depend on the topology and amount of objects at the heap, but also on the access patterns of the application. Additionally, the metrics are also not independent

[1] https://openjdk.java.net/jeps/333
[2] https://openjdk.java.net/jeps/189

variables; so, a variation of a particular parameter to achieve a specific objective may lead to other effects (positive or negative) on other variables. The aforementioned problems are also responsible for the difficulty in mapping a garbage collector to an application with specific performance requirements.

Measurements with benchmarks suites that comprise a small number of real use case programs can potentially hide problems that do not show up with such a small sample of scenarios. Another problem with using small benchmarks suites is that a garbage collector may be optimised for specific environments and therefore, responsible for introducing biased results to the experiment.

1.2 Goals

In this thesis, the goal is to understand how different garbage collectors impact different metrics (described in Section 2.3.2), more particularly latency, throughput, and memory usage. Using this knowledge, given an application with particular performance requisites (w.r.t application throughput, latency and memory usage), we give hints to which garbage collector implementation is most suitable to fulfil its requirements.

1.3 Previous Solutions

The use of small-scale benchmarks [7–9] in previous studies has long been criticised as inadequate, potentially hiding scalability problems in certain garbage collectors. In fact, such benchmarks are not able to exercise garbage collectors so that they would start showing degrading performance regarding throughput, pause times, and memory usage.

We know how newer garbage collectors, e.g., ZGC and Shenandoah, perform in benchmarks specifically meant to test pause times like SPECjbb2015 [15, 16]. However, we have yet to see how they perform in large scale benchmark suites consisting of widely used programs that are believed to represent a wide range of typical behaviour.

Additionally, many previous works evaluating garbage collectors in the past were executed on top of academic implementations of the Java Virtual Machine (JVM), (e.g., JikesRVM [8,9]) that are not meant to run Big Data applications. These implementations have other significant performance issues that are not inherent to the garbage collection process which may negatively affect the accuracy of the results.

1.4 Description of the Proposed Solution

To acquire the results on which to compare the tradeoffs between the variously selected garbage collectors (i.e., ParallelOld, CMS, G1, Shenandoah and ZGC), we use a combination of real-world applications and data running on top of an industrial-grade JVM, coupled with real-world-based benchmarks to analyse the performance. To such effect, the following workloads were chosen: DaCapo [17], Cassandra [18], Lucene [19], and GraphChi [20]. The decision to use the workloads mentioned above was mainly due to the following factors: (i) the use of real-world application and data provide a high degree of accuracy to the results (e.g., Cassandra, GraphChi and Lucene); (ii) using a widely used and scrutinised benchmark suite covering several distinct workloads allows us to determine how the selected garbage collector performs in various scenarios (e.g., DaCapo); (iii) using workloads used by past works enable us to compare our results with previous solutions (e.g., DaCapo). Additionally, a fine-grained evaluation on the garbage collectors is proposed, using small self-made benchmark that stress specific GC components, so that we can understand how certain techniques employed by these GCs affect performance on memory-hungry environments.

1.5 Contributions and Document Roadmap

We expect this research to contribute to the field of modern garbage collectors for Big Data environments through the following contributions:

- Provide better knowledge on how to match applications to a specific garbage collector, taking into consideration the application needs and the garbage collector tradeoffs.

- Give hints w.r.t. which garbage collector to pick when we want to optimize a particular performance metric.

- An overview of the state of art garbage collectors developed for the Java Virtual Machine and how they improve upon older implementations.

- Design and development of a fine-grained benchmark meant to stress-test specific GC components, e.g., write barriers, read barriers.

The following Chapter (Chapter 2) presents some background aspects which are fundamental to understand the remaining document (i.e., background concepts about Big Data environments, the JVM and GC). Then, Chapter 3 describes relevant related work regarding the current state of research related to studying and evaluating GC for Big Data Environments. Chapter 4 describes the mains aspects of the system used to compare the different GCs. Finally, in Chapters 5 and 6, we address the evaluation performed, and show some conclusions and future work, respectively.

2

Background

Contents

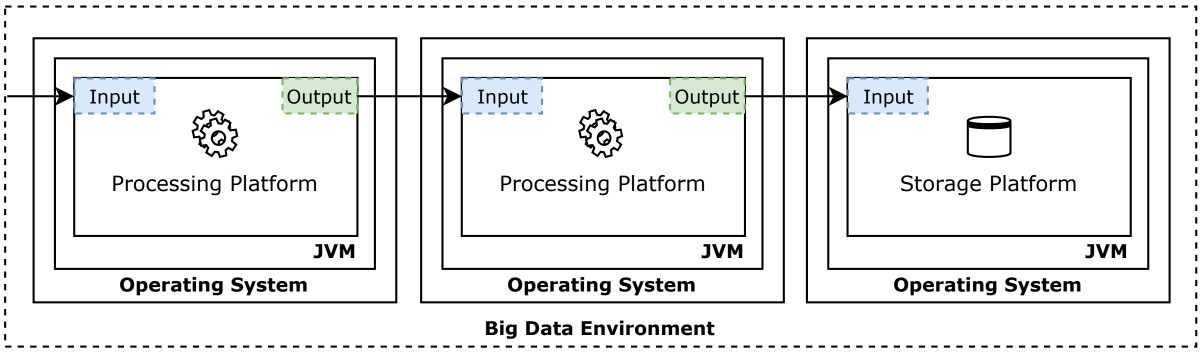


Figure 2.1: An example of a Big Data environment

To understand how garbage collectors perform, we need first to understand the environment in which they operate and how different implementations differ from one another. In this Chapter, we start with an overview of the different layers of the GC environment, notably the Big Data environment (Section 2.1) and the Java Virtual Machine (Section 2.2). Then (Section 2.3 and Section 2.4) we delve further into some particular components of the Java Virtual Machine, namely the heap and garbage collector, to understand the properties and techniques of the different GC implementations. To conclude, we provide a comparative overview of the various garbage collectors studied in this work (Section 2.5).

2.1 Big Data Environments

Big Data can be described as large data sets that may be analysed computationally to reveal patterns, trends, and associations, especially correlating to human behaviour and interactions. To handle such large amounts of data, specialised software tools are required, as previous software did not scale, in terms of performance metrics, to large data sets.

Analysing such data requires Big Data environments to be deployed, which refers to a group of one or more Big Data platforms, applications, and managed runtimes. Together, these are used to accomplish a particular task. Big Data platforms are processing or storing engines running on top of a managed runtime environment such as Java Virtual Machine, which are typically organised as a sequence, i.e., each platform takes as input, the output of the previous platform, as illustrated in Figure 2.1. Big Data applications refer to the user code executed by the engine inside Big Data platforms. Some platforms do not execute any user code, usually storage platforms, and on those we refer to the application as the platform itself.

There are different types of platforms, each with a specific purpose in the effort of extracting valuable information from vast volumes of data. Usually, we can designate a Big Data platform as being either a processing platform or a storage platform. On one hand, a processing platform, in its most generic form, is a system that receives data (from, e.g., another processing platform, a storage platform or directly from sensors), processes said data and generates an output; the destination of this output is either another processing platform, a storage platform or the final user. On the other hand, a storage platform can be generalised as a system that provides read and write operations to some managed data warehouse.

2.2 Java Virtual Machine Architecture

In this section, we provide further details regarding the relevant subsystems of a runtime system, more specifically the Java Virtual Machine. Figure 2.2 illustrates the high-level architecture of a runtime system, more explicitly the architecture of the OpenJDK HotSpot JVM, the most widely used industrial JVM implementation. Note that many other runtimes share most of the same elements and procedures, if not all. Additionally, Figure 2.2 presents a simplified architectural overview; the components that are not present in the figure are not relevant to the context of this thesis and consequently, mentioning them would only overcomplicate the architectural description.

Big data platforms and applications typically run on top of a runtime system, whose term is often utilised to refer a collection of managed resources and software required for the execution and operation of an application. The runtime primary function is the implementation of portions of an execution model for a given programming language. Interfacing both with low-level functions such as processor and memory management, runtime systems communicate with the software framework and libraries and may also debug, optimise and generate code while also preventing the execution of flawed code through type checking as high-level functions. Commonly, a runtime system may be a part of an operating system, but it can also be installed alongside a runtime environment such as Java Runtime Environment, which is the case with the Java Virtual Machine.

The Java Virtual Machine cannot directly interpret code written in Java. It must be compiled beforehand to an intermediate form of code, bytecode, a programming representation that uses an instruction set meant to provide efficient execution by a software interpreter. Since the instruction set used to create the bytecode is hardware agnostic, this characteristic is what enables Java to be hardware independent, provided that the system has a Java Virtual Machine. In order to compile a program from source code, e.g., Java, to bytecode, languages provide compilers such as the Java Compiler (javac). It is worthy to note that the Java Virtual Machine is not used to run Java applications solely; it can also run any programming language that can be compiled into Java bytecode, e.g., Python through Jython compiler.

Following the compilation of the source code and subsequent storage into a class file, the runtime system can then load and execute the bytecode, which at this point is an accurate representation of the original source code, without any optimisation. The class loader subsystem is the component responsible for loading the bytecode and preparing the necessary runtime data structures to execute it. These data structures are fundamental to support the execution of the program and are part of the Runtime

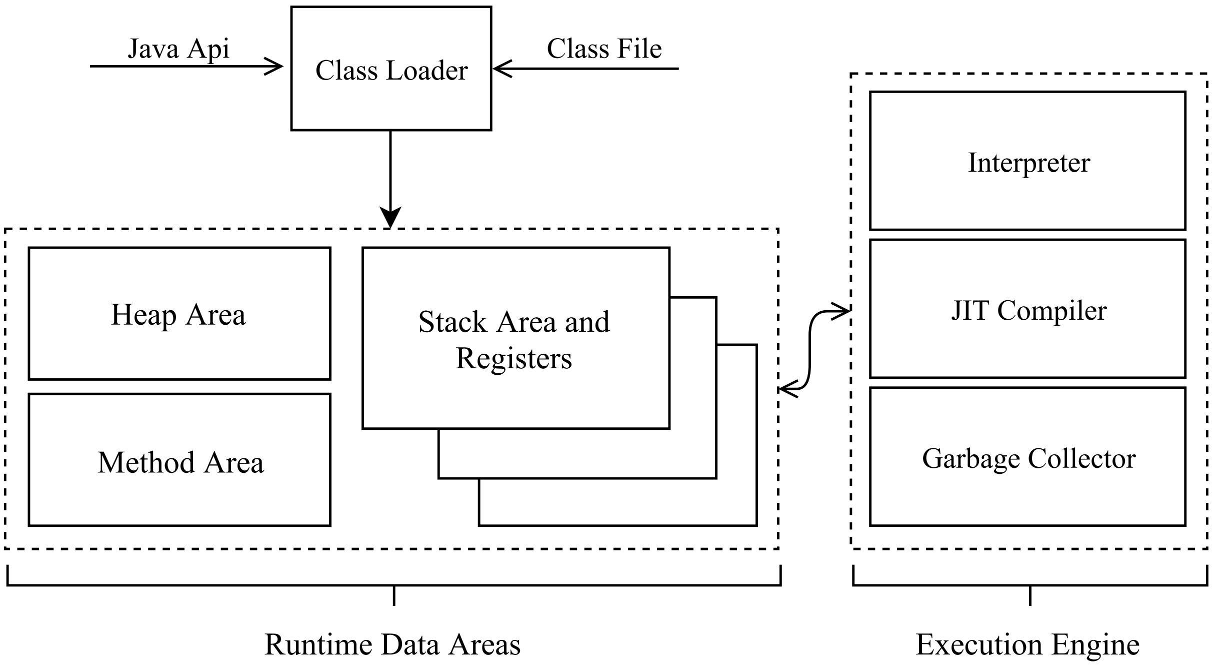


Figure 2.2: OpenJDK HotSpot JVM Architecture

Data Area, one of the main component groups in the runtime architecture. We now describe some of these data structures:

- Method Area – used to store all Class level data such as class name, immediate parent class name, methods, and others. Only one method area per Java Virtual Machine exists, shared across multiple threads;

- Heap Area – like the method area, the heap is also a shared resource. This component is where memory that is being used by a program is kept. It is used to store all the objects and their corresponding instance variables and arrays.

- Stack Area and Registers – these data structures are used to support the execution state of each thread. By keeping track of threads' state, it allows the runtime system to resume execution after performing operations such as garbage collection.

The remaining component group present in the architecture is the Execution Engine. Its components, with the support from the before mentioned data structures, are responsible for dictating the program's execution flow. In the Execution Engine, we focus on the following components:

- Interpreter – the Interpreter is a software component responsible for interpreting the bytecode and executing it line by line. Its primary disadvantage is that when one method is called multiple times, it requires interpretation every time, negatively affecting the application performance.

- JIT Compiler – the Just-in-time compiler counteracts the above mentioned disadvantage of the Interpreter. The JIT compiler compiles highly executed methods into a native, hardware-specific code, which is faster to execute (when compared to interpreted code). Additionally, with the help of the Interpreter, this component also performs optimisation on the native code such as null check eliminations, branch prediction, loop unrolling, method inlining, among others.

- Garbage Collector – this component is a part of the execution engine responsible for reclaiming the memory space occupied by unreferenced objects.

In summary, a runtime system serves to abstract specific responsibilities away from the developer, like memory management, code portability, and underlying optimisations. In order to achieve it, runtime systems, the OpenJDK Hotspot JVM, in particular, rely on the use of distinct runtime data structures coupled with a runtime engine containing some subsystems responsible for controlling the program's execution flow. In the following section, we go in depth on some of these data structures and subsystems, particularly the heap data structure and the garbage collector.

2.3 Memory Management

In this section, we start by describing some fundamental concepts (Section 2.3.1), required to understand how the garbage collector operates and how it affects an application. Then, we enumerate the different performance metrics by which collectors are compared (Section 2.3.2). We also present several different approaches, used by garbage collectors to identify unreferenced objects (in Section 2.3.3), followed by Section 2.3.4 where we discuss further optimisations to such approaches. In the last part (see Section 2.3.5) we discuss how partitioning the heap into multiple sub-heaps allows us to apply different GC approaches to each sub-heap, and what advantages this approach offers.

2.3.1 Background Concepts

As mentioned before (see Section 2.1) the heap stores all the application objects. From the JVM point of view, the heap is represented as a contiguous array of memory positions that are either occupied by objects or available for allocation by the user application (see Figure 2.3). An object is a sequential set of memory positions comprised of several fields. A field can either contain a reference or some other scalar non-reference value (for example, an integer). A reference is either a pointer to another object in the heap or the distinguished value null.

To better understand the links between the objects, the heap is usually depicted as a directed object graph where heap objects are represented as nodes, and the references stored in its fields as edges. In order to allow the tracing of the graph, the JVM holds special references, termed roots, that point to

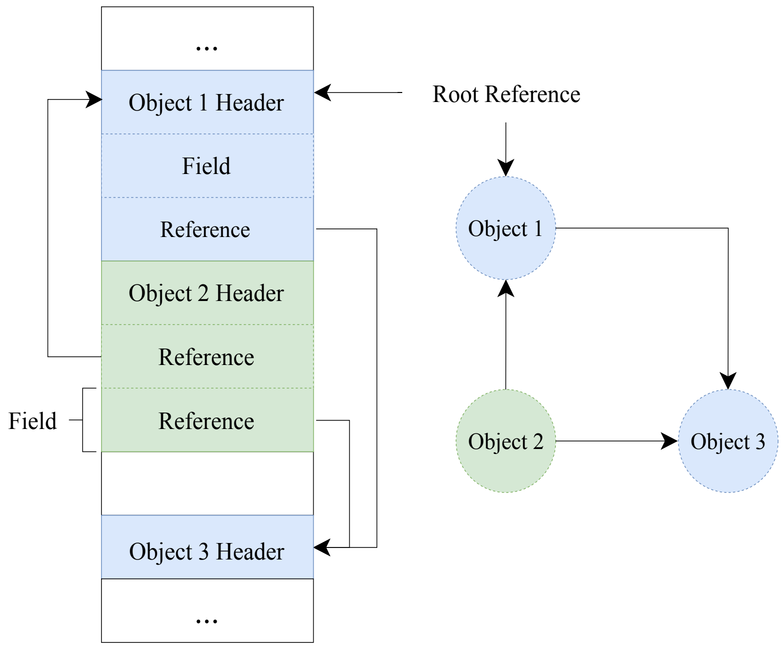


Figure 2.3: The heap (left) and corresponding object graph (right)

objects inside the graph, henceforth named root objects. In order to decide which objects are deemed for garbage collection, we first define each object in the object graph as reachable or unreachable (coloured blue and green in Figure 2.3, respectively). If an object is reachable and therefore live, it means that there is a path of references that reaches the object in question, starting from a root object. Alternatively, if no path reaches an object from any of the root objects, we consider the object dead and its memory safe for reclaiming.

According to Dijkstra [21], a garbage-collected program can be separated into semi-independent components, the mutator, and the collector. The mutator can be seen as representing a user application, which allocates new objects on the heap and mutates the object graph by changing those objects reference fields. As for the collector, it executes the garbage collection code, which is responsible for identifying unreachable objects and reclaiming their memory.

Due to concurrency within mutator threads, collector threads, and between the mutator and collector in concurrent garbage collectors, all collector algorithms require that specific code sequences appear to execute atomically. That is, other operations will appear to execute either before or after the atomic operation, but never interleaved between any of the steps that constitute the atomic operation.

2.3.2 Garbage Collectors Metrics

In this work, we discuss a broad range of garbage collectors, each devised with different workloads, hardware environment and performance requirements in mind. In this section, we point out the different metrics that can be considered when comparing the performance of such collectors.

Starting with pause time, this metric is used to measure how long an application must stop its execution so that the collector can execute. Typically, it is desirable that the time the application is stopped is as low as possible. Throughput can be seen as the sum of time available for the application threads to do their tasks seamlessly without getting blocked by the garbage collector. Space overhead can be seen as the sum of space required for the garbage collection process to perform. Different types of collectors impose different space overheads. For example, reference counting collectors may impose per-object space cost. Others, like the copying collectors, divide the heap into two partitions where the mutator can only work upon one of them, imposing per-heap space overhead. Regarding promptness, a garbage collector can be described as having high promptness if it takes a low amount of time to reclaim the memory used by an object after it becomes unreachable. Alternatively, the longer it takes to reclaim that same memory the lower the collector promptness is. Finally, scalability is the metric related to how an increase on the number of objects in memory influences the performance in any of the previous metrics. Ideally, an increase in the number of objects in memory should not indicate a decrease in performance.

2.3.3 Classic Garbage Collectors

Usually, GC implementations are designed around one of two different collection algorithms, reference tracing or reference counting. In this section, we describe the basis of each algorithm and how they differ from one another.

2.3.3.A Reference Tracing

As a collection algorithm, reference tracing consists in deciding which objects should be reclaimed by tracing reachable objects. It does so by following all sequences of references starting from the root objects, marking reachable objects along the way. This type of algorithm is deemed as performing an indirect collection because it identifies garbage as being all objects that were not marked during the tracing.

There are advantages to using reference tracing when compared to reference counting. In particular, reference tracing is complete, i.e., all garbage is eventually collected, even cyclic garbage, contrary to reference counting. A problem with the reference tracing approach is that it is not a deterministic algorithm because we are unable to predict when an object will be collected. Another problem is that the time spent tracing the object graph and marking reachable objects is proportional to the size of the entire heap, which is a concern, especially in Big Data applications.

2.3.3.B Reference Counting

Contrary to the algorithm discussed previously, reference counting is considered a direct collection algorithm as it identifies unreachable objects directly. As the name implies, reference counting keeps, for each object, the number of incoming references from other objects. If an object count reaches zero, then it has become inaccessible and can be reclaimed. Additionally, when an object is destroyed, all objects referenced by it have their counters decremented, potentially leading to further reclamation. It should be noted that these operations have to be atomic, as to not erroneously update counters.

In contrast to reference tracing, reference counting algorithms also present some compelling advantages: (i) the collector is responsible for decrementing object counters, so it promptly knows when an object is no longer reachable and therefore can be immediately collected; (ii) it is not dependent upon the total size of the heap, as the collections overhead is spread throughout the computation; (iii) there is no need to transverse the object graph for marking, which in turn leads to preservation of cache locality. However, it presents some problems such as (i) reference counting alone not being able to handle cyclic garbage because it keeps a positive counter even when the object is unreachable; (ii) the computation cost of having to update the counter for each object is expensive (mainly due to synchronisation operations and cache misses).

2.3.4 Design Choices

Nowadays, multiprocessors enjoy widespread commercial availability, therefore enabling the use of additional cores to cooperate on the garbage collection task, increasing applications performance. Multiple cores allow us to not only run multiple threads of both the collector and mutator but also to run both tasks concurrently with each other. Hence, by taking advantage of multi-core architecture, the before mentioned approaches to identifying unreachable objects, tracing and reference counting, can further be optimised with the design choices now described.

2.3.4.A Serial versus Parallel

A serial approach would be to use only one thread to interchangeably run the user application (mutator) and the collector. In contrast, by taking advantage of additional cores, we can run multiple threads of both the mutator and collector. For example, in reference tracing, traversing an object graph can now be done in parallel (multiple threads) increasing the speed, but it also requires a more careful implementation due to complex concurrency issues.

2.3.4.B Stop-the-World versus Incremental versus Concurrent

A stop-the-world approach means that in order to perform garbage collection, the user application must halt its execution. This means that all mutator threads must be stopped until such a periodical collection is over. Not only is this implementation the simplest, because there is no need for synchronisation between the mutator and collector, but it is also the best option for throughput-oriented applications because it does the collection in only one step, which allows the application to work at full speed the rest of the time.

However, if low latency requirements are our target, then an incremental or concurrent garbage collector is a more suitable choice. An incremental GC performs collection in steps, e.g., per memory page, per sub-heap, per sub-graph, which indeed decreases individual pause times but might overall require more time to collect all garbage. As for the concurrent approach, it allows both mutator and collector to run at the same time, allowing for lower pause times, but not without its tradeoffs. The necessary synchronisation between the mutator threads and collector threads is a significant source of memory overhead compared to other approaches.

2.3.4.C Non-Moving versus Compaction versus Copying

When an application first starts, a certain amount of memory in the heap is made available for allocating the application objects. However, over time, as the collector reclaims objects the memory space still available for allocation, becomes fragmented into smaller and smaller contiguous spaces, causing severe problems such as: (i) application objects being scattered through the heap; (ii) unable to allocate objects that do not fit in any fragmented space; (iii) as time passes more and more small fragments of memory space exist between objects, meaning that the total amount of memory used by the application is higher then what it actually needs.

Two solutions can be applied to solve the previous problems, compaction and copying. Compaction is frequently used to relocate all reachable objects to the start of some memory segment, e.g., memory page. The number of memory segments, e.g., memory pages, before and after compaction remains the same, allowing for segments with few live objects to exist. Alternatively, copying moves reachable objects from one memory segment to another, allowing for grouping of objects from multiple memory pages into a single page. Even though it requires more memory to perform, it eventually frees the pages from where the objects were copied.

2.3.5 Partitioned Algorithms

Until now, we have assumed that one and only one garbage collector does the garbage collection process and is responsible for managing all objects. However, it is often useful to partition the heap into multiple partitions/sub-heaps and apply a different garbage collector approach on each one, especially when a cluster of objects share a characteristic more suited to be managed by a different garbage collector. Bishop [22] was the first to explore the idea of heap partitioning in his influential thesis, and since then multiple models have been proposed. Examples include partitioning by mobility where it may be necessary to distinguish objects that can be relocated from those that can not or are too costly to move. Recently created objects tend to be modified more often than longer-lived objects [23–26]. Hence it may be efficient to partition objects by their tendency to be modified using partitioning by mutability. Other examples are partitioning by availability, partitioning for locality, etc. Due to the scope of this thesis, we focus on the most used partitioning approach, partitioning by yield, which is used to separate objects based on their estimated lifetime. This approach bases itself on studies supporting that Java object's lifetime follows a bimodal distribution [27, 28] and the weak generation hypothesis, saying that on most applications, most objects die young [29, 30]. The most common usage of partitioning by yield are generational collectors. As we mentioned before, most application's objects tend to die young. A simple example that often occurs is a method creating many objects but never storing them in a field. When the method exits, those objects are ready to be collected.

As every object is initially allocated on the young generation sub-heap, as time passes, objects that keep surviving collection cycles are eventually promoted to another sub-heap, called the old generation. Since the number of live objects in the young generation takes up a small percentage of the available space, using a copying collector makes sense, because we have space to place the live set and the work required to move them to the old generation is linear to its small size. However, as objects get moved to the old generation, newly created objects in the young generation that are referenced by those in the old are invisible to the young generation collector, therefore considered garbage. This is an incorrect assumption, as these objects are still reachable from the objects in the old generation, and therefore could still be live. In order to solve this problem, a card table and a remembered set are used.

If every time a young generation collector is executed, it had to look at the whole heap to determine which objects were dead, it would be too costly and in contradiction with the idea of a generational collection. Instead, a remembered set data structure is used to track outside (old generation) references to objects in the young generation. The remembered set holds for each object in question a pointer to a card table element. The card table is a data structure where each element represents an old generation region (see Figure 2.4). It can be seen as an array of 0s and 1s, where a 0 element indicates a region with no pointers to the young generation and one otherwise.

Every time the young generation collection executes, in addition to tracing the root objects of the young generation, it also scans the card table to identify regions with objects pointing to the young generation and traces them to identify which young objects are still reachable. The mechanism used to update these data structures, a write barrier, is described in Section 2.4.

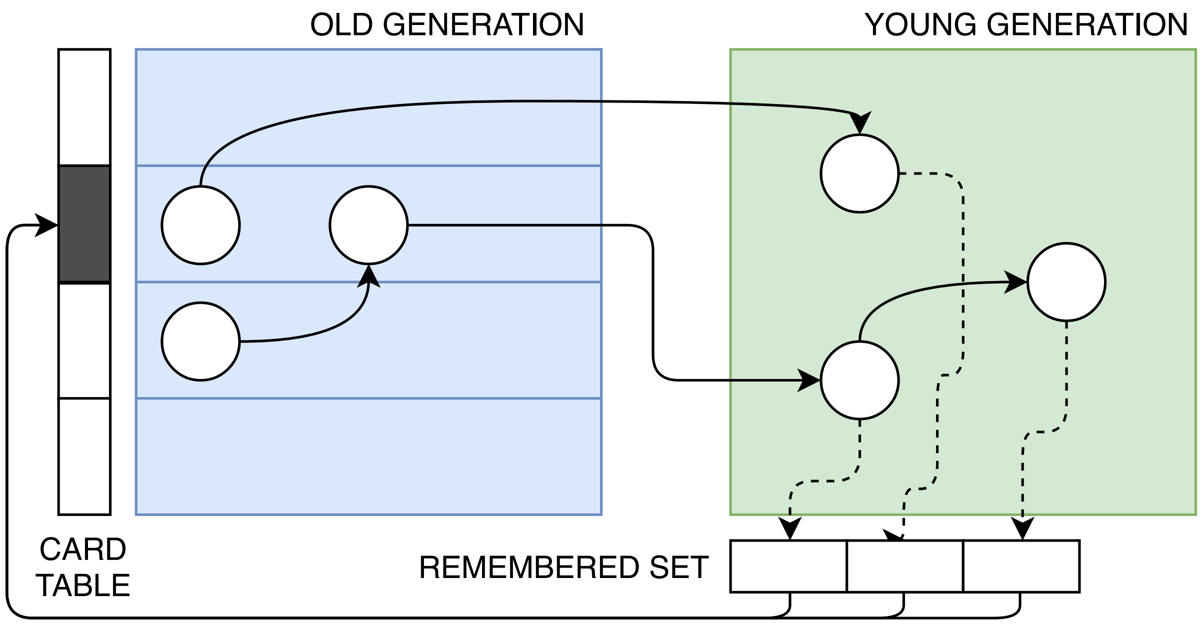


Figure 2.4: Card table and remembered set data structures representation example

2.4 Barriers

There are certain operations in garbage collectors algorithms, e.g., reads or writes of the mutator, that require specific additional actions to ensure a correct garbage collection implementation. Thus, the use of barriers is a way for collectors to know that a specific operation over a memory address has to be handled differently.

Usually, garbage collectors are not fully concurrent; typically, they are mostly-concurrent algorithms. These algorithms need to ”stop-the-world” to perform compaction, meaning that the mutator threads need to stop allocation so that the collector can update the references to recently moved objects. The use of read barriers [31] and write barriers [32] to synchronise both mutator and collector threads have been a common approach in concurrent garbage collectors. Current implementations take the form of inlined code, added by the compiler to assure the integrity of the memory management. Due to the frequency of data accesses in Big Data applications, the associated overhead imposed by software-based barriers is a significant performance overhead, requiring an efficient implementation.

2.4.1 Write Barrier

Every time a field of an object is updated or a new reference is stored, a write barrier is triggered by the mutator so that synchronisation may happen between the collector operation and the mutator access. Due to the variety of garbage collectors architectures, various write barriers implementations exist, each one tailored in some way to the collector architecture as shown by Blackburn et al. [33] and Yang et al. [34].

Write barriers are used not only for synchronisation between the collector and mutator threads; one such case is the use of write barriers for updating the card tables in generational collectors [35]. However, when referring to write barriers implementations in the subsequent sections, it should be regarded as the barrier responsible solely for the synchronisation between collector and mutator.

2.4.2 Read Barrier

Whenever the mutator loads a reference, e.g., loading an object field, a read barrier is triggered by the mutator. There are two variants of read barriers, read conditional and read unconditional [33, 34].

As mentioned before in Section 2.4, barriers require efficient implementations to minimise their impact on application performance. Therefore, there are usually two paths that can be applied when entering a conditional read barrier. The so-called fast path is selected when no synchronisation between mutator and collector is needed and therefore only comprised of a couple of lines of instructions that barely affect performance. On the other hand, the slow path is selected when some synchronisation is needed, and the performance cost associated is usually significant. Unconditional barriers, on the other hand, are generally used to apply a specific action to all read operations unconditionally, e.g., mask low-order address bits.

2.5 Garbage Collectors

All of the garbage collectors compared in this study (see Figure 2.5) are reference tracing algorithms. The reason for this is primarily due to the problems associated with reference counting, i.e., reference counting needing additional techniques to be a complete algorithm which is computationally costly, and reference counting requiring a write barrier in all reference modification instructions, which incur an extra overhead on performance. Therefore, reference counters algorithms are not utilised in most production JVMs.

2.5.1 ParallelOld GC

The ParallelOld GC [36] is a two-generational (see Section 2.3.5) parallel ”stop the world” garbage collector, which means that whenever a garbage collection occurs in either generation, all application threads are stopped, and the GC work is performed using multiple threads, as illustrated in Figure 2.6. This approach is usually the most efficient way to maximise the time spent doing application work

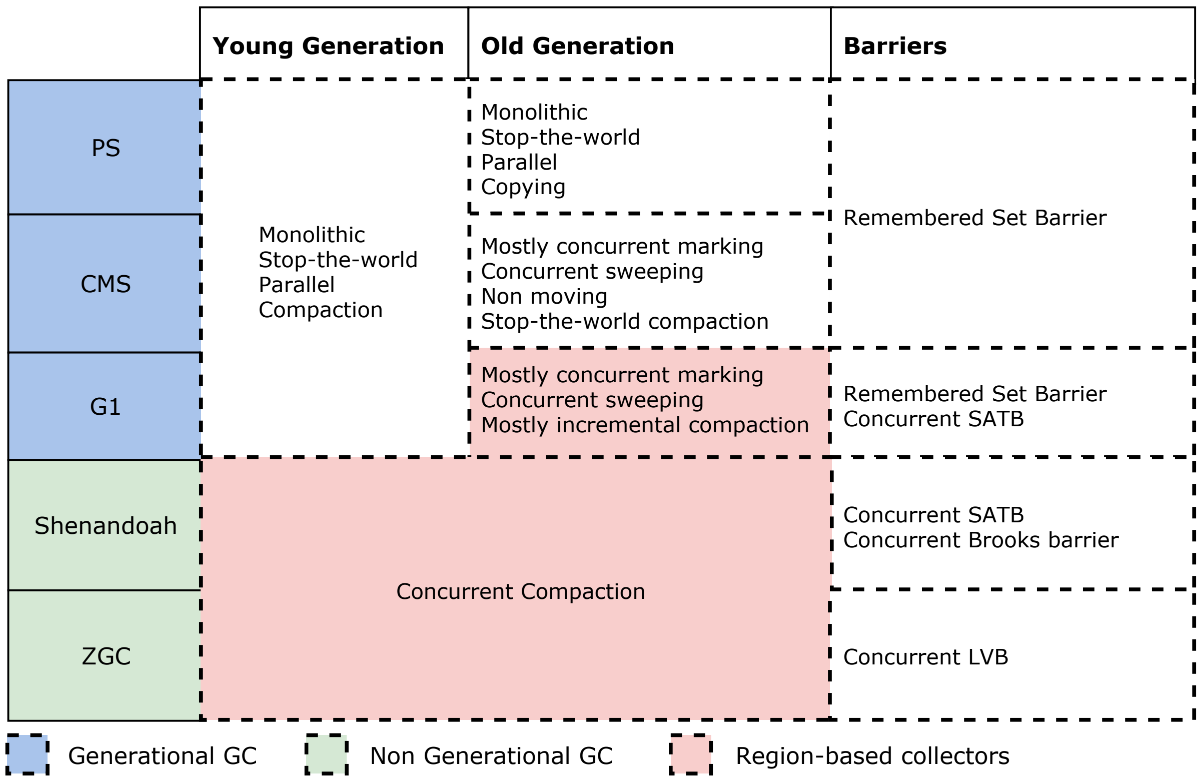


Figure 2.5: Illustrates the different garbage collectors design choices.

relative to the total time spent performing garbage collection. However, there is an overhead inherent to this approach in the form of long individual GC pause times caused by the monolithic ”stop the world” collections.

A monolithic ”stop-the-world” copying (see Section 2.3.4.C) collector manages the young generation of the ParallelOld GC. As for the management of the old generation, a parallel ”stop the world” garbage collector with compaction is adopted. Compaction (see Section 2.3.4.C) moves objects closer together, improving space locality and reducing fragmentation caused by promotions from the young generation to the old generation during collection. However, along with ”stop the world” collections, the overhead of performing compaction (usually a function of the size of the Java heap and the number and size of live objects in the old generation) is also significantly responsible for causing long individual GC pauses. As the heap size increases, so does the cost of performing ”stop the world” collections and compaction, which is significant problem with scalability in using the ParallelOld GC in Big Data environments.

2.5.2 CMS

The Concurrent Mark/Sweep collector [37] is a generational garbage collector. CMS was developed in response to a growing number of applications that demanded a collector with lower worst-case pause

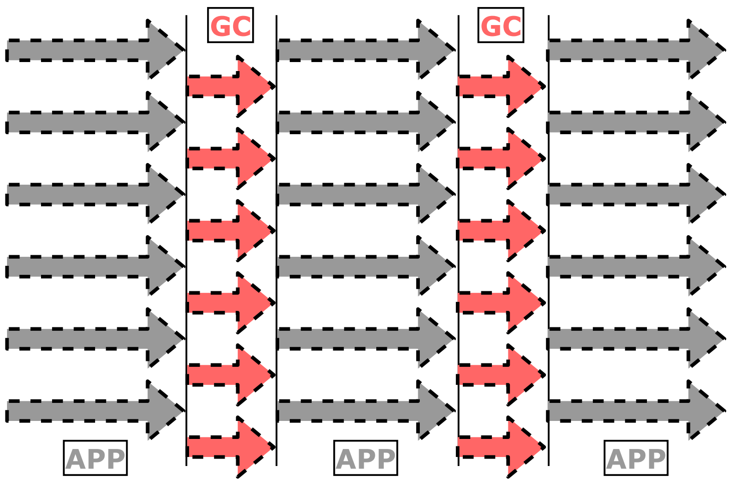


Figure 2.6: Illustrates how the Java application threads (grey arrows) are stopped and the GC threads (red arrows) take over to do the garbage collection work.

times than Parallel GC and where it was acceptable to forgo some application throughput to eliminate or considerably reduce the number of lengthy GC pauses. Like the Parallel collector, CMS uses a monolithic (the whole heap must be collected at once) ”stop-the-world” copying collector to manage all the objects in the young generation.

The main difference between the Parallel GC and CMS GC is that the old generation is managed by a mostly concurrent mark and sweep collector without compaction. It is called a mostly concurrent collector because most of its work is done concurrently with the application threads, except for a couple of phases of the old generation collection which require the halt of the application threads for synchronization purposes.

The CMS pauses the application twice during the concurrent collection cycle. The first pause occurs when performing the initial mark, which marks as live the objects directly reachable from the root objects and elsewhere in the heap (e.g., young generation). The second pause occurs after the concurrent marking phase concludes to discover objects that were missed by the concurrent tracing due to updates by the application threads of references in an object after the CMS collector had finished tracing that object. This second pause is referred to as the remark pause. An illustration of the CMS concurrent collection cycle is shown in Figure 2.7.

A young generation collection may happen while an old generation concurrent collection is taking place. When this situation occurs, the old generation concurrent collection is halted by the young generation collection and promptly resumes upon the latter’s conclusion. When objects can no longer be promoted to the old generation, or concurrent marking fails, it falls back to a monolithic ”stop-the-world” compaction of the old generation. This compaction requires tracing the whole old generation with the application threads on halt, which will be the main responsible for the CMS collector’s lengthy pause

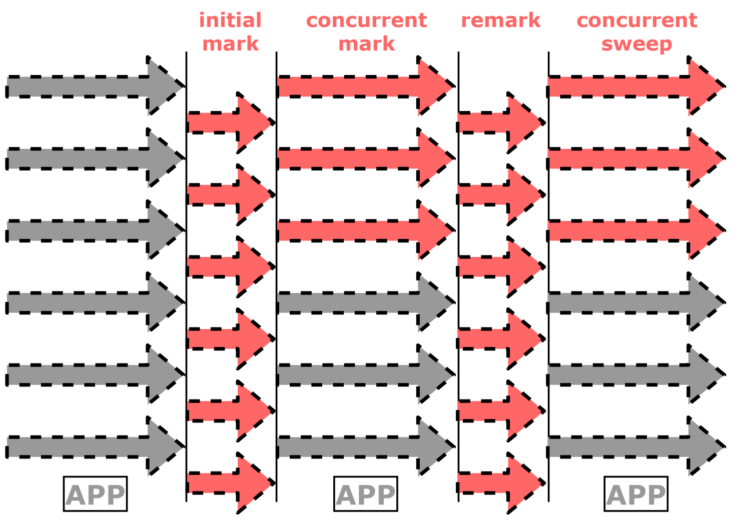


Figure 2.7: Illustrates how the Java application threads (grey arrows) are stopped to allow the two ”stop the world” pauses. And how Java application threads and GC threads work concurrently otherwise

times. A full collection is triggered when unable to finish reclaiming the unreachable objects before the old generation fills up, or if an allocation cannot be satisfied with the available free space blocks in the old generation.

2.5.3 G1

The Garbage-First [38] is the current default collector for OpenJDK Hotspot JVM and is one of the most widely used garbage collectors. G1 is a two-generational garbage collector that follows a different approach compared to the Parallel and CMS GC to address some of the shortcomings with those collectors. Instead of having the young and old generation be a contiguous chunk of memory where garbage collection is monolithic, in G1 both the young and old generations are a set of regions where most GC operations can be applied individually to each region. Also, regions that belong to the same set and therefore same generation do not need to be contiguous in memory. Figure 2.8 illustrates how G1 divides the Java heap.

For the young generation, similar to the previous collectors, G1 employs a parallel ”stop the world” copying collector which collects all regions belonging to the young generation region set (monolithic). For the old generation, on the other hand, G1 does not require the whole generation to be collected. Instead, just a subset of the old generation region set is collected at any one time during a mixed collection, using a mostly concurrent mark and sweep collector. A mixed collection is a young generation collection, where the subset of the old generation region set chosen is also collected together. Using a full heap trace allows G1 to track the amount of garbage in each region accurately and therefore, preferentially target regions that will yield the most garbage. In addition, incremental compaction is employed by G1, which

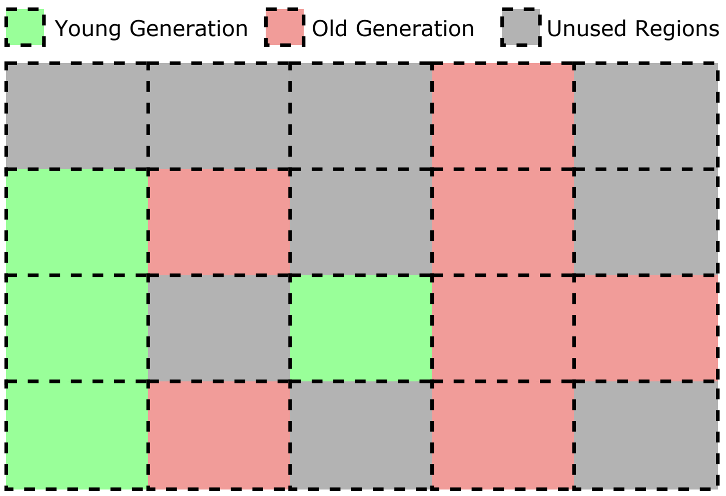


Figure 2.8: Illustrates how the G1 divides the Java heap. The set of green regions represents the young generation, and the set of red regions represents the old generation. The unused regions (grey) can be used by either the young or old generation by adding them to their corresponding set of regions.

means that on every old generation collection, all objects in the subset of regions being collected are relocated to unused regions. Regions that have all their reachable objects relocated (due to incremental compaction) become unused regions, which later can be used by either generation.

When objects can no longer be promoted to the old generation, it falls back to a monolithic ”stop-the-world” compaction of the old generation. However, with the incremental compaction and with sufficient tuning, G1 is designed in such as way that a full garbage collection being required should not occur. Nonetheless, this requires some application profiling to best tune the garbage collector to the application needs.

In addition to a memory barrier to update remembered sets, G1 also employs a snapshot-at-the-beginning (SATB) barrier. During a concurrent marking phase, an object initially believed to be garbage may become reachable due to a new allocation; therefore, a mechanism (SATB) that avoids the collection of these objects is employed. This mechanism is an additional hook that gets called whenever mutator threads write references field in objects during concurrent marking so that these objects may be marked as live after the marking ends.

Splitting the Java heap into regions and performing collection with incremental compaction on just a subset of the old generation reduces the lengthy pause times that were present in Parallel and CMS. However, this originates a significant overhead in memory usage. Due to having the old generation split into regions, a remembered set between each region is now required, which may result in an overhead of up to 20% on memory usage [39] when compared to previous collectors.

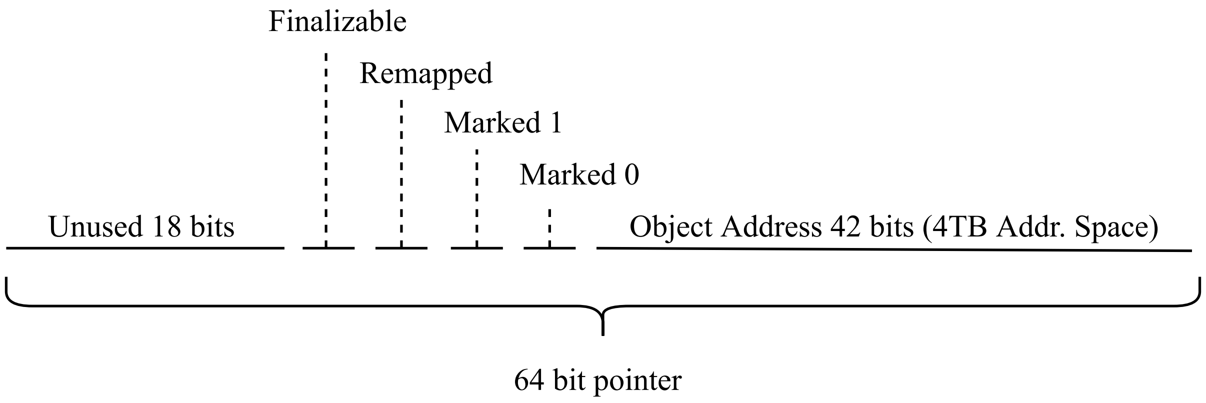


Figure 2.9: Coloured pointer

2.5.4 ZGC

The Z Garbage Collector, also known as ZGC, is an experimental scalable low-latency collector that is built to handle heaps varying from relatively small to potentially multi-terabytes sizes. Also, ZGC's pause time is supposed not to exceed 10ms even when increasing the heap or live-set size.

Compared to the Garbage-First collector, ZGC is also a region-based collector; however, it is not generational. It improves upon G1 by achieving concurrent compaction with the introduction of two core techniques, read barriers and coloured pointers. The coloured pointer is a technique that uses 4 of the 22 unused bits of a 64-bit reference to store some important metadata. As shown in Figure 2.9, the first 42-bits of an object reference are reserved for the actual address of the object, which gives a theoretical heap limit of 4TB address space. From the remaining unused bits, four of those are used as flags named finalizable, remapped, marked1 and marked0:

- finalizable - The bit is set if the object is only reachable through a finalise method.
- remapped - The bit is set if the reference points to the current address of the object. When the collector is performing a concurrent relocation, and an object is loaded by the application, a read barrier is triggered when loading the reference from the heap. The application will first check if the remapped bit is set. In case it is, it means the reference points to the current address of the object and the reference to the object is returned. Otherwise, it checks if the object is in the relocation set. If the object is not in the relocation set it means the reference points to the current address, but the remapped bit is yet to be set, so the application sets the bit and returns the address. In case the object is indeed in the relocation set, the application checks if the object has already been relocated or not. If it has then the reference is updated to the new current address of the object and returned. In case it has yet to be relocated, the application threads relocate the object and return the updated reference.

```
//  +--------------------------------+ 0x0000140000000000 (20TB)
//  |          Remapped View         |
//  +--------------------------------+ 0x0000100000000000 (16TB)
//  |     (Reserved, but unused)     |
//  +--------------------------------+ 0x00000c0000000000 (12TB)
//  |          Marked1 View          |
//  +--------------------------------+ 0x0000080000000000 (8TB)
//  |          Marked0 View          |
//  +--------------------------------+ 0x0000040000000000 (4TB)
//  .                                .
//  +--------------------------------+ 0x0000000000000000
```

Listing 2.1: Address Space and Pointer Layout [10]

- marked0 and marked1 - These are used to flag reachable objects. When the relocation phase concludes there may still be references that need to be remapped and hence have still one of the marked bits set from the last marking cycle set. If the subsequent marking phase used the same marking bit, the read-barrier would see this reference as already marked, incorrectly. Therefore, the marking phase alternates between using the marked0 and marked1 bit in each cycle.

When a read barrier is invoked by loading a reference, there are a few assembly instructions that need to be executed. Due to the high frequency of reads and writes in an application, both write and read barriers need to be extremely efficient as not to cause large performance overheads. Even though ZGC does not make use of write barriers, it uses read barriers called load-value barriers (LVB) to do concurrent compaction. In addition to the overhead caused by the read barrier, there is a cost associated with using coloured pointers, more specifically due to the need of dereferencing the object address from the pointer. To avoid this cost, when allocating a page, ZGC maps the same page to 3 different addresses, corresponding to the original address plus an offset caused by precisely one of the flags being set. This approach works because when reading from memory precisely one bit of marked0, marked1 and remapped is set. An illustration of the memory mapping used by ZGC can be seen in Listing 2.1.

2.5.5 Shenandoah

The Shenandoah garbage collector, like the ZGC, has the goal of reducing pause times on large heaps. It is also a region-based non generational collector that uses similar principles to ZGC but follows a different implementation strategy. Instead of coloured pointers, Shenandoah makes use of Brooks pointers, for allowing concurrent compaction of memory. The main idea behind a Brooks pointer is that each object has an additional reference field that always points to the current location of the object. The referenced location can either be the object itself or, as soon as the object gets copied to a new location,

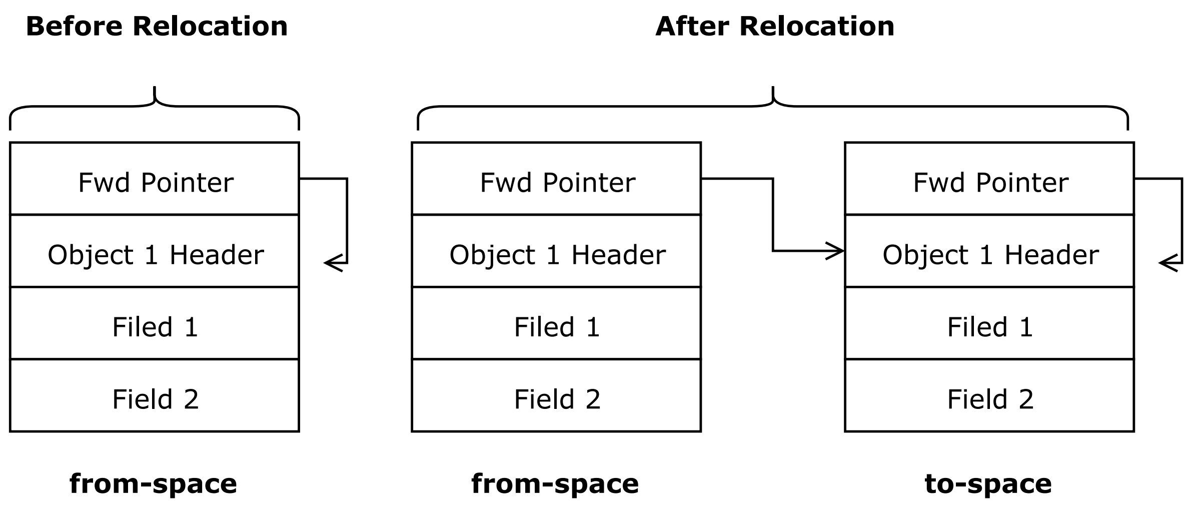


Figure 2.10: Brooks pointer example

to that new location.

During compaction, an object that is set to be relocated must be copied from the ”from-space” to the ”to-space”. The from-space, as the name implies, is the original location of the object, and the to-space is the destination of the object after the copying. A classic ”stop-the-world” compaction would then stop application threads so it could update all references to the old ”from-space” object, to the current ”to-space” reference. However, with Brooks pointers, we no longer need to stop the application to update all references leading to the ”from-space” object. Every object now has an additional reference field (forward pointer) that points to the object itself, or, as soon as the object gets copied to a new location, to that new location (as depicted in Figure 2.10). To assure that all writes are done on the ”to-space” object, a write barrier is triggered for all writes. This write barrier is responsible for dereferencing the forward pointer on the old object, to reach the current location of the object. Additionally, if during a copying phase a write barrier is triggered on a ”from-space” object that is set to be copied but has yet to be, the procedure is as follows: i) creates the ”to-space” object; ii) updates the ”from-space” forward pointer; iii) writes the value in the ”to-space” copy; iv) updates the reference that triggered the barrier to point to the new location. Eventually, when the copying phase terminates, references that still point to ”from-space” objects that were copied are updated during concurrent marking. When all references are updated, the ”from-space” object is collected.

This technique introduces some overheads, e.g., memory overhead caused by the forward pointer (usually one word per object), more instructions to verify that writes and reads are always done in the ”to-space” object, and the high possibility of cache misses due to pointer-chasing indirection. Furthermore, contrary to the ZGC collector, Shenandoah claims to work exceptionally well even in small heaps.

3

Related Work

Contents

In this Chapter, we delve into the current state of research related to studying and evaluating Garbage Collection for Big Data Environments. We split the focus of our study into two main categories: i) prior studies on garbage collectors performance when employed to Big Data Environments (Section 3.1); and ii) current solutions to the selection of application-specific garbage collectors (Section 3.2).

3.1 Garbage Collectors Performance in Big Data Environments

We start by presenting previous studies on Java garbage collection, where garbage collectors handled in this study or similar are examined on top of a Big Data environment.

3.1.1 R. Bruno et al. [1]

Presents a study of current Big Data environments and platforms focusing on how garbage collectors behave in these environments. The study provides insight into the state of the art of memory management for memory managed runtime environments; it does so by first analysing traditional garbage collectors scalability problems when applied to Big Data platforms. The authors present the following problems.

First, the need to trace the entire heap while using tracing algorithms to identify garbage can lead to memory exhaustion if the mutator allocates memory very aggressively. Therefore, triggering a full collection due to a large number of live objects in the heap. A full collection can take dozens or even hundreds of seconds to collect all objects in memory [40], consequently causing a severe impact on throughput and pause time for the application. In generational garbage collectors, the previously mentioned assumption that most objects die young is not valid for a wide range of Big Data platforms [41–43]. Second, the high fragmentation in the old generation leads to decrease locality and possibly situations where memory can no longer be allocated due to fragmentation, even though there is still enough space. As for reference counting algorithms, being not complete, they require additional techniques to achieve completeness, which comes at a high computational cost (reducing an application throughput). Additionally, reference counting algorithms require a write barrier in every reference modification instruction to update reference counters [44], which incurs into additional overheads on latency and throughput.

The study then shows how several solutions that have been trying to address these scalability challenges, such as:

- Broom (Gog et al. [45]) uses region-based memory management as a way to reduce the cost of managing massive amounts of objects usually created by Big Data Applications;
- FACADE (Nguyen et al. [46]), a compiler framework for Big Data platforms that modifies Big Data platforms to use programmer-managed native memory (off-heap) instead of the GC-managed memory (on-heap);

- Deca (Lu et al. [47]) that uses lifetime-based memory management so that objects are grouped according to their expected lifetime;
- NumaGiC (Gidra et al. [42]) that proposes several mechanisms to improve reference tracing locality to improve applications' throughput;
- DSA (Cohen and Petrank et al. [43]) tries to alleviate the GC overhead on the throughput performance by specifying which classes are data structures so that they can be allocated together and therefore improve its spatial locality.
- Taurus (Maas et al. [48]), a runtime system for coordinating Big Data platforms running across multiple physical nodes;
- Garbage-first collector presented in Section 2.5.3 and the Continuously Concurrent Compacting Collector;
- N-Generational Garbage Collector (Bruno et al. [41, 49, 50]) presents the idea of extending the well-established two-generational heap layout into an N-generational heap layout to eliminate the need for promotions and compactions.

To conclude, the authors present insights into possible new research opportunities to solve issues/challenges that remain untackled, such as: i) research on how to trace objects efficiently in reference tracing algorithms; ii) also related to the previous point, the need for a cache friendly collector, which either minimises or removes the effect of tracing objects by the collector on the application' cache locality; iii) current collectors implementation when promoting objects, do not take into consideration the distance in memory between objects that point to each other. For example, a data structure such as a list of objects, where each object has a reference to the next one in the list, may have each object moved to a different page in memory. When loading this list, the number of pages to load would be proportional to the number of objects in the list. However, if GC implementations had this distance between objects into consideration, the number of pages would be reduced, due to objects being close to each other in memory; iv) improved APIs that interact with the garbage collector would promote advanced programmers to interact with the collector to help make the application more predictable and controllable.

3.1.2 Xu et al. [2]

Presented a comprehensive experimental evaluation of three commonly used garbage collectors, i.e., Parallel, CMS, and G1, using four typical Big Data Spark applications. The primary purpose of this experiment consists of analysing the correlation between big data applications' memory usage patterns and the collectors' memory usage, to obtain findings regarding GC inefficiencies.

From all findings stated in this study, those related to CMS and G1 seem to be the most interesting ones in the scope of this work, as the evaluation of Parallel GC with a serial collector managing the young generation is not in the scope of this thesis. The authors concluded that: i) by allocating abundant old space without shrinkage, CMS tasks achieve approximately 48% less full GC pauses than G1 tasks; ii) G1 tasks use 1.1-1.2x higher physical memory than CMS tasks due to the need to allocate remembered sets for keeping object information used for GC; iii) For applications with humongous data objects, G1' non-contiguous region-based heap management has heap fragmentation problems that lead to out of memory errors; iv) CMS concurrent sweeping algorithm delivers 16x shorter full GC times than G1' incremental sweeping algorithm for iterative applications that require to reclaim long-lived objects.

3.1.3 Yu et al. [3]

Presents a performance analysis on the overall performance impact of Full GC in memory-hungry applications that handle large data sets, more particularly when using the Parallel Scavenge (PS) garbage collector. The Parallel Scavenge is technically very similar to the Concurrent Mark and Sweep (see Section 2.5.2), the main difference being that the former is not concurrent in any way and the young generation is managed by a serial collector. Detailed profiling uncovered that redundant accesses and calculations when updating references while compacting are the leading causes of lengthy full GC, due to the need to query two globally-preserved bitmaps. The bitmap search starts from a specific region start as a reference is updated, which potentially leads to a lot of unnecessary accesses that collectively can cause up to 70% of full GC time.

An incremental query scheme is presented to reduce the number of needed accesses to find the queried object, that relies on the critical observation on Parallel Scavenge that each query is initialised from a specific region start and confined to that region. Therefore, if two sequential queries are in the same region, the previous query might be closer to the current query object location than the region start, hence shortening the number of accesses needed to reach the object. To conclude, three optimising schemes are proposed to be used based on the locality of query patterns. Based on a set of experiments using both standard benchmarks like JOlden, Dacapo, SPECjvm2008 as well as world big-data frameworks like Spark and Giraph, the proposed incremental query scheme achieves up to 3.4x speedup in full GC throughput and 57.2% and 41.2% improvement in application throughput on a Xeon CPU server and a Xeon Phi coprocessor accordingly.

3.1.4 P. Pufek et al. [4]

Analyzes several garbage collectors (i.e., G1, Parallel, Serial and CMS) available in JDK 11 by benchmarking each one with the DaCapo benchmark suite, comparing the number of algorithms' iterations and the duration of the collection time. The experiments were performed within JDK 8 and JDK 11, as well as the OpenJDK 12 early-access build. During the benchmarking within JDK 8, it was shown that contrary to expectations, G1 did not perform better in most benchmarks. However, significant enhancements in performance were noticed with the newer JDK version 11, which can be supported by the introduction of parallel full garbage collection. According to the authors, G1 operated better or in an equal manner as Parallel GC in h2, pmd, xalan, and tradebeans considering the overall duration of collections in the latest Java release. On the other hand, CMS and Serial never showed any assured superiority over G1 and Parallel GC. This led to conclude that with an improved G1, region-based garbage collectors indeed represent the direction in which future GCs are heading towards.

In addition to a comparison between collectors, the authors also discuss how collectors such as Shenandoah and ZGC, who are still experimental and under development, compare to the current default JDK collector Garbage-First in terms of implementation decisions, which were also explained in this thesis in Section 2.5.

3.1.5 A. Prokopec et al. [5]

Propose a new benchmark suite called Renaissance, which is composed of modern, real-world, concurrent, and object-oriented workloads that exercise multiple concurrency primitives of the JVM. In addition, a performance comparison between two state-of-the-art, production-quality JIT compilers (i.e., Hotspot C2 and Graal) is performed to show that performance differences in this suite are more significant than on existing suites such as Dacapo and SPECjvm2008 (see Section 4.1). Even though this evaluation is focused on compilers, it is still relevant as these suites can also be used to evaluate other JVM components such as the garbage collector. We decided to use Dacapo in our evaluation in this thesis because it has been a de facto standard for JVM benchmarking. However, due to the complex object and memory behaviour shown with the Renaissance benchmark suite, a future GC evaluation with this suite would be interesting.

3.1.6 W. Zhao et al. [6]

The Garbage-First (G1) shares algorithmic foundations with other prominent contemporary collectors (i.e., ZGC, Shenandoah and C4). However, the design of the core algorithms and the performance tradeoffs they manifest have not been carefully analyzed in the literature. This is a problem that we try to help address with this thesis. In this paper, the authors also undertake the task of helping solve this problem by deconstructing the G1 algorithm and re-implement it from first principles to evaluate the impact of each of the significant elements of G1 on performance (pause time, remembered set footprint and barrier overheads). This performance evaluation is made within the JikesRVM using the Dacapo and SPECjvm98 [51] benchmark suites and the pjbb2005 [52] benchmark.

The authors found that the introduction of concurrent marking and generational collection reduced the 95-percentile GC pause times by 64% and 93% respectively. Also, they found that the memory space overhead of G1's remembered sets was typically under 1%. This finding may not be accurate for a broad range of applications and may represent a lack of benchmarking range, as Hunt et al. [39] found that G1's remembered set overhead could be up to 20% on memory usage. To conclude, the authors found that the barriers in G1 (e.g., remembered set barrier (see Section 2.4.1) and STAB barrier (see Section 2.5.3)) had an overhead of approximately 12% concerning total performance.

3.2 Selection of Application-Specific Garbage Collectors

The following works each represent a different approach to the selection of application-specific garbage collectors. Dynamic selection [8] is an approach that picks several suitable GC algorithms, and switches between them at runtime when certain heuristic thresholds are met, whereas static selection [7] chooses a single most optimal GC for the entire application execution. Still, both approaches require exhaustive profiling to determine which collectors are best suited for a given application. The final work presented in this section [9] presents a method to reduce the number of profiling runs drastically by resorting to machine learning.

3.2.1 R. Fitzgerald et al. [7]

Based on the premise that it can be challenging or even impracticable for an environment to substitute its garbage collector during runtime, especially if performance-critical components of the implementation such as allocation sequences and write barrier checks have been inlined into executable code. However, the use of a single garbage collector can hurt the application performance if the GC is not suited to handle the type of requisites and access patterns of a particular application [53]. This study investigates the value of pairing applications with suitable collectors.

Benchmarks from the suites SPECjvm98, MiscJava, Impact, and Doctor are combined with each of several garbage collectors in an environment that builds standalone compiled executables. The garbage collectors used are a 'null' collector, which allocates objects but never collects them, a generic copying collector, and a generational copying collector. To further evaluate the generational copying collector, it is coupled with each of 4 different write barriers. Those write barriers can be divided into two types, non-filtering barrier that registers all pointer stores, and filtering barrier that checks whether a stored pointer is cross-generational before recording it. It was shown that even though no single garbage collector enables the best performance for all programs, the overall performance (regarding execution time) of individual application improved by more than 15% when using the profile-directed collector selection compared to using any of several fixed collectors. The most critical choice concerning performance was whether to use a generational collector. Generational collectors did well on benchmarks that had high collection costs and did poorly for those that had low collection costs and high write barrier costs. The choice between the different write barriers did not affect the average performance across all benchmarks that much considering they all fell within 2% of each other.

3.2.2 S. Soman et al. [8]

Introduce a design, implementation, and experimental evaluation of a JikesRVM extension that facilitates dynamic switching between several very different and popular garbage collectors. A framework that can automatically switch between GC algorithms during runtime without having to restart and possibly rebuild the execution environment, as is required by existing systems, was developed. The experimental evaluation on the framework implementation using benchmarks suites, e.g., SPECjvm98 and SPECjbb, JOlden, and JavaGrande, showed performances 4% below the average compared to the best-performing collection system for a particular heap size given the range of heap sizes studied. However, there was a 24% average improvement on performance over the worst performing GC system at each heap size and 7% average improvement over always using the popular Generational/Mark-Sweep hybrid.

3.2.3 J. Singer et al. [9]

presented a new approach to determining the most suitable garbage collector algorithm, specifically in terms of minimal overall execution time, to use for a Particular application without having to profile that program with every available GC algorithm exhaustively. This exhaustive profiling has been a common approach for both static and dynamic selection of GC algorithms, which is its main drawback due to the lengthy time it takes. To avoid exhaustive profiling, machine learning is used to predict which GC algorithm performs best for a given JVM heap size, without having to execute all GC algorithms on that particular program. A model is built, that relates various applications with optimal GC algorithms based on certain features of that application. Then, to select the optimal GC for a not yet seen program, we measure its features, find the most ”similar” program in the model, and select the GC algorithm that the model predicts.

Various measurements of benchmarks on the JikesRVM are used as features that characterise an application, such as static program metrics which measure properties of the Java bytecode independent of any Virtual Machine or GC algorithm (e.g., count of the number of methods, length of the maximum path in the inheritance tree from the root java.lang.Object to each class, among others). Dynamic program metrics which measure runtime behaviour of the program running on a particular JVM with a default GC algorithm (e.g., object demographics like the number of allocated objects, and GC-dependent demographics such as the number of full GCs, among others). Virtual Machine metrics which measure heap properties of the JVM (e.g., initial heap size, maximum heap size. The benchmarks suites used are the SPECjvm98, SPECjbb2000, DaCapo and JOlden which represent a wide range of user applications from all different genres of general purpose software).

The authors claim an average speedup of 5% when compared to the default execution time, up to a maximum possible speedup of 17%, averaged over all benchmarks and heap sizes. Additionally, they estimate to have reduced the time spent profiling by 66% on a sample of six possible garbage collectors, where instead of having to run the application once for each garbage collector, this approach only requires the execution of the program two times.

There are two distinct approaches in recent literature when it comes to try and map the most suitable garbage collector to a particular application needs. The first approach revolves around performing extensive profiling of the application, before its execution, in order to select the most suitable GC. Fitzgerald et al. [7] was the first to present a profiling framework to choose a single most-suitable GC. Soman et al. [8] took another approach, selecting multiple garbage collectors instead and switching between them in runtime, based on which GC is most-suitable for a particular period. The main disadvantage of this approach is the requirement of extensive profiling, which Singer et al. [9] tries to reduce using machine learning techniques. A second approach consists of evaluating and comparing different garbage collectors w.r.t specific performance metrics, using one or multiple benchmark suites or applications. Xu et al. [2] analyses the correlation between big data applications' memory usage patterns and the collectors' memory usage to obtain findings regarding GC inefficiencies. Yu et al. [3] shows a performance analysis on the overall performance impact of Full GC in memory-hungry applications that handle large data sets, more particularly when using the Parallel Scavenge (PS) garbage collector. W. Zhao et al. [6] evaluates the impact of each of the significant elements of G1 on performance (pause time, remembered set footprint, and barrier overheads) by deconstructing the G1 algorithm and re-implement it from first principles. P. Pufek et al. [4] analyses several garbage collectors (i.e., G1, Parallel, Serial, and CMS) with the DaCapo benchmark suite, comparing the number of algorithms' iterations and the duration of the collection time.

Table 3.1 shows a summary of all the garbage collectors, benchmarks, and performance metrics presented in the previous papers. With this work, using the second approach, we hope to improve on earlier studies (see Table 3.1) through the use of a wider range of state-of-the-art garbage collectors, including fully concurrent implementations such as the ZGC and Shenandoah. Additionally, we will be evaluating a broader range of performance metrics like latency, memory usage, and throughput in contrast to a single performance metric evaluation, such as execution time [7–9], latency [3, 4], or memory usage [2]. The evaluation is performed using a widely used benchmark suite, Big Data platforms, and fine-grained benchmarks, on top of an industrial JVM (OpenJDK Hotspot), contrary to the regularly used

Table 3.1: Comparison between the presented papers

Paper	Garbage Collectors	Benchmarks	Performance Metrics
Xu et al. [2]	Parallel Scavenge Con. Mark Sweep Garbage-First	Spark Framework	Memory Usage
Yu et al. [3]	Parallel Scavenge	JOlden Dacapo SPECjvm2008 Spark Framework Giraph Framework	Latency
P. Pufek et al. [4]	Serial Parallel Scavenge Conc. Mark and Sweep Garbage-First	Dacapo	Latency
W. Zhao et al. [6]	Garbage-First	Dacapo SPECjvm98 pjbb2005	Memory Usage Latency Execution time
R. Fitzgerald et al. [7]	Null Copying Gen. Copying	SPECjvm98 MISCJAVA IMPACT DOCTOR	Execution time
S. Soman et al. [8]	Semispace Copying Mark Sweep Gen. Mark Sweep Gen. Semispace Non-gen. Semispace	SPECjvm98 SPECjbb2000 JOlden JavaGrande	Execution time
J. Singer et al. [9]	Copy Mark Sweep Gen. Mark Sweep Gen. Copying Mark Sweep Mark Compact Semispace	SPECjvm98 SPECjbb2000 Dacapo JOlden	Execution time

academic-oriented JikesRVM.

4
Proposed Solution

Contents

Workload	Description
fop	Takes an XSL-FO file, parses it and formats it, generating a PDF file
h2	Executes a JDBCbench-like in-memory benchmark
jython	Interprets a the pybench Python benchmark
luindex	Uses lucene to index a set of documents
lusearch	Uses lucene to do a text search of keywords over a corpus of data
pmd	Analyzes a set of Java classes for a range of source code problems
sunflow	Renders a set of images using ray tracing
tradebeans	Runs the daytrader benchmark via a Java Beans to a GERONIMO backend
tradesoap	Runs the daytrader benchmark via a SOAP to a GERONIMO backend
xalan	An XSLT processor for transforming XML documents into HTML

Table 4.1: DaCapo Benchmarks Summary

In this Chapter, we start by describing the benchmark suite and workloads used to evaluate the various garbage collectors (i.e., ParallelOld, CMS, G1, Shenandoah and ZGC) which were presented in Section 2.5. To approximate the results to real-world scenarios, a combination of real-world and synthetic benchmarks and workloads is used. An initial evaluation, that on one hand, allows us to visualise how newer garbage collectors, such as the ZGC and Shenandoah, behave in widely used and well-studied benchmark suites like the DaCapo (see Section 4.1). On the other hand, the benchmarking of Big Data Platforms (see Section 4.2) allows us to detect how the different collectors behave in memory-hungry environments in terms of throughput, latency and memory usage. Later in Section 4.3, a fine-grained evaluation of the garbage collectors is proposed, using a small self-made micro-benchmark that stresses particular garbage collectors components, so that we can better measure how specific concurrency techniques employed by collectors (e.g., synchronisation barriers) affect performance on memory-hungry environments. New garbage collectors (i.e., Shenandoah and ZGC) introduce read barriers in their implementation. We want to stress these mechanisms to expose the overhead associated with these barriers in our micro-benchmark. Every workload is run on top of a widely used industrial JVM, the OpenJDK 11 Hotspot, whose architecture was described previously in Section 2.2.

4.1 Dacapo Benchmark Suite

The DaCapo [17] is a well-known and widely used benchmark suite to analyse the performance of a JVM. It is composed of sub-benchmarks that simulate real-world workloads that focus on different performance features, i.e., non-trivial memory-intensive workloads, CPU intensive workloads, etc. This benchmark suite outputs the sub-benchmark' execution time over one or more iterations of each workload as its performance metric. All the sub-benchmarks come with pre-configured workloads, which we run with the various chosen garbage collectors. There are several advantages to using this suite, such as: i) allows the testing of the different garbage collectors (i.e., ParallelOld, CMS, G1, Shenandoah and ZGC) with many different workload types; ii) these workloads are well studied facilitating the task of understanding potential results; iii) due to the broad use of the benchmark suite it facilitates the comparison with other works [3, 4, 6].

A summary of all the workloads present in the DaCapo suite is presented in Table 4.1. Below we describe in more detail each workload present in DaCapo:

- fop – FOP is a formatter driven by XSL formatting objects. It is a Java application that reads a formatting object tree and then transforms it into a PDF document. The formatting object tree can be in the form of an XML document (output by an XSLT engine like Xalan) or can be passed in memory as a DOM Document.

- h2 – A workload that exercises database operations. The h2 benchmark uses a relational database management system, Apache Derby, and is based on the TPC-C workload which simulates a complete environment where a population of terminal operators executes transactions against a database. It is a composite of read-only and update intensive transactions that mimic the activities found in complex online transaction processing environments.

- jython – The jython workload interprets a python benchmark, pybench, which is a compilation of tests that provides a standardized approach to measure the performance of Python implementations.

- luindex – Performs a write-intensive workload upon Lucene, an open-source search engine software library. The workload performs full-text indexing of a set of documents.

- lusearch – The lusearch workload performs full-text searching of keywords over a pre-indexed set of documents. The workload exercises mostly read operations.

- pmd – The pmd workload is a static source code analyzer. It finds common programming flaws like empty catch blocks, unused variables, unnecessary object creation, and so forward.

- sunflow – The sunflow workload tests graphics visualization using an open source, internally multithreaded global illumination rendering system. The sunflow library is threaded internally, i.e. it's possible to run several bundles of dependent threads to render an image. It starts with half the number of threads as the number of logical CPUs, and each of these threads spawns four threads inside the program.

- tradebeans and tradesoap – The tradebeans and tradesoap workloads run the daytrader benchmark application, which emulates an online stock trading system allowing users to login, view their

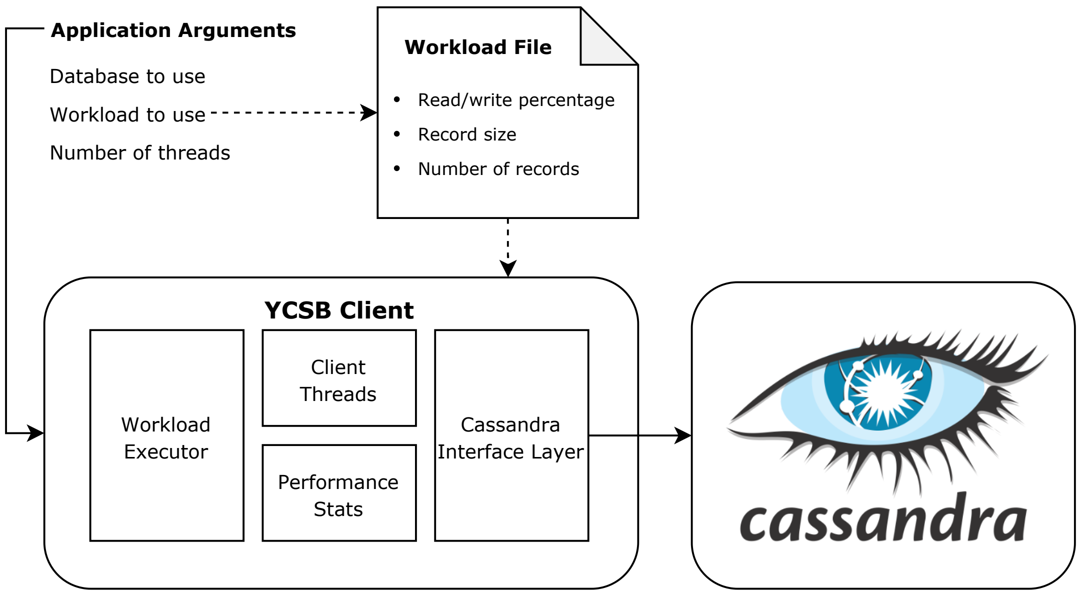


Figure 4.1: Yahoo! Cloud Serving Benchmark framework

portfolio, look up stock quotes, buy and sell stock shares, and more. The tradebeans benchmark runs the daytrader benchmark via a Java Beans to a GERONIMO backend while the tradesoap benchmark does so via SOAP.

- xalan – The xalan workload is an XSLT processor for transforming XML documents into HTML. This workload has a high allocation rate, high level of contended locks and heavily exercises string operations.

4.2 Big Data Platforms

To ensure the results obtained have a high degree of accuracy, we use a combination of different Big Data Platforms in our evaluation. We opted to use Cassandra and Lucene as distinct examples of storage platforms because they are both widely used platforms (e.g., Netflix [54], Uber [55], CERN [56]), where downtime or data loss is unacceptable (i.e., there is a strong emphasis on latency in these applications). Cassandra is a distributed key-value store; while Lucene is a in-memory text engine. In addition, to have a representation of the different types of Big Data platforms, we use GraphChi, a graph processing engine, in our evaluation as an example of a Big Data processing platform and also a throughput-oriented application.

4.2.0.A Cassandra

Apache Cassandra [18] is a wide column store and one of the most popular open-source distributed NoSQL database management systems designed to handle large amounts of data across many commodity servers. Cassandra is used in our evaluation as an example of a Big Data storage platform. Three different workloads, with varying percentages of read and write queries over 10000 queries per second are used in our evaluation with Cassandra: i) a read-intensive (RI) workload (consisting of 75% read queries, and 25% write queries); ii) a write-intensive (WI) workload (75% write queries, and 25% read queries); iii) a balanced workload (RW) (50% read queries and 50% write queries). The choice to perform 10000 queries per second was to not bottleneck the application throughput in any of the experiments in our evaluation. Each evaluation performs a fixed number of read and write operations (i.e., the cumulative number of read and write queries performed during the evaluation) over numerous records. These workloads are synthetic but mirror real-world settings of real users performing similar workloads upon database systems.

Yahoo! Cloud Serving Benchmark [57] (YCSB) is used to benchmark Cassandra utilising the various garbage collectors. YCSB is an open-source Big Data benchmarking tool [58], often used to compare the relative performance of NoSQL database management systems. An overview of YCSB framework is presented in Figure 4.1. YCSB receives as parameters: i) a workload file defined by the user, which specifies the percentage of read and write queries; ii) the working set size composed by the number of records and corresponding size per record; and iii) the database to benchmark and how many client threads to use. YCSB then populates and performs multi-threaded client requests upon the database (Cassandra in our evaluation) while concurrently measuring and storing performance metrics.

4.2.1 Lucene

Lucene [19] is a free and open-source search engine software library. While fitting for any application that requires full-text indexing and searching capability, Lucene is primarily used to implement Internet search engines, and local single-site searching. A well-known usage of Lucene in Big Data is through its sibling project Apache Solr [59]. Figure 4.2 describes the framework to independently evaluate Lucene, using a benchmark that performs text-indexation and search. The benchmark starts by building an in-memory text index using a 31 GB Wikipedia dump divided in 33M documents, which represents a real-world use-case of Lucene. The workload is comprised of 20000 writes (document updates) and 5000 reads (document searches) per second, which represents an example of a worst-case scenario for a garbage collector latency metric due to it being a write-intensive computation. For the worst case of the read operation (document searches), we loop through the 500 top words in the dump, which is a read-intensive computation for Lucene. Although Lucene was used in our evaluation with the DaCapo

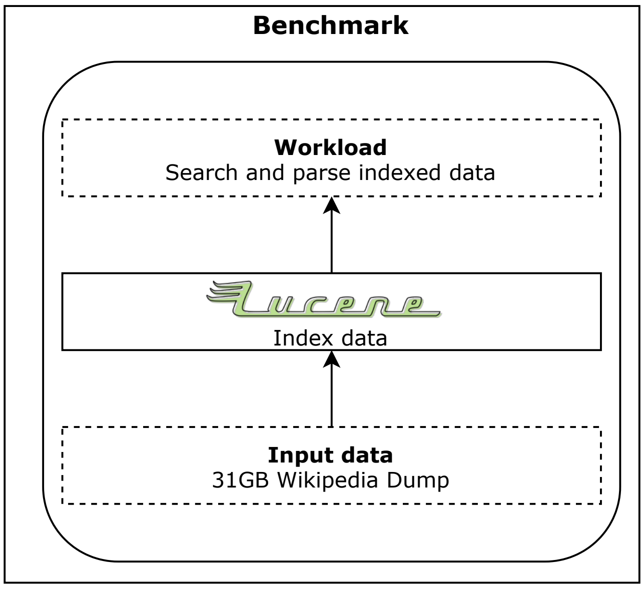


Figure 4.2: Lucene benchmark

Benchmark Suite, we decided to also independently benchmark Lucene, to observe its behaviour with a large input data and obtain other pertinent performance metrics such as latency and memory usage (DaCapo only offers execution time as its performance metric).

4.2.2 GraphChi

In contrast to the previously described platforms (Cassandra and Lucene), GraphChi [20] is used in our evaluation as an example of a processing platform. GraphChi is a disk-based system for computing efficiently on graphs with billions of edges in a single system. The main advantage of using GraphChi is that we can compute on large graphs without the hassle of having to use a large distributed system. Performance-wise GraphChi is highly competitive with large Hadoop clusters. Therefore, we use GraphChi as a throughput-oriented system used to run two well-known algorithms, PageRank, and connected components. Both algorithms are supplied with a Twitter graph [60], which is an example of a real-world workload, consisting of 42 million vertexes and 1.5 billion edges where all in-edges from the graph are stored in shards. The intent is to load into primary memory these vertexes and corresponding edges in batches, where GraphChi main task is responsible for estimating a memory budget to limit the number of edges to load from the shards into memory in each batch. The preceding task (memory budget estimation) depicts an iterative process (illustrated in Figure 4.3), wherein each iteration, a new group of vertexes is loaded and processed. Figure 4.3 shows an iteration of GraphChi, where GraphChi estimates a batch of vertexes and corresponding edges from the shards to be supplied to PageRank or the Connected Components algorithms. GraphChi continues to iterate until all vertexes and edges in

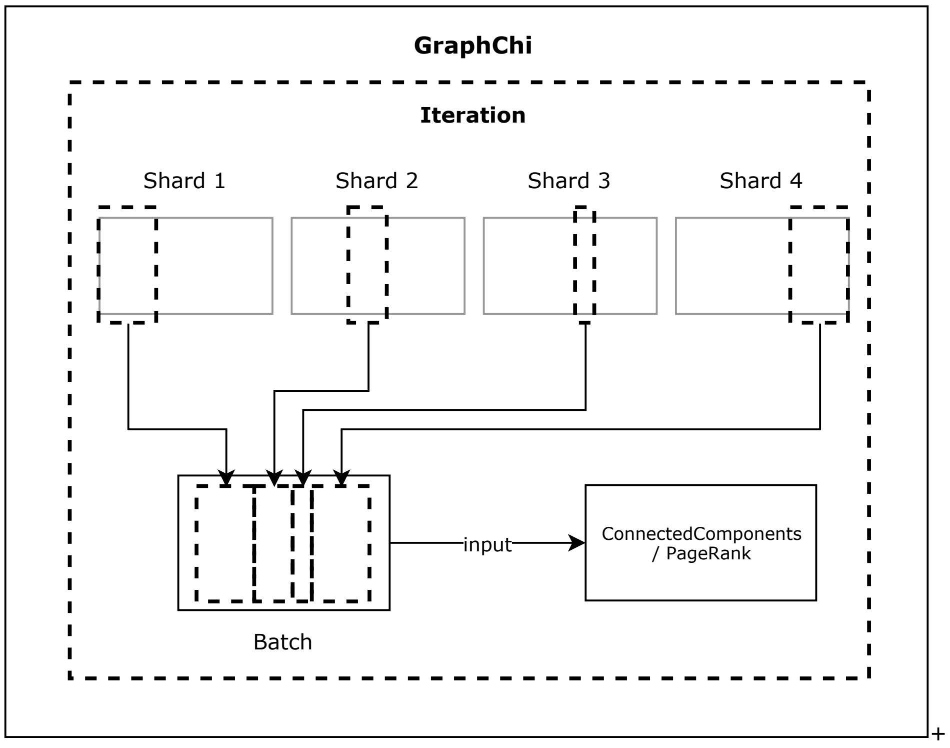


Figure 4.3: GraphChi iterative process

the shards are processed.

4.3 Fine-Grained Benchmark

Concurrent compaction in recent garbage collectors (e.g., ZGC and Shenandoah), as described before in Section 2.5.4 and 2.5.5, introduce new overheads on the application performance. Typically, this overhead presents itself as a consequence of the need for synchronisation between mutator and collector, usually affecting the application' throughput (e.g., barriers), memory usage (e.g., brooks pointers), pressure over the TLB (e.g., coloured pointers) and execution time by correlation with the former. In generational collectors, a write barrier being triggered to update the remembered set, every time a field of an object is updated or a new reference is stored, also presents a overhead on performance. To better understand precisely how and how much these techniques affect performance metrics, we developed a micro-benchmark meant to stress these particular components.

The main idea behind the fine-grained benchmark is that for some garbage collectors, certain workload operations trigger specific garbage collector components, which has a performance impact (e.g., read and write operations trigger read and write barriers, respectively). Therefore, the base design of a fine-grained benchmark would consist of populating a small data structure (e.g., HashMap in Java) in memory and running a workload on it to try and stress a particular garbage collector component. In

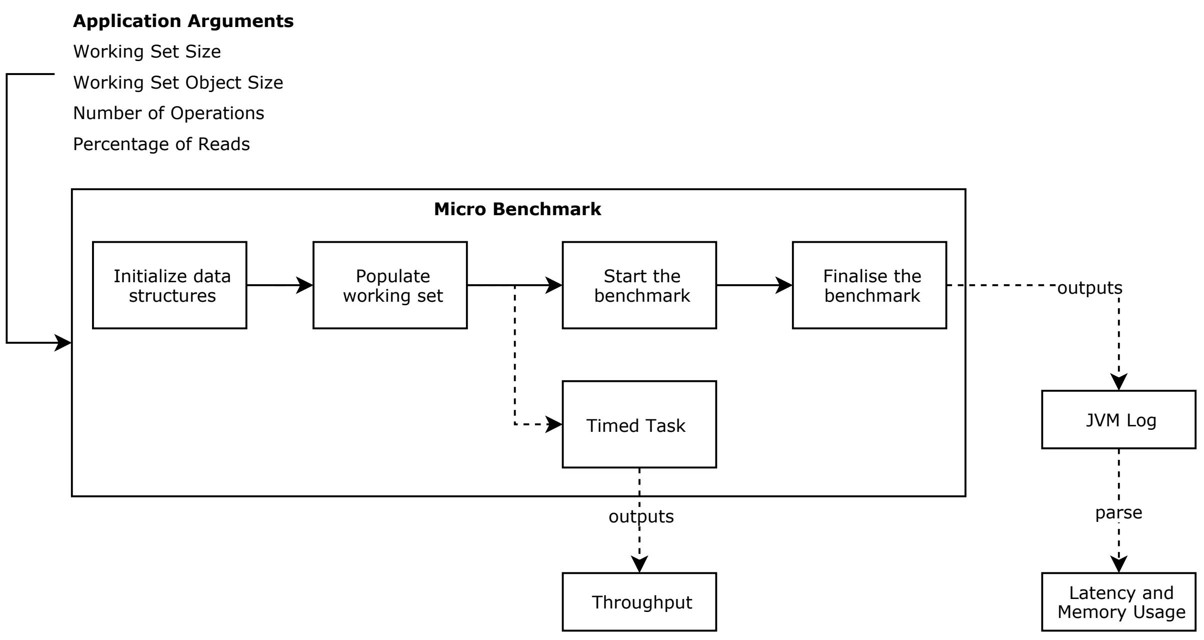


Figure 4.4: High-level view of our micro benchmark

```java
static class GenericObject {
    public long[] obj;
    public int number;

    public GenericObject(int value, int objectSize) {
        obj = new long[128 * objectSize];
        number = value;
    }
}
```

Listing 4.1: Implementation of each object in the working set

Figure 4.4, we show the life cycle of our micro-benchmark. We start by receiving from the user information such as: i) how large should the working set be; ii) the size of each object of the working set; ii) how many operations shall be performed upon the working set; and finally, iv) what percentage of those operations shall be reads and per consequence how many shall be writes. Following that, we initialise and populate the working set and set up two direct ByteBuffers that will support the benchmark execution. The choice to use direct ByteBuffers was because these are structures (buffers) that are allocated outside the garbage-collected heap (i.e., structures not managed by the garbage collecter) and therefore do not affect the performance of the garbage collector in general.

The working set is then populated with objects with two fields, a primitive type and a non-primitive type. The reason to have both fields is so that we can evaluate our benchmark with both types of objects since synchronisation barriers in some garbage collectors (e.g., Shenandoah) are affected by primitive types, and others are not (e.g., ZGC). Both the object size and size of the working set, as we mentioned before, are passed as input parameters by the user. Having the flexibility to run the benchmark with different object' sizes is important since some garbage collectors may behave differently for larger objects (e.g., G1). Listing 4.1 shows the implementation of a generic object like we described before. As we start the execution of the workload, we start a timed task to output the application' throughput periodically (i.e., a method that is automatically invoked periodically to store the number of operations that were successfully performed in that period). Then, the workload is initialised performing a number of read and write operations (implementation show in Listing 4.2), passed by the user at the beginning of the benchmark execution, over random objects in the working set. As shown in Listing 4.2, the main loop of the benchmark consists in performing (until a total number of operations desired by the user is performed) the following steps: 1) pick a random object from the working set; 2) based on the user's desired percentage of read and writes, decide if a read or write operation should be performed (i.e., pops a random Integer from one of the ByteBuffers, which stores random values between 0 and 100; if the popped value is less than the read percentage desired it performs a read operations, otherwise it performs a write operation); 3) executes the chosen operation upon the chosen random object.

Consistency between different benchmarks executions is ensured by having a static seed generating the arbitrary values of the ByteBuffers, who store the information of which object should be accessed and what type of operation should be performed upon the object in each iteration. Each read and write operation, depending on the underlying garbage collector and which barriers it implements (see Figure 2.5), causes a read and write barrier to be invoked respectively. The goal is that by varying the percentages of read and write operations over different micro-benchmark executions, we can determine the cost in terms of throughput, latency and memory usage of the read and write barriers associated with those operations.

Additionally, in our evaluation, we also use the micro-benchmark to assert the existence of an application throughput overhead caused by the necessity of a memory barrier (i.e., a read barrier as described in Section 2.4.2) to perform concurrent compaction. We do so by running the benchmark in two configurations over ZGC. The first configuration where the benchmark performs only read operations over primitive types, which do not trigger read barriers for ZGC. In the second configuration, we perform read operations solely over non-primitive types, which triggers a read barrier for every read operation. Performing multiple experiments using these two configurations while increasing the percentage of read operations, allows us to assert if there is an increasing application throughput overhead (on the non-primitive configuration) as the percentage of reads increases compared to the primitive configuration.

```java
// Benchmark main loop for non-primitive types
private void benchmark(int readPercentage) {
    for (int i = 0; i < numOperations; i++) {
        GenericObject object = population.get(String.valueOf(bufferAccess.getInt()));
        if (bufferPercentage.getInt() < readPercentage){
            readObject(object);
        }
        else {
            writeObject(new long[128 * objectSize], object);
        }
    }
}
// Write operation of a non-primitive type
private void writeObject(long[] newObject, GenericObject object){
    object.obj = newObject;
}
// Read operation of a non-primitive type
private void readObject(GenericObject object){
    discard_obj = object.obj;
}
```

Listing 4.2: Implementation of a generic benchmark for non-primitive types

5 Evaluation Results

Contents

Depending on an application's needs, specific performance metrics are more desired than others for that particular application. An example is one of the applications developed by Feedzai:[1] it validates credit card transactions using near real-time machine learning to analyse Big Data stored in JVM-powered databases. Failing a Service Level Agreement (SLA) for a particular transaction due to a long garbage collection pause while performing a database read or write operation would significantly impact the company negatively.

Usually, garbage collectors can be divided into three groups regarding performance: i) those that offer a guarantee of low pause times, typically under ten milliseconds, such as the ZGC and Shenandoah; ii) those that seek to achieve the best throughput possible, such as the ParallelOld; iii) and those that attempt to strike a compromise between low pause times without sacrificing the application's throughput too much, such as the CMS and G1 collectors. To better understand the trade-offs between each group, we study specific performance metrics using the benchmarks described in Chapter 4. In particular, we focus our analysis on the following performance metrics: application throughput, memory utilisation, and latency.

The experiments consist of all possible combinations of the following:

1. We use several Garbage Collectors, more specifically the ParallelOld (see Section 2.5.1), CMS (see Section 2.5.2), G1 (see Section 2.5.3), ZGC (see Section 2.5.4), and Shenandoah (see Section 2.5.5).

2. We increase the size of the Java Virtual Machine (OpenJDK 11 Hotspot) heap, so that we may observe performance wise how each garbage collector behaves with different sized heaps and interpret why it performs that way.

3. We use the benchmarks described in Chapter 4 (i.e., DaCapo, Cassandra, Lucene, GraphChi, and the micro-benchmark), which determine the type and numbers of accesses used in each experiment.

The results were extracted either directly from the log file produced by the JVM (we did not change the logging infrastructure for the JVM) or from the benchmark log file. From the JVM log, we extracted the memory available before and after each garbage collection cycle for memory usage, and the time an application threads had to halt execution for a garbage collection cycle to be performed. The throughput metric was extracted from the benchmark log file. The experiment results were analysed in different ways according to the particular performance metric in question. We decided to extract the number of operations per second performed by an application and use a 95% confidence interval as the application throughput metric. Latency was measured across multiple percentiles (i.e., 99th, 99.9th, worst) of

[1] Feedzai (www.feedzai.com) is a world leader data science company that detects fraud in omnichannel commerce. The company uses near real-time machine learning to analyse Big Data to identify fraudulent payment transactions and minimise risk in the financial industry.

all pauses. Lastly, memory utilization was determined as the percentage of heap space used by an application over the defined total heap space with a 95% confidence interval, extracting as well the max memory usage the application reached during the benchmark execution. The reason we use 95% confidence intervals instead of higher confidence intervals is so that the result intervals are tight enough that there is sufficient differentiation between the garbage collectors without losing much confidence. With higher confidence intervals, the intervals become significantly wider, which causes considerable overlapping when comparing results between different garbage collectors.

5.1 Evaluation Setup

The evaluation was performed on a server equipped with an Intel Xeon E5505, with 16 GB of RAM, with a Linux version 3.13. Each experiment runs five times in complete isolation, enough to detect and discard outliers. To ensure minimal overhead caused by the Java Virtual Machine (e.g., JVM loading, JIT compilation), the first two minutes of execution in each experiment were discarded. Heap sizes vary between the values of 2GB up to a maximum of 12GB for each of the used garbage collectors, i.e., ParallelOld, CMS, G1, ZGC, Shenandoah. For all benchmarks, we set the initial heap size at the same value as the maximum heap size and pre-touched the pages to avoid runtime resizing, and memory commit hiccups. The number of concurrent threads with respect to the application threads and stop-the-world worker threads is left to the default value of the JVM.

5.2 Micro Benchmark

In this section, we start by analysing fine-grained benchmarks that stress-tests the component that we suspect to be the playing a major factor for the trade-off between performance metrics, as described in Section 4.3.

The workload stresses the memory barriers present in the different collectors in a controlled environment, more accurately the remembered-set barriers in the generational collectors (e.g., ParallelOld, CMS and G1) and the memory barriers that allow concurrent compaction in the concurrent garbage collectors (e.g., Shenandoah and ZGC). We benchmark each garbage collector with variable percentages of read and write operations.

5.2.1 Application Throughput and Latency

The workload in our micro-benchmark mainly allocates small objects that are very likely to be promoted to the old generation (i.e., it does not abide by generational hypothesis that most objects die young) while also being referenced by old generation objects, which exercises the remembered set barriers.

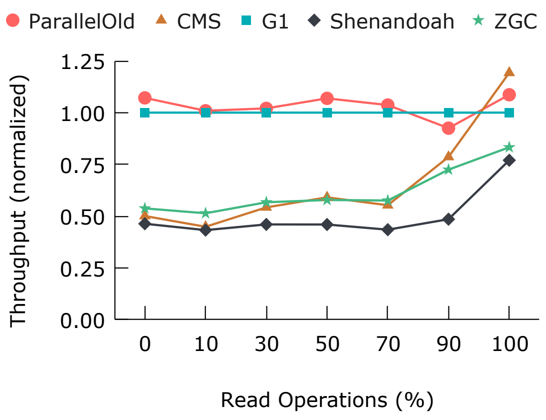


Figure 5.1: Throughput Results for the Micro Benchmark

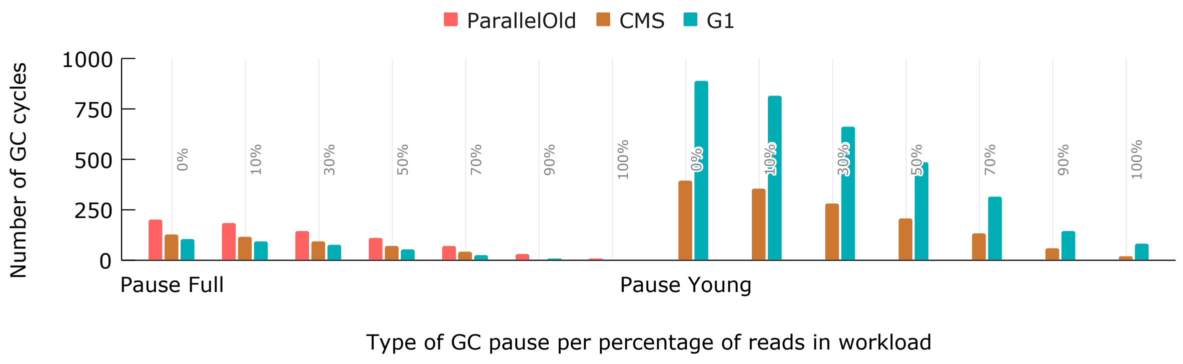


Figure 5.2: Volume of GC Pauses per Percentage of Reads in Workload

Combined with a large allocation rate and garbage being created in the old generation, this workload presents a highly stressful memory management for the generational collectors. To exercise the memory barriers in the concurrent collectors, all operations (i.e., read and writes) are done over non-primitive types, which triggers a read and write barrier, respectively.

Figure 5.1, shows for each garbage collector the application throughput for our workload as we increase the percentage of read operations. When comparing the generational collectors, CMS is the most affected by the workload characteristics as shown by the up to 50% decrease in throughput when compared to G1 and ParallelOld. This decrease is caused by CMS trying to promote objects from the young generation while concurrently performing mark-and-sweep of the old generation. However, CMS ultimately falls back to a full GC as it fails to concurrently collect the old generation fast enough, as shown in Figure 5.2. Since the ParallelOld only collects the old generation through full collections, the frequency of a full GC is higher, but their duration is lower than CMS, as shown in Figures 5.2 and 5.3, respectively. The region-based memory management of G1 allows it to reduce the frequency of a full GC significantly, to collect and compact the old generation, resulting in comparable throughput results

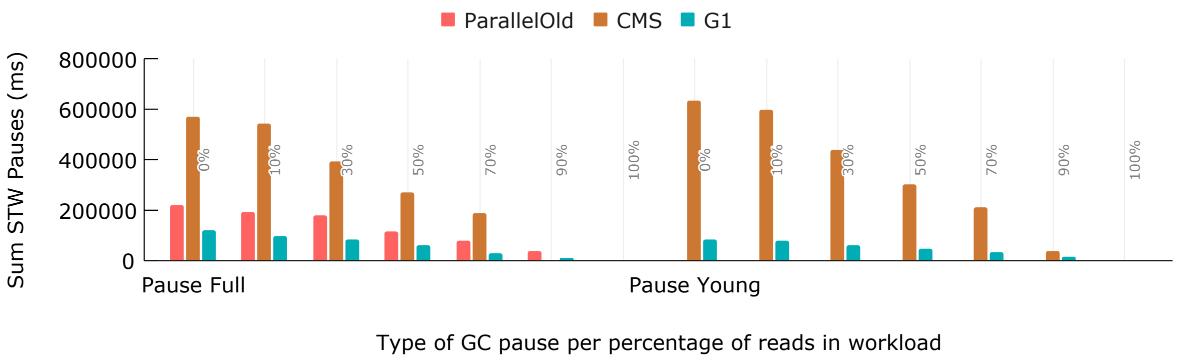


Figure 5.3: Sum of GC STW Pauses per Percentage of Reads in Workload

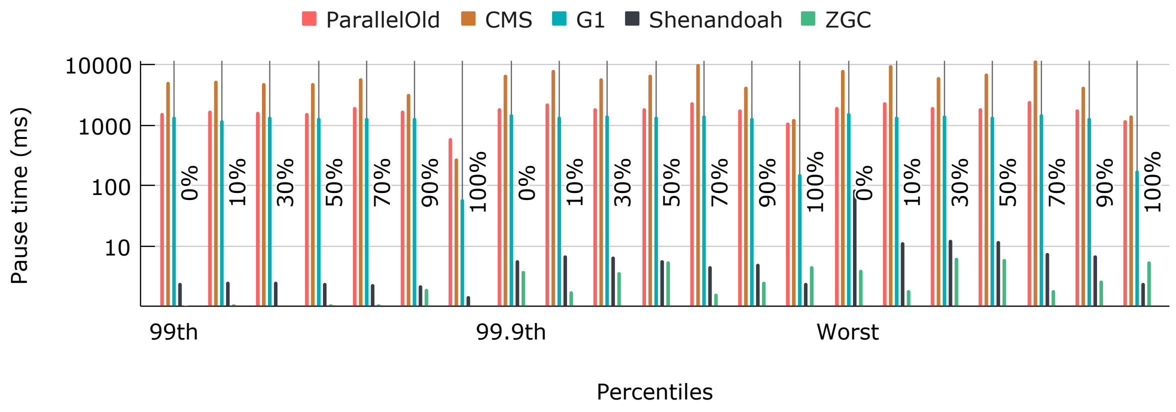


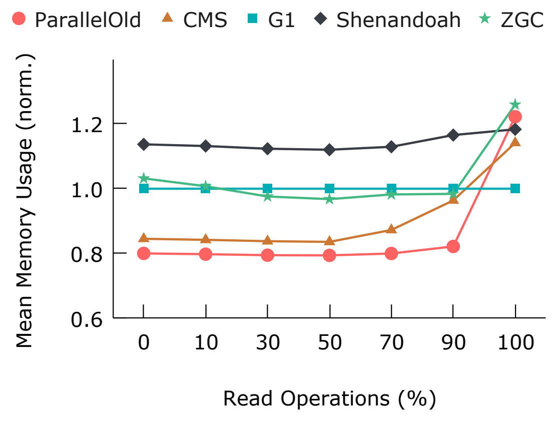
Figure 5.4: Pause Time Percentiles (ms) the Micro Benchmark

to the ParallelOld.

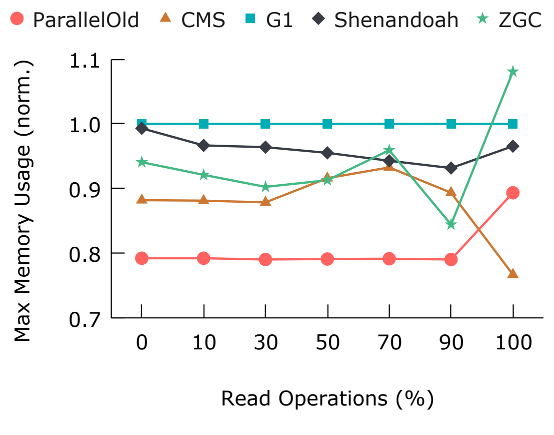
As for the concurrent collectors (i.e., Shenandoah and ZGC), these present significantly lower throughput than the generational collectors due to synchronization barriers that allow concurrent compaction. When comparing ZGC and Shenandoah, the former shows slightly higher throughput as it does not require the use of write barriers in its implementation. The existence of a throughput trade off introduced by synchronization barriers is validated in Section 5.2.3. As for the latency metric, both concurrent collectors manage to keep STW GC pauses below 10 ms with ZGC showing a slightly higher sensitivity to the percentage of write operations, as shown in Figure 5.4.

5.2.2 Memory Usage

Figures 5.5(a) and 5.5(b), show for each garbage collector evaluated the average and max memory usage, respectively, as we increase the percentage of read operations in the workload. When comparing the concurrent (i.e., Shenandoah and ZGC) and generational collectors (except G1), the first one shows



(a) Average Memory Usage Results for the Micro Benchmark



(b) Max Memory Usage Results for the Micro Benchmark

Figure 5.5: Average and Max Memory Usage (normalized to G1) for the Micro Benchmark

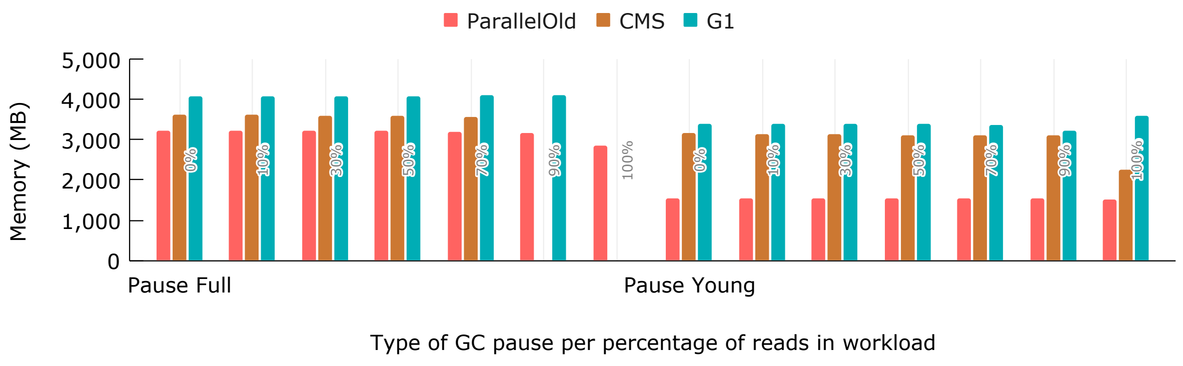


Figure 5.6: Average Threshold of Memory before GC Pauses per Percentage of Reads in Workload

significantly higher memory footprints than the second. Non-generational collectors take considerable longer to collect garbage as they require the whole heap to be traced in each collection, which results in a larger amount of accumulated garbage in each collection. Shenandoah particularly shows a higher memory footprint than all other collectors, which is mainly due to a design choice (i.e., the usage of Brooks pointers as described in Section 2.5.5 and the default adaptive heuristics).

When comparing the memory footprint between generational collectors, the ParallelOld has a well-defined constant memory ceiling, which is mainly dependent on the available memory space as full garbage collections of the heap are triggered when there is a failure in promoting a young generation object. Since most garbage is being created in the old generation, this requires a full garbage collection so that memory may be made available for objects to be promoted (as shown by the higher count of a full GC when compared to other collectors in Figure 5.2). The mostly-concurrent mark-and-sweep

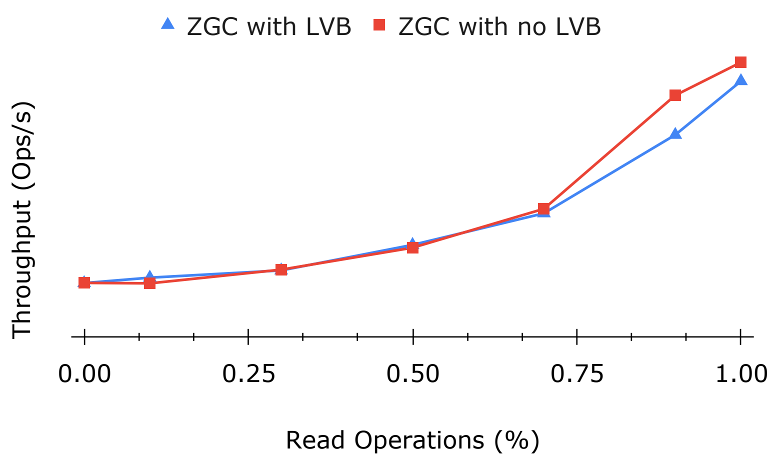


Figure 5.7: Throughput Results comparing ZGC with LVB and ZGC without LVB

collector allows CMS to delay or even remove the need for a full garbage collection to collect the old generation. However, as these mostly-concurrent collections are either triggered by default at 90% of the heap capacity or started with the aim of completing the collection cycle before the old generation is exhausted, this may result in slightly higher memory footprints for specific workloads (e.g., write-intensive) when compared to ParallelOld mainly due to garbage collection promptness. As shown in Figure 5.6, due to the high allocation rate, CMS tradeoff does not pay off as it still falls back to full garbage collections to reclaim and compact the old generation at a higher memory threshold. For G1, as a region-based collector, the trade-off for significantly less full garbage collections of the heap presents itself in part in higher memory usages as a remembered set must be maintained for each region and the collection of highly used regions is delayed.

The results of this experiment with low percentages of read operations are highly similar to the results with the processing platform (i.e., GraphChi), which shows that the CMS collector is the one that suffers the most in performance as the old generation will eventually fall back to a lengthy Full GC for Big Data applications whose objects do not behave according to the generational hypothesis.

5.2.3 Concurrency Barriers in Concurrent Garbage Collectors

As described in Section 2.5.4, ZGC requires read barriers to ensure a successful implementation of concurrent compaction while assuring the integrity of memory management. With that in mind, the primary purpose of this evaluation is to assert the existence of a throughput overhead in concurrent collectors (e.g., ZGC and Shenandoah) prompted by the necessity of memory barriers to performing concurrent compaction. For this purpose, we built a micro benchmark to measure a generic application, as described in Section 4.3. To measure ZGC read barriers, named load-value barriers (LVB), we

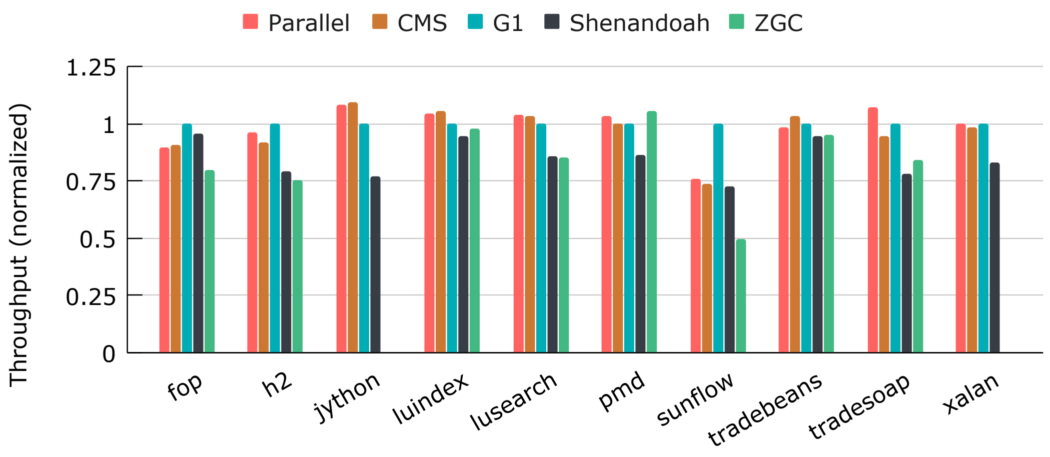


Figure 5.8: Throughput for the DaCapo Benchmark

analyse the throughput difference between the application performing read operations over primitive types (which do not trigger LVB) and read operations over non-primitive types (which triggers an LVB on every read operation). Figure 5.7 shows the throughput result for the experiment, which confirms a slight throughput overhead for the non-primitive execution of the generic application. As shown in the plot, the overhead is only noticeable for workloads with a high percentage of read operations (above 60%) where it shows up to 19% overhead on throughput. Furthermore, it shows that as the percentage of read operations increases, so does the overhead on throughput.

5.3 Benchmark Suite

The main reason we use the DaCapo Benchmark suite in our evaluation is to understand how the concurrent collectors (i.e., Shenandoah and ZGC) behave with different workloads in smaller heap sizes and how they compare to the generational collectors. Some benchmarks are omitted from the results (e.g., eclipse and tomcat) because we were unable to execute these benchmarks for some garbage collectors. Even though we were unable to benchmark ZGC with xalan and jython, since we were successful in running these benchmarks with Shenandoah, we still presented the results for these benchmarks.

5.3.1 Application Throughput

Figure 5.8 shows the results for the throughput of the DaCapo Benchmark Suite, normalized to the current default JDK garbage collector G1.

For most sub-benchmarks, both concurrent collectors behaved as was expected and showed lower throughput than the generational collectors due to a clear tradeoff to maintain ultra-low (under 10ms) GC

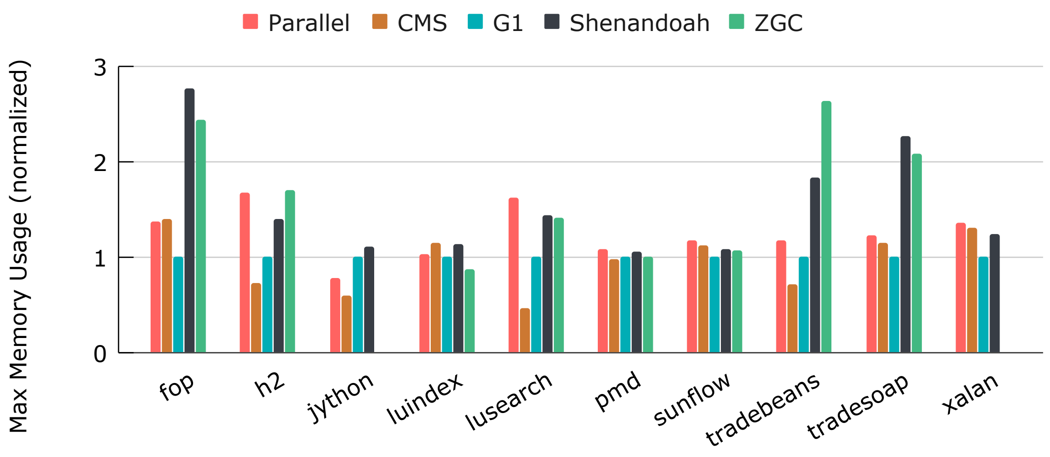


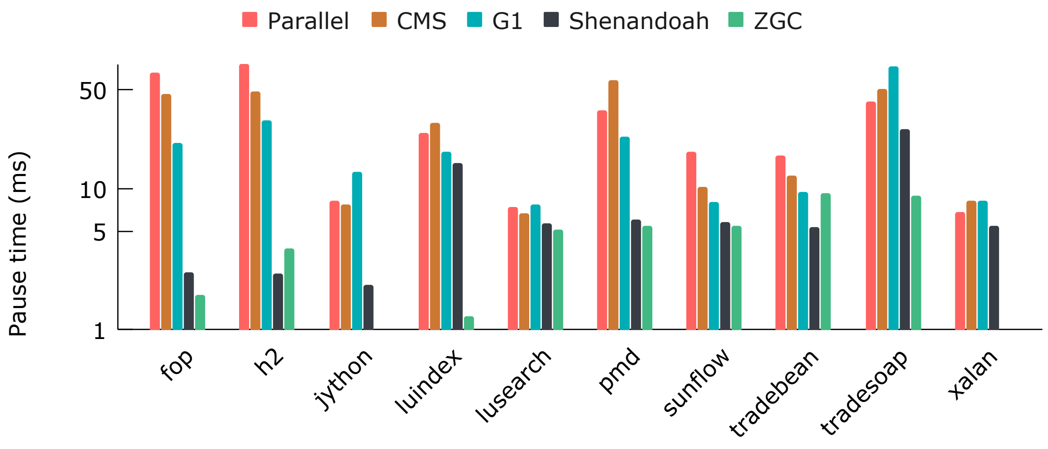
Figure 5.9: Max Memory Usage for the DaCapo Benchmark

pause times. The difference in throughput between generational collectors (which do not employ read barriers) and concurrent collectors was shown in Section 5.2. However, this behaviour did not stand true for all sub-benchmarks, as ZGC showed slightly higher throughput than all other collectors for the pmd sub-benchmark and Shenandoah showed higher throughput than ParallelOld and CMS for the fop sub-benchmark.

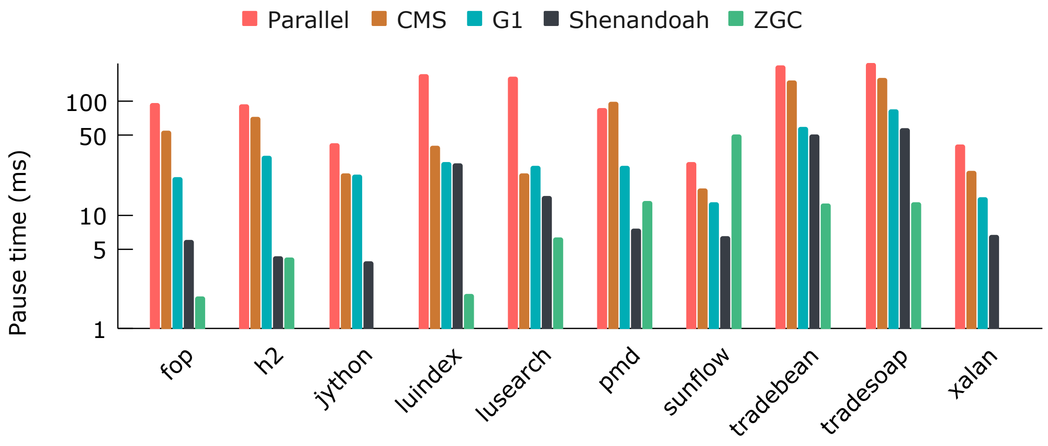
Regarding the sub-benchmarks where the generational collectors outperformed both concurrent collectors, ZGC and Shenandoah showed on average a smaller throughput of between 6% and 11% for the luindex and tradebeans sub-benchmarks, respectively. For the remaining sub-benchmarks (i.e., h2, jython, lusearch, sunflow, tradesoap and xalan), the concurrent collectors showed on average between 21% and 26% lower throughput results compared to the generational collectors, respectively.

5.3.2 Memory Usage and Latency

Figure 5.9 shows the results for max memory usage of the DaCapo Benchmark Suite, normalized to the current JDK garbage collector G1. The average memory usage is omitted as these experiments are performed over small heap sizes over a short period of time, producing similar results to the max memory usage due to the low frequency of garbage collections. Shenandoah and ZGC showed for most sub-benchmarks, memory footprints superior to G1 except for the luindex benchmark. The luindex sub-benchmark is a write-intensive benchmark that does not create large quantities of garbage, favouring ZGC as it does not require write barriers to perform concurrent compaction, which results in a lower memory footprint and, as shown in Figure 5.10, significantly lower GC pause times. However, for the Shenandoah collector, the cost of performing write barriers is presented as lengthier GC pause times (up to 30ms for the luindex benchmark) which consequently explains the overhead in throughput compared to ZGC for that particular sub-benchmark.



(a) 99th Percentile Pause Times for the DaCapo Benchmark



(b) Worst Pause Times for the DaCapo Benchmark

Figure 5.10: Pause Times for the DaCapo Benchmark

Regarding the sub-benchmarks where Shenandoah and ZGC showed significantly max memory usages compared to the generational collectors (i.e., tradebeans and tradesoap), they presented a challenge for Shenandoah as shown in Figures 5.10(a) and 5.10(b). Shenandoah shows abnormally high GC pause times in both percentiles (up to 59ms in the worst-case). ZGC, on the other hand, managed to keep GC pause times below 10ms for all sub-benchmarks except the sunflow benchmark, where it showed up to 51ms pause times in the worst-case.

5.4 Big Data Platforms

Considering the evaluation of the garbage collectors with the Big Data platforms, our primary purpose was to determine the different garbage collectors' scalability regarding application throughput, latency and memory usage. To that purpose, in order to support our analysis, we calculated the Pearson correlation[2] between the evaluated performance metrics (throughput, memory usage, and latency) and the heap size, for each garbage collector. The P-value which tells if a corresponding correlation was significant at a chosen significance level (usually denoted as α or alpha) is also presented in the results. The significance level indicates the probability (risk) of concluding that a correlation exists when no correlation actually exists. If the P-value is less or equal to the alpha level, then the correlation is significant. If the P-value is greater than the alpha level, then there is inconclusive evidence about the association's significance between the variables. Our results use a fixed alpha value of 0.05, presenting in some cases, the alpha value 0.01 as well if it applies to any result. Significant correlation allows us to more confidently predict the various garbage collectors' behavior for heap sizes more extensive than those we could test with our environment.

5.4.1 Cassandra

In this section, we look into how the different garbage collectors relate to one another regarding an increase in heap size and workload type for the Cassandra application. The workloads used to benchmark Cassandra were presented in Section 4.2.0.A, and operates upon a 2GB working set. Regarding the heap size available to all workloads, we benchmark Cassandra with 2, 4, 6, and 8GB heap sizes for all garbage collectors. The choice of heap sizes was limited by the evaluation setup (see Section 5.1), where evaluations with Cassandra on heap sizes above 8GB showed results highly affected by the over-committing of memory by the system.

[2]A Pearson Correlation [61] is a number between -1 and +1 that indicates to which extent two variables (X and Y) are linearly related. A value of 1 implies that a linear equation describes the relationship between X and Y perfectly, with all data points lying on a line for which Y increases as X increases. A value of -1 implies that all data points lie on a line for which Y decreases as X increases. A value of 0 implies that there is no linear correlation between the variables.

Table 5.1: Application Throughput (norm. to G1 at the smallest heap size) for Cassandra Workloads

WL	GC	$Throughput$				PC	PV
		$2GB$ $Heap$	$4GB$ $Heap$	$6GB$ $Heap$	$8GB$ $Heap$		
RI	$ParallelOld$	1.07 ± 0.004	1.08 ± 0.004	1.09 ± 0.003	1.09 ± 0.003	0.967	0.033^1
	CMS	1.03 ± 0.003	1.03 ± 0.003	1.03 ± 0.002	1.03 ± 0.003	-0.169	0.831
	$G1$	1.00 ± 0.005	1.00 ± 0.004	1.01 ± 0.004	1.02 ± 0.004	0.912	0.088
	$Shenandoah$	0.98 ± 0.004	1.00 ± 0.004	1.03 ± 0.004	1.03 ± 0.004	0.963	0.037^1
	ZGC	1.00 ± 0.008	1.04 ± 0.004	1.13 ± 0.007	1.13 ± 0.004	0.947	0.053
RW	$ParallelOld$	1.16 ± 0.005	1.15 ± 0.005	1.17 ± 0.005	1.17 ± 0.005	0.806	0.194
	CMS	1.10 ± 0.004	1.10 ± 0.004	1.10 ± 0.004	1.10 ± 0.004	0.123	0.877
	$G1$	1.05 ± 0.006	1.06 ± 0.005	1.08 ± 0.005	1.07 ± 0.005	0.877	0.123
	$Shenandoah$	1.05 ± 0.006	1.08 ± 0.005	1.09 ± 0.005	1.11 ± 0.006	0.963	0.037^1
	ZGC	1.06 ± 0.016	1.13 ± 0.025	1.14 ± 0.015	1.16 ± 0.011	0.905	0.095
WI	$ParallelOld$	1.29 ± 0.006	1.28 ± 0.007	1.27 ± 0.006	1.28 ± 0.007	-0.426	0.574
	CMS	1.22 ± 0.004	1.20 ± 0.005	1.20 ± 0.004	1.21 ± 0.004	-0.422	0.578
	$G1$	1.18 ± 0.006	1.16 ± 0.006	1.18 ± 0.006	1.18 ± 0.005	0.472	0.528
	$Shenandoah$	1.18 ± 0.006	1.19 ± 0.006	1.21 ± 0.005	1.22 ± 0.007	0.992	0.008^2
	ZGC	1.20 ± 0.018	1.28 ± 0.007	1.28 ± 0.008	1.29 ± 0.008	0.855	0.145

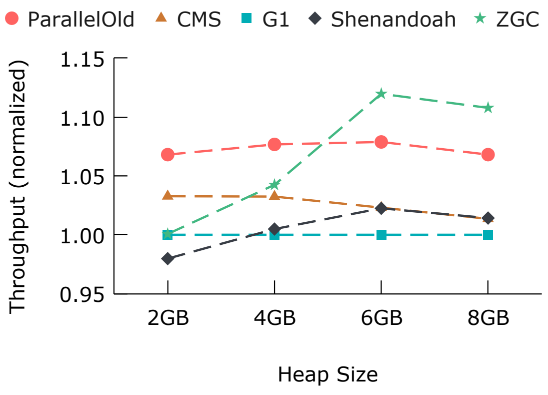
1 Correlation is significant at the 0.05 alpha level (2-tailed).
2 Correlation is significant at the 0.01 alpha level (2-tailed).

5.4.1.A Application Throughput

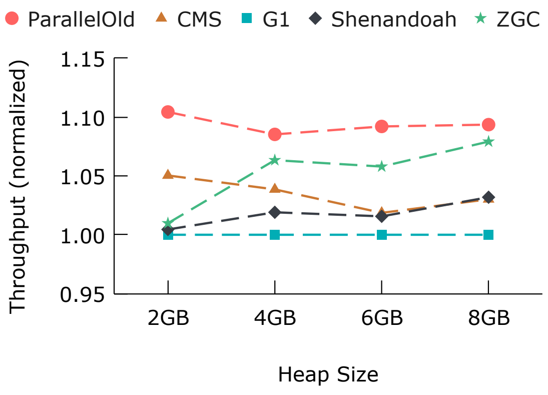
Table 5.1 presents for each workload and garbage collector with which we benchmark Cassandra (first and the second column respectively): i) column throughput – where each of its sub-columns shows the application throughput with a 95% confidence interval (normalized to the current JDK garbage collector G1, running the RI workload with a 2GB heap size) for a particular heap size; ii) columns PC and PV – represent the Pearson Correlation and corresponding P-value, respectively, which determines the relationship between the variable heap size and the application's throughput as described in Section 5.4. The top rows refer to the read-intensive (RI) workload, which performs 75% read queries and 25% write queries; the middle rows refers to the balanced (RW) workload, performing 50% read queries and 50% write queries, while the bottom rows refer to the write-intensive (WI) workload, which performs 25% read queries and 75% write queries. The decision to normalize all throughput results to this value was so that we could not only compare throughput variances between the collectors for a particular workload and heap size but also across different heap sizes and workloads.

Figure 5.11 shows the throughput growth of all collectors for the different evaluated heap sizes across all workloads with which we evaluated Cassandra. All throughput results in Figure 5.11 are normalized to G1, meaning that G1 throughput has the value one for all heap sizes.

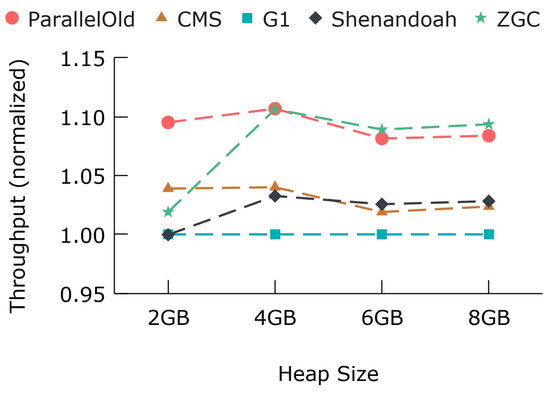
Starting with an analysis of the throughput results for the Cassandra workloads, we noticed that, as



(a) Application Throughput for RI Workload



(b) Application Throughput for RW Workload



(c) Application Throughput for WI Workload

Figure 5.11: Application Throughput (normalized to G1) for Cassandra Workloads

expected, ParallelOld was the collector that produced the best results overall (as shown in Table 5.1). It was primarily noticeable for smaller heap sizes (i.e., below 4GB). This remained true for all workloads (i.e., RI, RW and WI) with which we measured Cassandra. For heap sizes greater than 4GB, we noticed that ZGC for all but the balanced workload managed to surpass ParallelOld in terms of throughput.

Our understanding for this to happen was due to ParallelOld failing to scale significantly with an increase in heap size, having an average increase on the throughput of 0.62%, 0.44%, and -0.12% (on average) for the read-intensive, balanced, and write-intensive workloads, respectively. In turn, the ZGC presented significantly higher increases of 4.15%, 3.06%, and 2.65% (on average) for the same workloads, respectively, which allowed ZGC to stay highly competitive with the ParallelOld collector. Throughput wise, for the RI workload, ZGC showed to be able to surpass the ParallelOld and CMS collectors at a heap size three times and two times the size of the working set (2GB for this particular application), respectively. As the percentage of write queries increased (transitioning to a write-intensive workload), ZGC surpassed ParallelOld and CMS at heap sizes 2x and 1.5x times the size of the working set, respectively.

As for CMS and Shenandoah, both stayed in the middle of the pack showing similar results throughput wise. However, similarly to the ParallelOld, CMS also showed a non-substantial increase in throughput as the heap size increased with the Cassandra benchmarking. On the other hand, Shenandoah, showed a notable increase in throughput as the heap size increased and also displayed a significant positive correlation (shown in Table 5.1) between heap size and application throughput for all workloads, therefore showing a better promise for scalability than CMS. Shenandoah showed to be able to surpass CMS with a heap size 3x the size of the working set. Finally, the G1 collector was the worst-performing collector for all workloads in the Cassandra benchmarking.

For the throughput variance between workloads, the results obtained were similar for all garbage collectors. Between the RI workload and the RW workload (i.e., transitioning from 25% write queries and 75% read queries to 50% write queries and 50% read queries) showed on average an improvement on the application throughput of 6%. From the RW workload to the WI workload (i.e., transitioning from 50% write queries and 50% read queries to 75% write queries and 25% read queries) showed an improvement on the application throughput of 11% (on average). ZGC showed the most notable increase in throughput when transitioning from the RW to the WI workload, due the fact that it does not require write barriers to perform concurrent compaction, which present an overhead on throughput as shown in Section 5.2.

Figure 5.11 shows the throughput growth of all collectors for the different evaluated heap sizes across all workloads with which we evaluated Cassandra. All throughput results in Figure 5.11 are normalized to G1, meaning that G1 throughput has the value one for all heap sizes. In general, as shown in Figure 5.11, throughput wise ParallelOld and ZGC managed to perform on average between 7-9% better than G1,

Table 5.2: Max Memory Usage (% of the heap size) for Cassandra Workloads

WL	GC	$Max\ Memory\ Usage\ (\%)$						
		$2GB$ $Heap$	$4GB$ $Heap$	$6GB$ $Heap$	$8GB$ $Heap$	$Mean$	PC	PV
RI	$ParallelOld$	0.97	0.99	0.84	0.69	0.87 ± 0.14	-0.924	0.076
	CMS	0.90	0.73	0.47	0.36	0.61 ± 0.24	-0.991	0.009^2
	$G1$	0.89	0.88	0.89	0.84	0.88 ± 0.02	-0.798	0.202
	$Shenandoah$	0.84	0.85	0.85	0.85	0.85 ± 0.01	0.735	0.265
	ZGC	0.75	0.82	0.85	0.87	0.82 ± 0.05	0.965	0.035^1
RW	$ParallelOld$	0.97	0.97	0.94	0.94	0.96 ± 0.02	-0.928	0.072
	CMS	0.91	0.91	0.78	0.59	0.80 ± 0.15	-0.928	0.072
	$G1$	0.88	0.89	0.88	0.86	0.88 ± 0.01	-0.741	0.259
	$Shenandoah$	0.84	0.85	0.85	0.85	0.85 ± 0.01	0.762	0.238
	ZGC	0.74	0.87	0.80	0.82	0.81 ± 0.05	0.396	0.604
WI	$ParallelOld$	0.96	0.94	0.96	0.95	0.95 ± 0.01	-0.272	0.728
	CMS	0.91	0.91	0.91	0.82	0.89 ± 0.05	-0.764	0.236
	$G1$	0.89	0.88	0.88	0.86	0.88 ± 0.01	-0.758	0.242
	$Shenandoah$	0.84	0.85	0.85	0.85	0.85 ± 0.01	0.752	0.248
	ZGC	0.77	0.79	0.79	0.80	0.79 ± 0.01	0.944	0.056

[1] Correlation is significant at the 0.05 alpha level (2-tailed).
[2] Correlation is significant at the 0.01 alpha level (2-tailed).

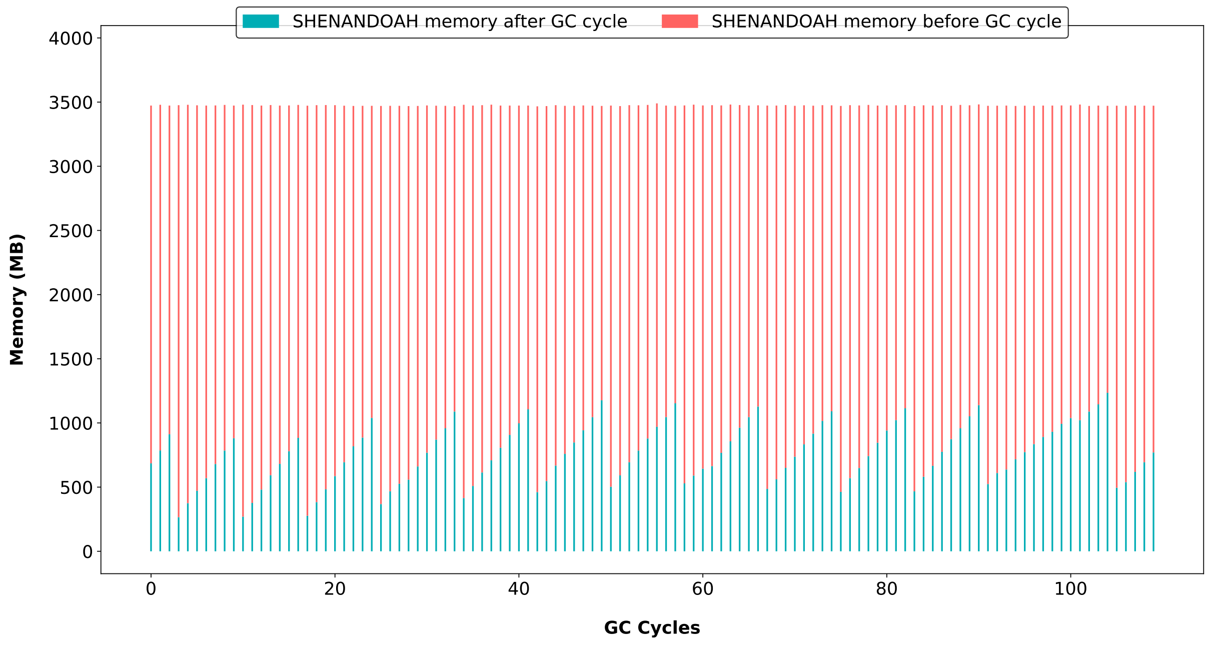
and CMS and Shenandoah performed on average between 2-3% better than G1 for all workloads.

5.4.1.B Memory Usage

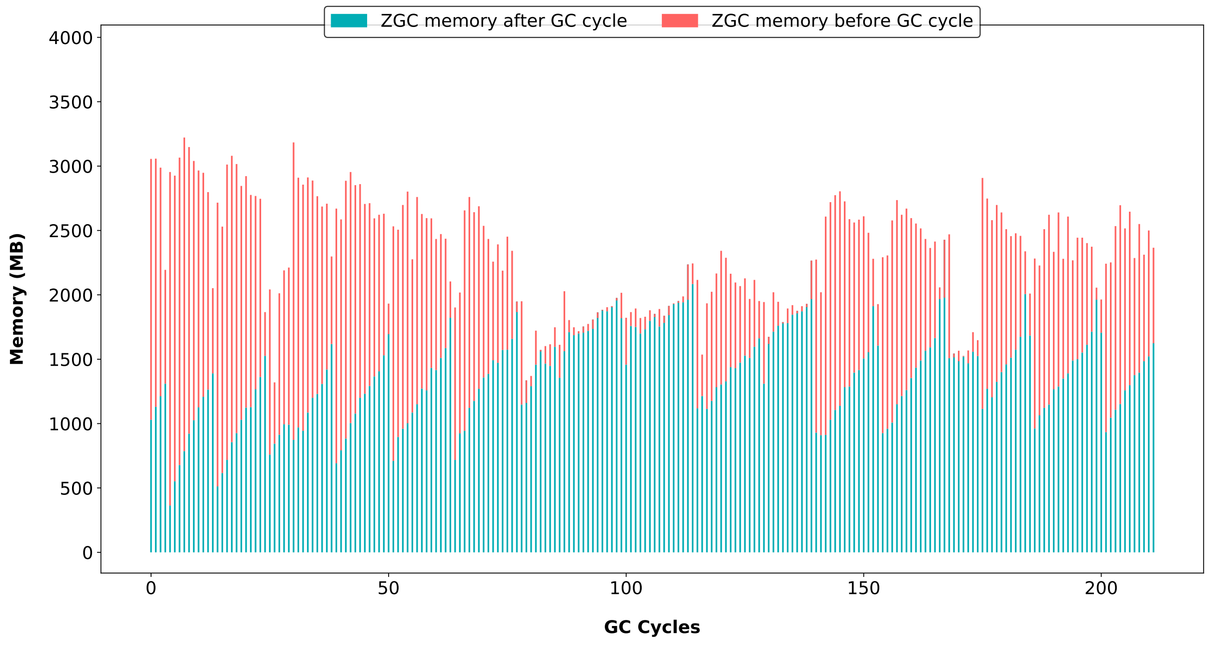
Table 5.2 presents for each workload and garbage collector with which we benchmark Cassandra (first and the second column respectively): i) column Max Memory Usage - where each of its sub-columns shows the max memory usage for a particular heap size as a percentage of the total heap; ii) column PC and PV – represent the Pearson Correlation and corresponding P-value, respectively, which determines the relationship between the variable heap size and the application's max memory usage as described in Section 5.4. The top rows (Table 5.2) refer to the read-intensive (RI) workload of the Cassandra benchmark; the middle rows refers to the balanced (RW) workload, while the bottom rows refer to the write-intensive (WI) workload of the benchmark.

Table 5.3 shows an analogous structure to Table 5.2 as described before, presenting the average memory usage as a percentage of the total heap size instead of the max memory usage. The average memory usage is presented as a 95% confidence interval, which the reason to use was described in the introduction to Section 5.

Regarding max memory usage, both G1 and Shenandoah maintained a constant memory ceiling for all workloads of 88% and 85% of the total heap, respectively. G1 max memory usage is mainly dictated



(a) Shenandoah



(b) ZGC

Figure 5.12: Memory Footprint and Collection Frequency Comparison between Shenandoah and ZGC

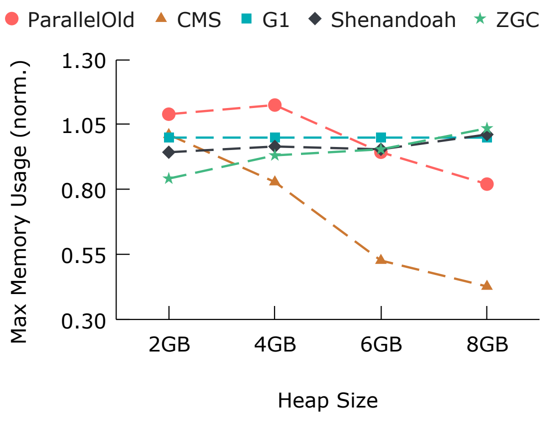
Table 5.3: Average Memory Usage (% of the heap size) for Cassandra Workloads

		$Average\ Memory\ Usage$ (%)					
WL	GC	$2GB$ $Heap$	$4GB$ $Heap$	$6GB$ $Heap$	$8GB$ $Heap$	PC	PV
RI	$ParallelOld$	0.45 ± 0.011	0.47 ± 0.020	0.44 ± 0.022	0.36 ± 0.022	-0.828	0.172[1]
	CMS	0.52 ± 0.012	0.40 ± 0.010	0.26 ± 0.007	0.20 ± 0.005	-0.990	0.010[2]
	$G1$	0.50 ± 0.015	0.55 ± 0.030	0.49 ± 0.034	0.39 ± 0.038	-0.741	0.259
	$Shenandoah$	0.54 ± 0.021	0.53 ± 0.038	0.51 ± 0.052	0.49 ± 0.065	-0.995	0.005[2]
	ZGC	0.51 ± 0.004	0.46 ± 0.011	0.46 ± 0.020	0.47 ± 0.036	-0.592	0.408
RW	$ParallelOld$	0.52 ± 0.012	0.43 ± 0.017	0.45 ± 0.024	0.48 ± 0.028	-0.394	0.606
	CMS	0.54 ± 0.012	0.46 ± 0.013	0.42 ± 0.013	0.32 ± 0.010	-0.986	0.014[1]
	$G1$	0.47 ± 0.013	0.47 ± 0.025	0.49 ± 0.032	0.43 ± 0.036	-0.550	0.450
	$Shenandoah$	0.53 ± 0.022	0.52 ± 0.041	0.51 ± 0.055	0.49 ± 0.068	-0.994	0.006[2]
	ZGC	0.52 ± 0.005	0.47 ± 0.014	0.47 ± 0.027	0.47 ± 0.037	-0.714	0.286
WI	$ParallelOld$	0.51 ± 0.012	0.45 ± 0.017	0.44 ± 0.023	0.45 ± 0.028	-0.729	0.271
	CMS	0.55 ± 0.013	0.50 ± 0.015	0.44 ± 0.017	0.44 ± 0.015	-0.948	0.052
	$G1$	0.46 ± 0.013	0.43 ± 0.022	0.43 ± 0.028	0.46 ± 0.035	0.110	0.890
	$Shenandoah$	0.52 ± 0.025	0.52 ± 0.044	0.50 ± 0.060	0.49 ± 0.075	-0.982	0.018[1]
	ZGC	0.51 ± 0.006	0.45 ± 0.014	0.47 ± 0.030	0.47 ± 0.039	-0.576	0.424

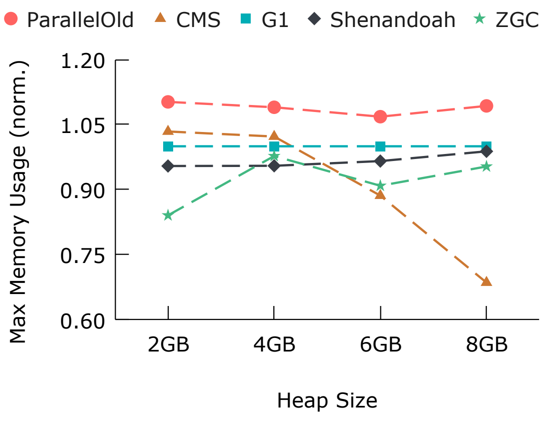
[1] Correlation is significant at the 0.05 alpha level (2-tailed).
[2] Correlation is significant at the 0.01 alpha level (2-tailed).

by the threshold at which concurrent marking is triggered and the flag ”G1ReservePercent”, which sets the percentage of reserved memory to keep free (the default value is 10%) to reduce the risk of to-space overflows. Shenandoah makes a careful decision to balance the garbage collection frequency and its memory footprint. By default, Shenandoah runs with adaptive heuristics which targets to use the entirety of the heap to its fullest. It does so by observing the previous GC cycles and tries to start the next GC cycle so that it could complete before the heap is exhausted (see Figure 5.12(a)). This approach results in less garbage collection cycles when compared to ZGC, which is also shown in Figure 5.12. However, a natural disadvantage regarding this approach is the higher memory footprint, which is one of the reasons why Shenandoah showed higher average and max memory usage than ZGC across all workloads, as seen in Tables 5.2 and 5.3. The other reason was due to the necessity of a forward pointer (usually one word per object) for Shenandoah to perform concurrent compaction. In the worst case, for objects with no payload (only two-word objects), requiring every object to have a word to store the forward pointer translates into a 50% increase in memory. However, for the Cassandra benchmark, which employed more realistic workloads, Shenandoah registered, on average, a 2-3% higher memory usage over ZGC, that also performs concurrent compaction not requiring a forward pointer. However, it is relevant to note that Shenandoah showed a significant negative correlation between average memory usage and heap size, which implies that as the heap size continues to increase, the memory usage will decrease.

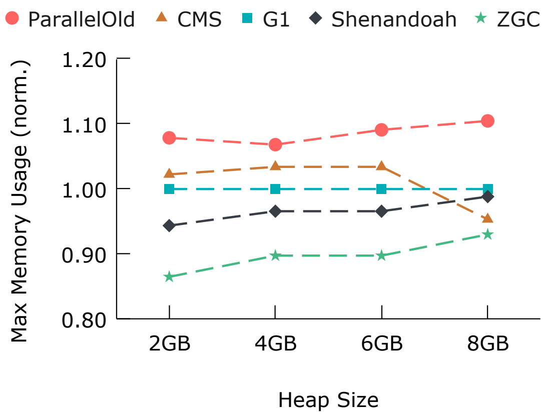
ZGC contrary to Shenandoah, showed a clear preference for a lower memory footprint, therefore,



(a) Max Memory Usage for RI Workload



(b) Max Memory Usage for RW Workload



(c) Max Memory Usage for WI Workload

Figure 5.13: Max Memory Usage (normalized to G1) for Cassandra Workloads

increasing the garbage collection frequency and presenting a significantly higher number of GC cycles when compared to Shenandoah, as shown in Figure 5.12(b). Even though an increase in heap size showed an increase in max memory usage, ZGC managed to maintain a lower memory footprint than Shenandoah for all experiments.

As for ParallelOld and CMS, they showed a slightly superior average and max memory usage than the other collectors for smaller heap sizes. With ParallelOld and CMS, the old generation fills up until a full collection or a mostly concurrent collection are triggered, respectively. CMS mostly concurrent collections are triggered at a 90% occupancy rate of the old generation (by default) or started with the aim of completing the collection cycle before the old generation is exhausted, which explains CMS slightly lower memory footprint when compared to ParallelOld. However, in CMS, the concurrent marking cycle is triggered a bit later than in G1, resulting in a higher memory footprint. For larger heap sizes, ParallelOld and CMS showed significant decreases in max and average memory usages, which was due to the workloads not being able to allocate objects fast enough for the memory ceiling to be reached. Allocation rate is a main factor in determining how fast the memory ceiling is reached, which explains why the low max memory usages mostly occurred with the read-intensive workload on larger heap sizes.

As the heap size increases, when the heap size is 2x the size of the working set (2GB), CMS shows a lower memory footprint than both ZGC and Shenandoah for the RI workload; for the RW workload, CMS shows a lower memory footprint at a heap size 3x the size of the working set; and lastly, for the WI workload, it does so at 4x the size of the working set. ParallelOld only managed to surpass the allocation rate for the RI workload with a heap size 3x the size of the working set. Figure 5.13 shows the max memory usages (normalized to G1) for all workloads with which we evaluated the Big Data Platforms. The threshold at which ParallelOld and CMS managed to outdo the allocation rate and consequently present lower memory footprints than the other collectors for the different Cassandra workloads are shown in Figures 5.13(a) to 5.13(c).

5.4.1.C Latency

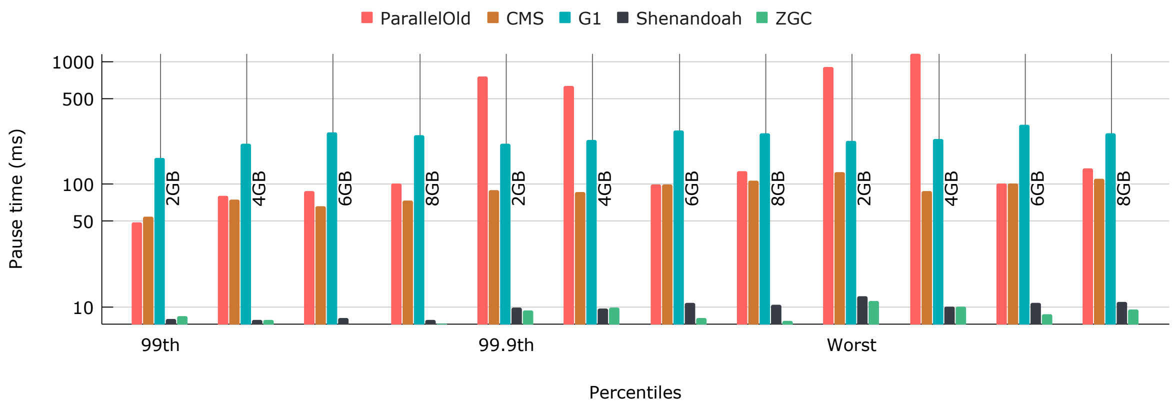
Figure 5.14 shows, on a logarithmic scale, the GC pause times for the different workloads with which we benchmarked Cassandra. For each plot, we show the latency results for the selected garbage collectors with different heap sizes, across multiple percentiles (99th, 99.9th percentiles, and worst pause times).

Regarding the concurrent collectors, Shenandoah showed, as was expected, pauses of only up to 12ms for all workloads. However, as shown in Figures 5.14(b) and 5.14(c), ZGC, for workloads with a higher percentage of writes (i.e., RW and WI), showed pause times of up to 50ms.

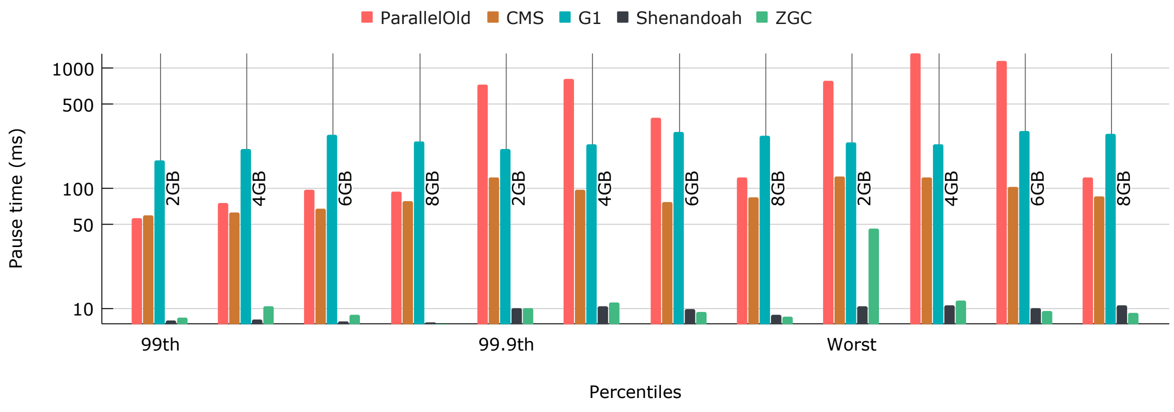
The generational collectors (i.e., ParalellOld, CMS and G1) showed a slight increase in pause times as the percentage of write operations increased (i.e., RW and WI workloads show higher pause times than RI, and the WR shows lower pause times than the WI). As the percentage of write operations increase, so does the speed at which the memory ceiling is reached, triggering a full collection of the heap which are responsible for most lengthy GC pause times in ParallelOld. CMS showed lower pause times than ParallelOld by performing mostly concurrent collections, which allowed CMS to avoid falling back to full collections of the heap. G1 showed pause times higher than CMS and ParallelOld, mainly caused by a large number of humongous objects in the Cassandra workloads (as shown in Figure 5.15) and lack of G1 specific tuning (G1 targets by default 200ms pause times).

Figure 5.15 shows for all the Big Data workloads in our evaluation, the rate of G1 humongous regions. This rate is calculated as the number of humongous regions divided by the number of non-humongous region in the old generation for the different evaluated heap sizes.

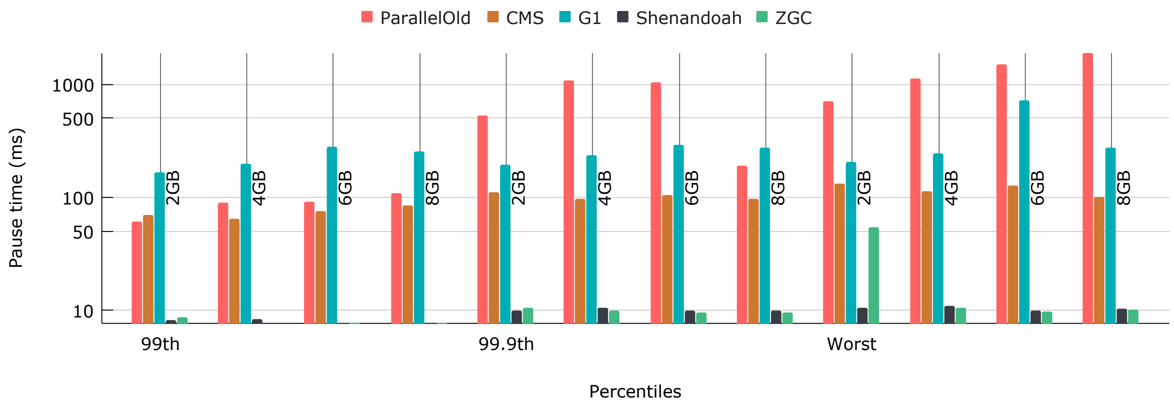
When performing the same evaluation with Cassandra but changing G1 to target pause times similar



(a) Pause Time Percentiles (ms) for Cassandra RI Workload



(b) Pause Time Percentiles (ms) for Cassandra RW Workload



(c) Pause Time Percentiles (ms) for Cassandra WI Workload

Figure 5.14: Pause Time Percentiles (ms) for Cassandra Workloads

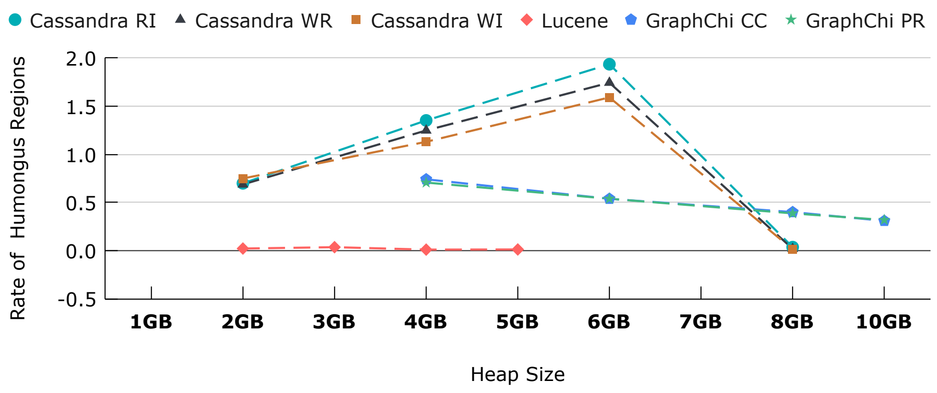
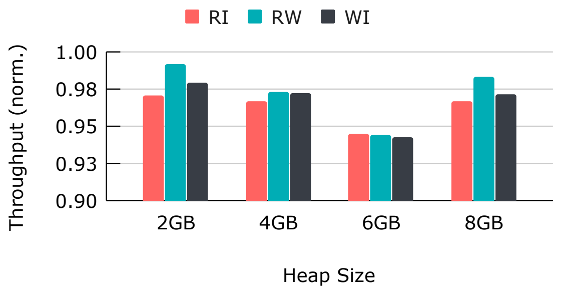
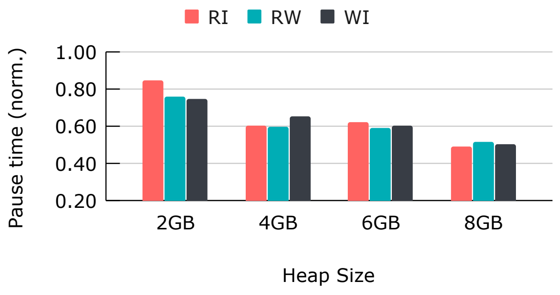


Figure 5.15: Rate of G1 Humongous Regions per non-Humongous Region for the Big Data Platforms Workloads



(a) Application Throughput (norm.) for G1 targetting 50ms pause times



(b) 99th Percentile GC pause times (norm.) for G1 targetting 50ms pause times

Figure 5.16: Application Throughput and GC pause times for G1 targetting 50ms pause times (normalized to G1 targetting 200ms pause times)

to CMS and ParallelOld (50ms), G1 showed to be able to reduce 99th percentile pause times by 16% in smaller heap sizes and up to 50% in larger heap sizes. However, targeting lower pause times presented an overhead on throughput, as shown in Figure 5.16(a), where G1 shows a reduction in application throughput of 3% for smaller heap sizes and up to 6% in larger heap sizes. Figures 5.16(a) shows G1 application throughput for each Cassandra workload with G1 targeting 50ms pause times. All values are normalized to the same experiment but with G1 targeting 200ms pause times. Even though G1 managed to reduce pause times significantly, it still shows higher pause times than all other collectors (up to 138% in smaller heap sizes and 25% in larger heap sizes). We estimate G1 to be able to show pause times similar to ParallelOld at 10-12GB heap sizes, which represents 5-6x the size of the working set.

As for the scalability of the generational collectors, all showed an increase in the 99th percentile of GC pause times as the heap size increased. As the heap size increases, so does the size of the young generation for the ParallelOld, which results in slightly longer but still small collections. However, if a full garbage collection is triggered (which occurred for the experiments with high max memory usages with ParallelOld), a trace of the whole heap is required to reclaim unreachable objects and perform compaction of the old generation. The time it takes to perform this endeavour is proportional to the size of the whole heap, which explains the increase in worst GC pause times as the heap size increases for ParallelOld. On average, ParallelOld, CMS and G1 showed a 23%, 9% and 16% increase in 99th percentile GC pause times, respectively, as the heap size increased. However, G1 seems to reduce the GC pause times across all percentiles when the heap size is at least 3-4x the size of the working set which is related to a drastic decrease in the number of humongous regions. This decrease in the number of humongous regions is due to G1 increasing the regions' sizes as the heap size increases, which also automatically increases the threshold size for a object to be placed in a humongous region.

This reduction in humongous regions is shown in Figure 5.15 where the Cassandra workloads (i.e., RI, RW and WI) show a drastic reduction in the rate of humongous regions between the experiments with 6GB and 8GB heap sizes.

5.4.2 Lucene

Regarding the evaluation with Lucene, as described in Section 4.2.1, it is an example of a Big Data Storage Platform and consequently a latency-oriented application. For the Lucene benchmark, we use a workload comprised of 20000 writes (document updates) and 5000 reads (document searches) per second, which represents an example of a worst-case scenario for a garbage collector latency metric (as this is a write-intensive computation). On top of Lucene, we perform client searches while continuously updating the index (read and write transactions), and since these are done in separate java virtual machines we were limited to half the available memory for each virtual machine. Hence, we were only

Table 5.4: Application Throughput (norm. to G1 at the smallest heap size) for Lucene Workload

GC	*Application Throughput* $2GB$ $Heap$	$3GB$ $Heap$	$4GB$ $Heap$	$5GB$ $Heap$	PC	PV
$ParallelOld$	0.87 ± 0.023	0.89 ± 0.014	1.05 ± 0.013	1.16 ± 0.010	0.969	0.031[1]
CMS	1.20 ± 0.015	1.07 ± 0.011	1.16 ± 0.012	0.88 ± 0.016	-0.797	0.203
$G1$	1.00 ± 0.008	0.80 ± 0.008	1.01 ± 0.009	1.23 ± 0.013	0.665	0.335
$Shenandoah$	0.86 ± 0.009	1.03 ± 0.010	1.07 ± 0.011	1.15 ± 0.007	0.957	0.043[1]
ZGC	0.86 ± 0.006	1.10 ± 0.005	0.83 ± 0.006	0.73 ± 0.006	-0.530	0.470

[1] Correlation is significant at the 0.05 alpha level (2-tailed).

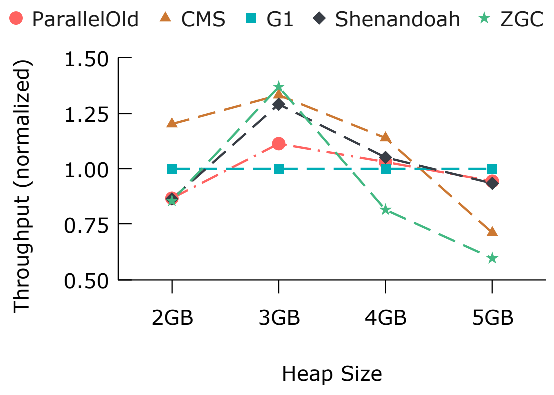


Figure 5.17: Application Throughput (normalized to G1) for Lucene Workload

able to evaluate Lucene with 2, 3, 4, and 5GB total heap size for each of the virtual machines performing client searches and index updates.

5.4.2.A Application Throughput

Table 5.4 shows the throughput results for the Lucene workload. Column wise, it presents an identical structure to Table 5.1 described in Section 5.4.1.A. All application throughput values in Table 5.4 are normalized to G1 at a 2GB heap size so that we can not only compare throughput variances between the collectors for a particular heap size but also across different heap sizes.

From Table 5.4, we conclude that ParallelOld and Shenandoah show the most potential for scalability with the Lucene workload. Both collectors show similar throughput values and a significant positive correlation between throughput and heap size, displaying on average an improvement of 10% on throughput as the heap size grows. As for G1, it shows similar improvements as ParallelOld and Shenandoah; however, it does not do so consistently, showing along with CMS and ZGC contradictory throughput values for a 3GB heap size. CMS shows the highest throughput for smaller heap sizes; however, as with G1, it

Table 5.5: Max Memory Usage (% of the heap size) for Lucene Workload

GC	$Max\ Memory\ Usage$ (%)					PC	PV
	$2GB$ $Heap$	$3GB$ $Heap$	$4GB$ $Heap$	$5GB$ $Heap$	$Mean$		
$ParallelOld$	0.75	0.65	0.63	0.98	0.75 ± 0.16	0.538	0.462
CMS	0.66	0.91	0.49	0.87	0.73 ± 0.20	0.129	0.871
$G1$	0.90	0.88	0.80	0.74	0.83 ± 0.07	-0.982	0.018[1]
$Shenandoah$	0.87	0.86	0.86	0.86	0.86 ± 0.00	-0.925	0.075
ZGC	0.90	0.78	0.82	0.84	0.83 ± 0.05	-0.332	0.668

[1] Correlation is significant at the 0.05 alpha level (2-tailed).

Table 5.6: Average Memory Usage (% of the heap size) for Lucene Workload

GC	$Average\ Memory\ Usage$ (%)				PC	PV
	$2GB$ $Heap$	$3GB$ $Heap$	$4GB$ $Heap$	$5GB$ $Heap$		
$ParallelOld$	0.54 ± 0.003	0.32 ± 0.005	0.33 ± 0.007	0.35 ± 0.008	-0.692	0.308
CMS	0.52 ± 0.002	0.35 ± 0.001	0.42 ± 0.001	0.81 ± 0.001	0.590	0.410
$G1$	0.60 ± 0.007	0.58 ± 0.013	0.50 ± 0.010	0.44 ± 0.015	-0.985	0.015[1]
$Shenandoah$	0.49 ± 0.007	0.48 ± 0.011	0.48 ± 0.016	0.46 ± 0.016	-0.940	0.060
ZGC	0.29 ± 0.001	0.29 ± 0.004	0.35 ± 0.007	0.38 ± 0.011	0.918	0.082

[1] Correlation is significant at the 0.05 alpha level (2-tailed).

shows inconsistent variances as the heap size increases. However, it is relevant to note that contrary to G1, CMS and ZGC show a tendency to present a negative correlation between its throughput and heap size.

5.4.2.B Memory Usage

Figure 5.18 shows the various GCs max memory usage evolution (normalized to G1) for the different heap sizes with which we evaluated Lucene. The Lucene workload shows an opposite memory footprint compared to the Cassandra workloads. After investigating the GC logs, we noticed that with the Lucene workload, aside from the initial promotion of the working set objects to the old generation, newly allocated objects were no longer surviving enough garbage cycles to be promoted. This finding shows that the Lucene workload strongly follows the weak generational hypothesis, stating that most objects die young, which explains the lower memory footprint with ParallelOld and CMS, as the memory usage in these circumstances is limited to the size of the young generation. CMS shows lower max memory usages than ParallelOld and all other collectors; this is due to CMS not increasing the size of the young generation (by default) as the heap size grows, contrary to ParallelOld that proportionally increases the size of the young generation with the size of the heap.

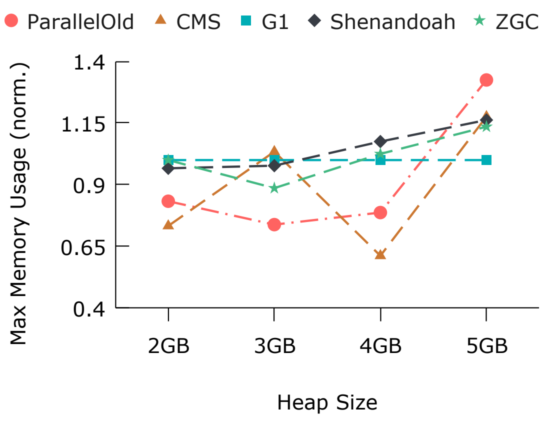


Figure 5.18: Max Memory Usage (normalized to G1) for Lucene Workload

Regarding the concurrent collectors (i.e., Shenandoah and ZGC), identically to the Cassandra workloads, Shenandoah shows steadily up to 3% higher max memory usages across all heap sizes than ZGC due to default adaptive heuristics (described in Section 5.4.1.B) and the overhead imposed by brooks pointers (described in Section 2.5.5).

Regarding the memory footprint sensitivity to an increase in heap size, G1 is the only collector that shows significant negative correlations for both average and max memory usage, meaning that as the heap increases, G1 will tend to decrease its memory footprint. Although G1 shows the highest average and max memory usage for the smallest heap size in our evaluation, it shows to be able to reduce its memory footprint below all other collectors at 5x (5GB) the working set size (1GB). Shenandoah also shows a tendency to reduce its memory footprint slightly as the heap increases; however, due to the small decreases in memory footprint, it fails to register a significant correlation. Regarding the remaining collectors (i.e., ParallelOld, CMS and ZGC), they all show an inconclusive relationship between memory footprint and variance in heap size.

5.4.2.C Latency

Figure 5.19 shows on a logarithmic scale, the evolution of the GC pause times (on various percentiles) for the different evaluated collectors as we increase the heap size. When compared to Cassandra workloads (see Figure 5.14), Lucene shows the same behaviour as the heap size increases for all the collectors. However, the length of the GC pause times is drastically reduced when compared to Cassandra. This reduction occurs due to two factors: i) a good spatial locality of unreachable objects in the old generation, causing less fragmented heaps, which take considerably less effort (and therefore time) to collect than a highly fragmented heap caused by poor spatial locality; ii) a drastic reduction in the size of the working set objects, which is shown in Figure 5.15 (the number of G1 humongous regions

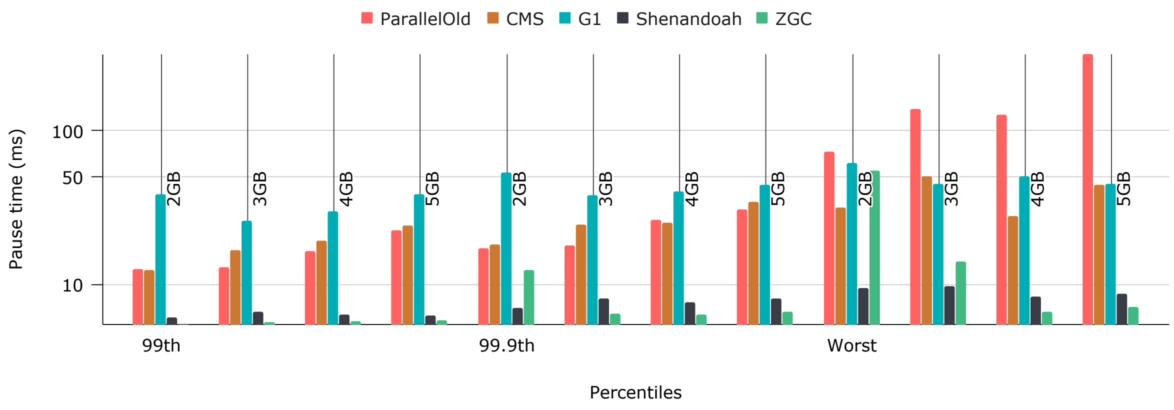


Figure 5.19: Pause Time Percentiles (ms) for Lucene Workload

never surpasses 3% the number of old regions).

Similarly to the WI workload, ZGC under the Lucene workload showed up to 55ms worst-case GC pause times for the smallest heap size evaluated. Shenandoah, on the other hand, managed to maintain pause times under 10ms across all percentiles as it did with the WI workload (12ms). As for the ParallelOld and CMS collectors, they present significant increases on GC pause times as the heap increases (as was the case with WI workload), with ParallelOld registering higher raises than CMS. For the worst GC pause times, Lucene shows a significant reduction in the length of these pauses when compared to the Cassandra workload due to the spatial locality of unreachable objects, as explained before. ParallelOld, CMS and G1 shows on average a 22%, 25% and 22% increase in the 99th percentile GC pause times, respectively, as the heap size increases.

5.4.3 GraphChi

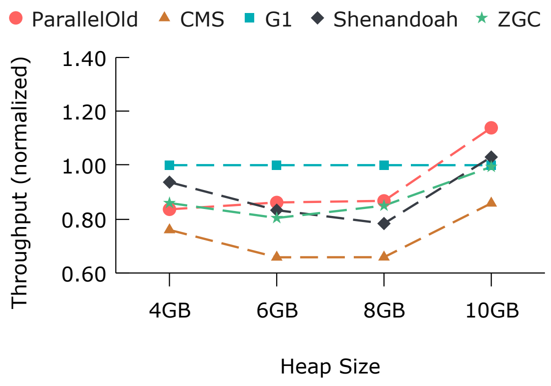
In this Section, we analyze how the different garbage collectors compare to one another regarding an increase in heap size for the GraphChi application. GraphChi is a disk-based system for computing efficiently on graphs with billions of edges in a single system. It is used in our evaluation as an example of a Big Data processing platform and throughput-oriented application. The workloads used to benchmark GraphChi were presented in Section 4.2.2. Regarding the heap size available to both workloads (i.e., CC and PR), we benchmark GraphChi with 4, 6, 8 and 10GB heap sizes for all garbage collectors.

5.4.3.A Application Throughput

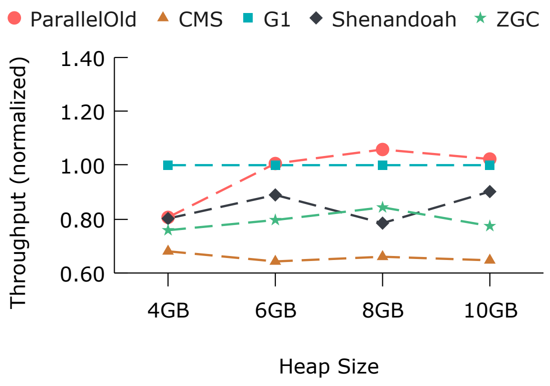
Table 5.7 shows the throughput results for the GraphChi workloads. Column wise, it presents an identical structure to Table 5.1 described in Section 5.4.1.A. The top rows refer to the GraphChi benchmark using the ConnectedComponents algorithm, while the bottom rows refer to the PageRank algorithm. All

Table 5.7: Application Throughput (norm. to G1 at the smallest heap size) for GraphChi Workloads

		Mean Throughput			
WL	*GC*	*2GB Heap*	*4GB Heap*	*6GB Heap*	*8GB Heap*
CC	*ParallelOld*	0.84 ± 0.085	0.88 ± 0.053	0.88 ± 0.051	1.00 ± 0.037
	CMS	0.76 ± 0.056	0.68 ± 0.041	0.67 ± 0.046	0.76 ± 0.062
	G1	1.00 ± 0.031	1.03 ± 0.031	1.02 ± 0.034	0.88 ± 0.048
	Shenandoah	0.94 ± 0.032	0.85 ± 0.039	0.80 ± 0.041	0.91 ± 0.028
	ZGC	0.86 ± 0.022	0.83 ± 0.018	0.86 ± 0.023	0.87 ± 0.021
PR	*ParallelOld*	0.81 ± 0.080	1.07 ± 0.038	1.08 ± 0.036	1.05 ± 0.026
	CMS	0.68 ± 0.048	0.68 ± 0.044	0.67 ± 0.045	0.66 ± 0.045
	G1	1.00 ± 0.029	1.06 ± 0.031	1.02 ± 0.030	1.03 ± 0.033
	Shenandoah	0.80 ± 0.040	0.94 ± 0.029	0.80 ± 0.040	0.93 ± 0.028
	ZGC	0.76 ± 0.030	0.85 ± 0.029	0.86 ± 0.020	0.79 ± 0.029



(a) Application Throughput for GraphChi CC Workload



(b) Application Throughput for GraphChi PR Workload

Figure 5.20: Application Throughput (normalized to G1) for GraphChi Workloads

Table 5.8: Mean Memory Usage Results from the GraphChi Workloads

WL	GC	Average Memory Usage (%) 2GB Heap	4GB Heap	6GB Heap	8GB Heap	PC	PV
CC	$ParallelOld$	0.45 ± 0.021	0.40 ± 0.020	0.29 ± 0.027	0.18 ± 0.034	-0.986	0.014^1
	CMS	0.54 ± 0.020	0.43 ± 0.022	0.42 ± 0.024	0.33 ± 0.027	-0.965	0.035^1
	$G1$	0.39 ± 0.011	0.33 ± 0.012	0.32 ± 0.015	0.30 ± 0.015	-0.936	0.064
	$Shenandoah$	0.52 ± 0.096	0.51 ± 0.124	0.48 ± 0.145	0.49 ± 0.196	-0.885	0.115
	ZGC	0.33 ± 0.041	0.30 ± 0.086	0.28 ± 0.115	0.24 ± 0.124	-0.994	0.006^2
PR	$ParallelOld$	0.47 ± 0.020	0.38 ± 0.021	0.25 ± 0.036	0.21 ± 0.037	-0.981	0.019^1
	CMS	0.50 ± 0.020	0.44 ± 0.021	0.43 ± 0.026	0.41 ± 0.027	-0.930	0.070
	$G1$	0.40 ± 0.010	0.33 ± 0.013	0.34 ± 0.018	0.30 ± 0.016	-0.874	0.126
	$Shenandoah$	0.54 ± 0.077	0.51 ± 0.135	0.48 ± 0.145	0.49 ± 0.196	-0.885	0.115
	ZGC	0.33 ± 0.028	0.31 ± 0.069	0.27 ± 0.110	0.27 ± 0.119	-0.954	0.046^1

[1] Correlation is significant at the 0.05 alpha level (2-tailed).
[2] Correlation is significant at the 0.01 alpha level (2-tailed).

throughput results are normalized to G1 at a heap size of 2GB.

From Table 5.7, we can conclude that G1 significantly outperforms CMS, Shenandoah and ZGC in both workloads, which outlines a different behaviour compared to previous workloads (as explained in the following section). CMS presented the worse throughput results, registering 26% and 34% lower throughput (on average) when compared to G1 across all heap sizes w.r.t. CC and PR workloads, respectively. CMS behavior with GraphChi is similar to the micro-benchmark results. Due to the high allocation rate and the generational hypothesis not occurring, CMS tries to concurrently mark-and-sweep the old generation but ultimately falls back to lengthier full GC of the heap, which impacts the throughput of the application drastically. Regarding the concurrent collectors (i.e., Shenandoah and ZGC), both present similar throughput across the various evaluated heap sizes, with ZGC slightly outperforming Shenandoah. When compared to G1, Shenandoah and ZGC show on average 15% worse throughput across all heap sizes. While the ParallelOld shows similar results to the concurrent collectors in the CC workload, the same does not apply to the PR workload in regards to larger heap sizes where it shows similar throughput results to G1.

Figures 5.20(a) and 5.20(b) show for each GC the throughput evolution regarding heap size for the GraphChi workloads (normalized to G1).

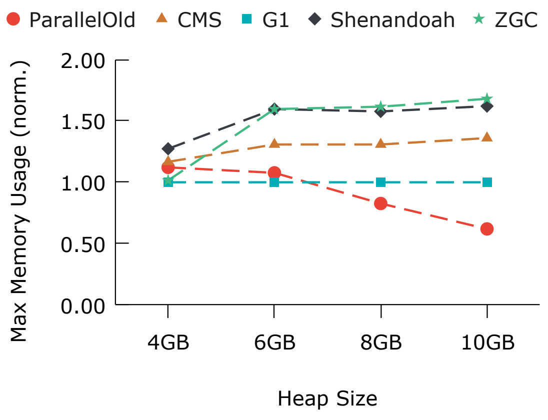
5.4.3.B Memory Usage

Tables 5.8 and 5.9 show for the GraphChi workloads (i.e., CC and PR), the results for the average and max memory usage for each garbage collector as the heap size increases. Column wise, the tables present an identical structure to the Tables 5.2 and 5.3, respectively, as described in Section 5.4.1.B. As

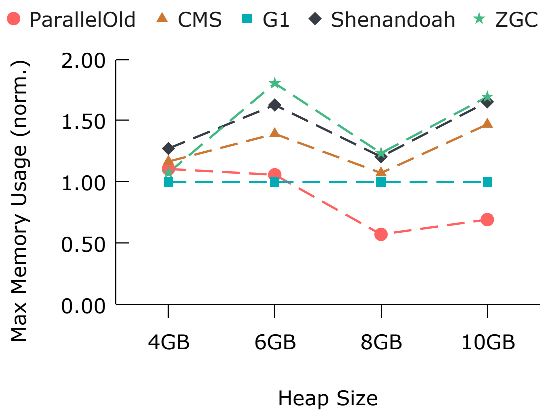
Table 5.9: Max Memory Usage Results from the GraphChi Workloads

WL	*GC*	*Max Memory Usage (%)* 2GB Heap	4GB Heap	6GB Heap	8GB Heap	*Mean*	*PC*	*PV*
CC	*ParallelOld*	0.74	0.56	0.43	0.31	0.51 ± 0.19	-0.996	0.004[2]
	CMS	0.77	0.68	0.68	0.68	0.70 ± 0.05	-0.777	0.223
	G1	0.66	0.52	0.52	0.50	0.55 ± 0.07	-0.846	0.154
	Shenandoah	0.84	0.83	0.82	0.81	0.82 ± 0.01	-0.973	0.027[1]
	ZGC	0.67	0.83	0.84	0.84	0.80 ± 0.08	0.799	0.201
PR	*ParallelOld*	0.73	0.54	0.39	0.34	0.50 ± 0.18	-0.972	0.028[1]
	CMS	0.77	0.71	0.73	0.72	0.73 ± 0.03	-0.665	0.335
	G1	0.66	0.51	0.68	0.49	0.59 ± 0.10	-0.440	0.560
	Shenandoah	0.84	0.83	0.82	0.81	0.83 ± 0.01	-0.962	0.038[1]
	ZGC	0.71	0.92	0.84	0.83	0.83 ± 0.09	0.406	0.594

[1] Correlation is significant at the 0.05 alpha level (2-tailed).
[2] Correlation is significant at the 0.01 alpha level (2-tailed).



(a) Max Memory Usage for GraphChi CC Workload



(b) Max Memory Usage for GraphChi PR Workload

Figure 5.21: Max Memory Usage (normalized to G1) for GraphChi Workloads

for the rows, the top rows (Tables 5.8 and 5.9) refer to the GraphChi benchmark using the Connected-Components (CC) algorithm, while the bottom rows refer to the PageRank (PR) algorithm.

From Tables 5.8 and 5.9, we can conclude that regarding the memory footprint of the different collectors, GraphChi shows for both workloads, a completely different memory behaviour when compared to Cassandra except with the Shenandoah collector. Shenandoah shows similar memory footprints as in Cassandra and Lucene, since it maintains the same behaviour described in Section 5.4.1.B. Shenandoah has a clear preference for a higher memory footprint, always presenting max and average memory usages of 80-85% and 50-55% of the heap's size, respectively.

With G1, very large objects are directly allocated in the old generation, which has the clear disadvantage that these objects are collected later than they would be if allocated in the young generation (causing a higher memory footprint). As shown in Figure 5.15, GraphChi mainly allocates small objects, representing the vertexes and edges described in Section 4.2.2, to be used in a single iteration. Since a single iteration is not long enough for these objects to be promoted to the old generation (contrary to what happens with ParallelOld and CMS), combined with a reduction in large object allocations explains the decrease in G1's average and max memory usages. For the ParallelOld and CMS, since the collection of these objects is delayed until a full collection or mostly concurrent marking is triggered, respectively, it leads to an increase in the average and max memory usages compared to G1.

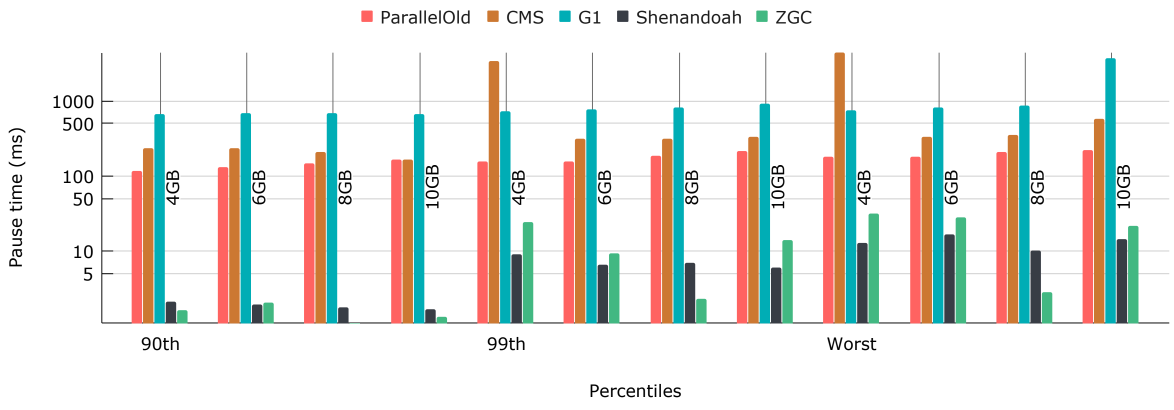
Regarding the memory usage sensitivity to a variance in heap size, for both workloads, all collectors show similar variances in memory usage as the heap size increased (as shown in Figures- 5.21(a) and 5.21(b)). ParallelOld shows the most sensitivity to an increase in heap size, presenting on average a 25% and 23% decrease in memory usages w.r.t CC and PR workloads, respectively. Regarding the remaining collectors, they show a decrease between 1% and 8% (on average) for both workloads as the heap size increased.

5.4.3.C Latency

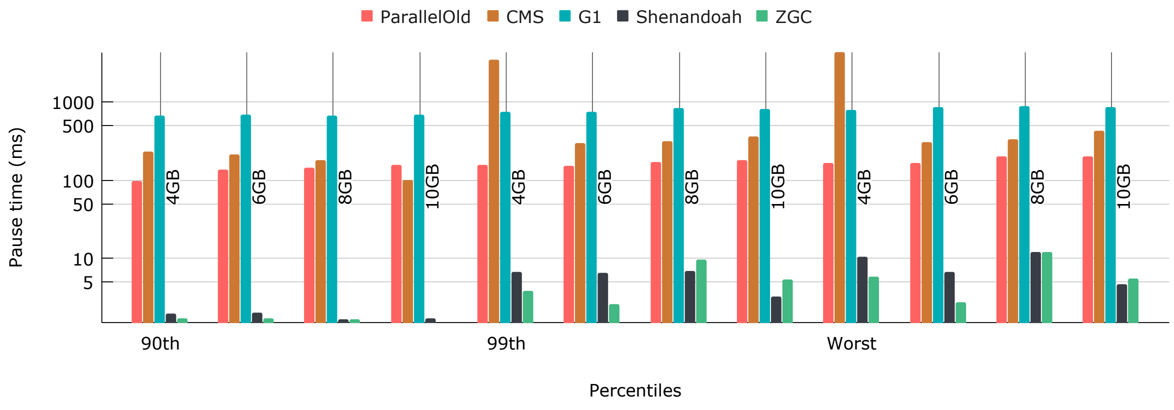
Figures 5.22(a) and 5.22(b) show on a logarithmic scale, the latency profile for each garbage collector as the heap size increases for the ConnectedComponents and PageRank workloads, respectively.

Regarding GC pause times, the concurrent collectors (i.e., Shenandoah and ZGC) managed to keep all pause times under 12ms (across all percentiles) for the PageRank workload of the GraphChi benchmark. For the ConnectedComponents workload, ZGC shows pause times of up to 31ms and an increase in latency across all percentiles. A superior throughput for ZGC over Shenandoah on the CC workload comes at the cost of lengthier pause times and an inability to maintain ultra-low pause times (under 10ms).

The generational collectors show similar small increases in GC pause times as the heap size increases. However, when comparing the GC pause times between Cassandra/Lucene and the GraphChi



(a) Pause Time Percentiles (ms) for GraphChi CC Workload



(b) Pause Time Percentiles (ms) for GraphChi PR Workload

Figure 5.22: Latency Percentiles from the GraphChi Workloads

workloads, the later shows significantly lengthier pause times across all percentiles. This increase is most notable on CMS, where the time to promote objects from the young generation to the old generation is the leading reason for the lengthy pause times, caused by a high-fragmentation of the heap due to an increase in mostly-concurrent collections of the old generation.

5.5 Discussion

In this section, we try to answer the following question: given an application with specific characteristics, which is the GC that best suits that application out of the box? According to the study, we can divide the applications into two types, storage platforms (e.g., Cassandra and Lucene) and processing platforms (e.g., GraphChi).

The results presented with Cassandra and Lucene, for storage platforms, show that if we want to minimize the latency metric, ZGC and Shenandoah both guarantee the 99.9th percentile of STW GC pauses below 10 ms. However, ZGC showed worst-case pauses of up to 50ms, which may require some profiling and tuning. If allocating more memory is not a constraint, ZGC is the best collector as it surpasses the other collectors' throughput while maintaining all pauses under 10ms. If we want to maximize the throughput metric, we have to consider the size of the objects we are dealing with to make a choice. Suppose the workload is mostly comprised of small objects. In that case, Lucene's results show that Shenandoah and the ParallelOld are the best collectors if memory is not a constraint as they appear to scale better than the other collectors. If memory is a limitation, G1 and CMS show the highest throughput; however, its scalability could not be ascertained. Suppose the workload comprises variable-sized objects where the percentage of humongous regions with G1 is superior to 50%. In that case, the ParallelOld is the most suitable GC according to our results with Cassandra. If memory is not a constraint, then ZGC quickly surpasses the ParallelOld as the heap increases.

For processing platforms, the results with GraphChi shows significant similarities with our micro-benchmark. Both workloads allocate small objects that are mostly guaranteed to be promoted to the old generation, which show that the CMS is the collector who suffers the most in performance as the old generation will eventually fall back to a full GC for Big Data applications whose objects do not behave according to the generational hypothesis. According to our results, if we want to maximize the throughput metric, then G1 is the most suitable GC for our application if memory is not a limitation. Otherwise, ParallelOld shows better throughput results for larger heap sizes while showing smaller STW GC duration, as a result of more memory resulting significantly in less full garbage collections of the heap. If we want to minimize the latency metric, Shenandoah guarantees pauses under 10ms, while ZGC guarantees pauses of less than 30ms with similar throughput results. However, it is important to remember that Shenandoah has a memory footprint superior to ZGC.

For smaller heap sizes, according to our results with the Dacapo benchmark suite, which is composed of sub-benchmarks that simulate real-world workloads that focus on different performance features, we found that the concurrent collectors on average perform 21%-26% worse than the generational collectors throughput-wise. This finding is in agreement with our evaluation with ZGC and Shenandoah (see Section 5.2), where we confirmed a slight throughput overhead with the memory barriers that allow Shenandoah and ZGC to perform concurrent compaction. We found that both collectors were inconsistent with maintaining pauses under 10ms for the various benchmarks for the latency metric. Both collectors reached worst-case pauses of up to 50ms (similar to the generational collectors) in several benchmarks. Regarding memory footprint and latency, in our evaluation with DaCapo, Shenandoah shows a higher sensitivity to smaller heap sizes than ZGC.

6 Conclusion

Contents

6.1 Concluding Remarks

We have seen an increase in the number of languages that run on top of runtime systems in recent years. Some examples of widely adopted languages that run on top of such runtimes are JavaScript, Java, C#, Scala, Python, and Go. The widespread use of such languages shows that application developers want to take advantage of all the benefits of using a runtime system and show that current runtimes' design is mature, providing competitive performance compared to traditional languages such as C or C++. Therefore, considering this, we foresee that runtime system utilization will continue to grow in the future, suggesting the need for more research in this area (such as the one presented in this work). In previous chapters, we presented an overview of modern garbage collectors used to memory manage Big Data applications and provide a comparison of throughput vs latency vs memory of these collectors in Big Data Environments. To sum up, in Chapter 1, we introduced the problem, showing the motivation behind this work and what it tries to address. Chapter 2 presented fundamental background concepts such as i) an overview of the OpenJDK HotSpot JVM architecture; ii) composition of Big Data environments and iii) an overview of automatic memory management. After, in Chapter 3, it was presented the existing work in the literature from which we desire to improve upon with this thesis. Chapter 4 describes the mains aspects of the solution used to compare the different GCs. And finally, in Chapter 5, we present and discuss the results from the experiments performed upon the garbage collectors, giving hints to which garbage collector implementation is most suitable to fulfil its requirements (w.r.t application throughput, latency and memory usage).

In summary, for smaller heap sizes, current (classic) garbage collectors are still de facto most suited collectors. ZGC and Shenandoah significant tradeoff in lower throughput does not pay off, as it presents comparable pause times to the generational collectors for small heaps. However, for Big Data applications, ZGC and Shenandoah are always the best collectors if we want to minimize the latency metric. They present significant reductions in pause times compared to the generational collectors, independently of the heap size. ZGC and Shenandoah also show that if memory is not a constraint, they scale better than the generational collectors, providing in most cases higher or comparable throughput. Otherwise, if we want to maximize the throughput metric under a memory constraint, we found that the generational collectors are the still better option. However, we have to consider the size of objects in the working set, if the workload follows the generational hypothesis, etc., to decide which particular GC to use. For a middle-ground between latency and throughput, we found G1 to be the better collector. However, its performance is highly dependant on the size of objects in the working set and tuning performed.

6.2 Future Work

To conclude our research contribution, we now elaborate on how this work could be expanded and/or used together with new research developments.

Ultra-low pause times GC algorithms (e.g., C4, Shenandoah and ZGC) for Java are becoming more and more common as Big Data continues to grow and more applications require low pause times. As these algorithms are still in development, it is important that more studies are performed to keep up with the development of said algorithms. Since the experimental data was recorded for this thesis, for example, a newer version of Shenandoah no longer requires the extra word for each object, therefore, decreasing Shenandoah's memory footprint and potentially increasing throughput as the memory ceiling would be reached later. Other examples that present major throughput improvements are LRB barriers (JDK-8221766), self-fixing barriers (JDK-8231087), and the multitude of other minor improvements to both barrier code, optimizations around them, and general heuristics goodness added since the start of this thesis. It would be interesting to research how to automate GC benchmarking to cope with the constant stream of updates in newer garbage collectors.

Another interesting research would be identifying the most suitable garbage collector for an application if we know nothing about the application (i.e., the application is a complete black-box, where access patterns, topology and amount of objects at the heap are unknown, etc).

Bibliography

[1] R. Bruno and P. Ferreira, "A study on garbage collection algorithms for big data environments," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, p. 20, 2018.

[2] L. Xu, T. Guo, W. Dou, W. Wang, and J. Wei, "An experimental evaluation of garbage collectors on big data applications," *Proceedings of the VLDB Endowment*, vol. 12, no. 5, pp. 570–583, 2019.

[3] Y. Yu, T. Lei, W. Zhang, H. Chen, and B. Zang, "Performance analysis and optimization of full garbage collection in memory-hungry environments," *ACM SIGPLAN Notices*, vol. 51, no. 7, pp. 123–130, 2016.

[4] P. Pufek, H. Grgic, and B. Mihaljevic, "Analysis of garbage collection algorithms and memory management in java," in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 2019, pp. 1677–1682.

[5] A. Prokopec, A. Rosà, D. Leopoldseder, G. Duboscq, P. Tuma, M. Studener, L. Bulej, Y. Zheng, A. Villazón, D. Simon *et al.*, "On evaluating the renaissance benchmarking suite: Variety, performance, and complexity," *arXiv preprint arXiv:1903.10267*, 2019.

[6] W. Zhao and S. M. Blackburn, "Deconstructing the garbage-first collector," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2020, pp. 15–29.

[7] R. Fitzgerald and D. Tarditi, "The case for profile-directed selection of garbage collectors," *ACM SIGPLAN Notices*, vol. 36, no. 1, pp. 111–120, 2001.

[8] S. Soman, C. Krintz, and D. F. Bacon, "Dynamic selection of application-specific garbage collectors," in *Proceedings of the 4th international symposium on Memory management*. ACM, 2004, pp. 49–60.

[9] J. Singer, G. Brown, I. Watson, and J. Cavazos, "Intelligent selection of application-specific garbage collectors," in *Proceedings of the 6th international symposium on Memory management*. ACM, 2007, pp. 91–102.

[10] D. I., "A First Look into ZGC," 2018. [Online]. Available: https://dinfuehr.github.io/blog/a-first-look-into-zgc/

[11] M. Chen, S. Mao, and Y. Liu, "Big data: A survey," *Mobile networks and applications*, vol. 19, no. 2, pp. 171–209, 2014.

[12] T. White, *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[13] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[14] J. Hunt and F. Long, "Java's reliability: An analysis of software defects in Java," *IEE Proceedings-Software*, vol. 145, no. 2, pp. 41–50, 1998.

[15] "Shenandoah SPECjbb2015 performance chart." [Online]. Available: http://dacapobench.sourceforge.net/

[16] "ZGC SPECjbb2015 performance chart." [Online]. Available: http://cr.openjdk.java.net/{~{}}pliden/slides/ZGC-FOSDEM-2018.pdf

[17] "The DaCapo Benchmark Suite." [Online]. Available: http://dacapobench.sourceforge.net/

[18] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[19] M. McCandless, E. Hatcher, and O. Gospodnetic, *Lucene in action: covers Apache Lucene 3.0*. Manning Publications Co., 2010.

[20] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-Scale Graph Computation on Just a \${\$PC\$}\$," in *Presented as part of the 10th \${\$USENIX\$}\$ Symposium on Operating Systems Design and Implementation (\${\$OSDI\$}\$ 12)*, 2012, pp. 31–46.

[21] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, "On-the-fly garbage collection: an exercise in cooperation," *Communications of the ACM*, vol. 21, no. 11, pp. 966–975, 1978.

[22] P. B. Bishop, "Computer systems with a very large address space and garbage collection," Ph.D. dissertation, Massachusetts Institute of Technology, 1977.

[23] M. Wolczko and I. Williams, "Multi-level garbage collection in a high-performance persistent object system," in *Persistent Object Systems*. Springer, 1993, pp. 396–418.

[24] S. M. Blackburn and K. S. McKinley, "Ulterior reference counting: Fast garbage collection without a long wait," in *ACM SIGPLAN Notices*, 2003.

[25] D. F. Bacon and V. T. Rajan, “Concurrent cycle collection in reference counted systems,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2001.

[26] Y. Levanoni and E. Petrank, “An on-the-fly reference counting garbage collector for Java,” *ACM SIGPLAN Notices*, 2011.

[27] R. E. Jones and C. Ryder, “Garbage collection should be lifetime aware,” *Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS’2006)*, 2006.

[28] ——, “A study of Java object demographics,” in *Proceedings of the 7th international symposium on Memory management*. ACM, 2008, pp. 121–130.

[29] L. P. Deutsch and D. G. Bobrow, “An efficient, incremental, automatic garbage collector,” *Communications of the ACM*, vol. 19, no. 9, pp. 522–526, 1976.

[30] D. Ungar, “Generation scavenging: A non-disruptive high performance storage reclamation algorithm,” *ACM Sigplan notices*, vol. 19, no. 5, pp. 157–167, 1984.

[31] H. G. Baker Jr, “List processing in real time on a serial computer,” *Communications of the ACM*, vol. 21, no. 4, pp. 280–294, 1978.

[32] S. Nettles, J. O’Toole, D. Pierce, and N. Haines, “Replication-based incremental copying collection,” in *International Workshop on Memory Management*. Springer, 1992, pp. 357–364.

[33] S. M. Blackburn and A. L. Hosking, “Barriers: Friend or foe?” in *Proceedings of the 4th international symposium on Memory management*. ACM, 2004, pp. 143–151.

[34] X. Yang, S. M. Blackburn, D. Frampton, and A. L. Hosking, “Barriers reconsidered, friendlier still!” *ACM SIGPLAN Notices*, vol. 47, no. 11, pp. 37–48, 2012.

[35] S. M. Blackburn and K. S. McKinley, “In or out?: putting write barriers in their place,” *ACM SIGPLAN Notices*, vol. 38, no. 2 supplement, pp. 175–184, 2002.

[36] “Parallel GC.” [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/parallel.html

[37] *Concurrent Mark Sweep (CMS) Collector*. [Online]. Available: https://docs.oracle.com/javase/7/docs/technotes/guides/vm/cms-6.html

[38] D. Detlefs, C. Flood, S. Heller, and T. Printezis, “Garbage-first garbage collection,” in *Proceedings of the 4th international symposium on Memory management*. ACM, 2004, pp. 37–48.

[39] C. Hunt, B. Rutisson, P. Parhar, and M. Beckwith, *Java Performance Companion*, ser. Addison-Wesley Java series. Addison-Wesley, 2016. [Online]. Available: https://books.google.pt/books?id=41bcoQEACAAJ

[40] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro, "A study of the scalability of stop-the-world garbage collectors on multicores," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 229–240, 2013.

[41] R. Bruno, L. P. Oliveira, and P. Ferreira, "NG2C: pretenuring garbage collection with dynamic generations for HotSpot big data applications," *ACM SIGPLAN Notices*, vol. 52, no. 9, pp. 2–13, 2017.

[42] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen, "NumaGiC: A garbage collector for big data on big NUMA machines," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 661–673, 2015.

[43] N. Cohen and E. Petrank, "Data structure aware garbage collector," *ACM SIGPLAN Notices*, vol. 50, no. 11, pp. 28–40, 2015.

[44] R. Jones, A. Hosking, and E. Moss, *The garbage collection handbook: the art of automatic memory management*. Chapman and Hall/CRC, 2016.

[45] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingam, M. Costa, D. G. Murray, S. Hand, and M. Isard, "Broom: Sweeping out garbage collection from big data systems," in *15th Workshop on Hot Topics in Operating Systems (HotOS ${$XV$}$)*, 2015.

[46] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu, "Facade: A compiler and runtime for (almost) object-bounded big data applications," *ACM Sigplan Notices*, vol. 50, no. 4, pp. 675–690, 2015.

[47] L. Lu, X. Shi, Y. Zhou, X. Zhang, H. Jin, C. Pei, L. He, and Y. Geng, "Lifetime-based memory management for distributed data processing systems," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 936–947, 2016.

[48] M. Maas, K. Asanović, T. Harris, and J. Kubiatowicz, "Taurus: A holistic language runtime system for coordinating distributed managed-language applications," *ACM SIGOPS Operating Systems Review*, vol. 50, no. 2, pp. 457–471, 2016.

[49] D. Patrício, R. Bruno, J. Simão, P. Ferreira, and L. Veiga, "Locality-aware gc optimisations for big data workloads," in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 2017, pp. 50–67.

[50] J. Simão, S. Esteves, A. Pires, and L. Veiga, "Gc-wise: A self-adaptive approach for memory-performance efficiency in java vms," *Future Generation Computer Systems*, vol. 100, pp. 674–688, 2019.

[51] S. Benchmarks, "Standard performance evaluation corporation," 2000.

[52] S. M. Blackburn, M. Hirzel, R. Garner, and D. Stefanovic, “pjbb2005: The pseudojbb benchmark,” *URL http://users. cecs. anu. edu. au/steveb/research/research-infrastructure/pjbb2005*, 2005.

[53] A. W. Appel, “Garbage Collection: Algorithms for Automatic Dynamic Memory Management by Richard Jones and Rafael Lins, John Wiley & Sons, 1996.” *Journal of Functional Programming*, vol. 7, no. 12, pp. 219–225, 1997.

[54] “NoSQL at Netflix.” [Online]. Available: https://netflixtechblog.com/nosql-at-netflix-e937b660b4c

[55] “Meet Michelangelo: Uber’s Machine Learning Platform.” [Online]. Available: https://eng.uber.com/michelangelo

[56] “A Persistent Back-End for the ATLAS Online Information Service (P-BEAST).” [Online]. Available: https://cdsweb.cern.ch/record/1432912

[57] “The Yahoo Cloud Serving Benchmark.” [Online]. Available: https://research.yahoo.com/news/yahoo-cloud-serving-benchmark/

[58] M. Barata, J. Bernardino, and P. Furtado, “Ycsb and tpc-h: Big data and decision support benchmarks,” in *2014 IEEE International Congress on Big Data*. IEEE, 2014, pp. 800–801.

[59] D. Smiley, E. Pugh, K. Parisa, and M. Mitchell, *Apache Solr enterprise search server*. Packt Publishing Ltd, 2015.

[60] H. Kwak, C. Lee, H. Park, and S. Moon, “What is Twitter, a social network or a news media?” in *Proceedings of the 19th international conference on World wide web*. AcM, 2010, pp. 591–600.

[61] R. S. G. Britain), *Proceedings of the Royal Society of London*. Taylor & Francis, 1895, no. v. 58. [Online]. Available: https://books.google.pt/books?id=60aL0zIT-90C