A Performance Comparison of Modern Garbage Collectors for Big Data Environments

Carlos Daniel Oliveira Gonçalves
Instituto Superior Técnico
Lisboa, Portugal
carlos.d.oliveira@ist.utl.pt

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, for 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 Platforms, Java Virtual Machine, Automatic Memory Management

1 INTRODUCTION

There is an increasing need to manage Big Data [5], 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 [21], Spark [24]), 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 unreachable 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 [10].

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 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. The goal of this thesis is to understand how different garbage collectors impact different performance metrics, more particularly latency, throughput, and memory usage. So that with this knowledge, given an application with particular performance requisites, we may give hints on which garbage collector implementation is most suitable to fulfil its requirements.

We expect this research to contribute to the field of modern garbage collectors for Big Data environments through the following contributions: first, provide better knowledge on how to match applications to a specific garbage collector, taking into consideration the application needs and the garbage collector tradeoffs; second, give hints w.r.t. which garbage collector to pick when we want to optimize a particular performance metric; third, an overview of the state of art garbage collectors developed for the Java Virtual Machine and how they improve upon older implementations; and finally, the design and development of a fine-grained benchmark meant to stress-test specific GC components such as read and write barriers.

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

In the next section, we describe the garbage collectors studied and the possible overheads inherent to each implementation. Section 3 describes how recent literature has approached the problem of how to map an application to the best-suited GC and how we hope to improve these works. Section 4 describes the most relevant aspects of the methodology used to evaluate the different collectors, with Section 5 showing the evaluation results. The document ends with the conclusion (Section 6), where we give hints on which GC is best suited for an application w.r.t to latency, memory usage, and throughput.

2 BACKGROUND

To understand how garbage collectors perform, we need first to understand the how they operate and how implementations differ from one another.

2.1 Generational Collectors

Most GCs in this work are generational collectors (e.g., ParallelOld, CMS and G1), which separates the objects in the heap based on their estimated lifetime. This approach bases itself on studies supporting that Java object's lifetime follows a bimodal distribution [11, 12] and the weak generation hypothesis, saying that on most applications, most objects die young [7, 20]. In this work, the generational collectors are divided into two generations, with every object being 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, the work required to move them to the old generation is linear to its small size.

Both the Parallel GC [2] and the Concurrent Mark/Sweep collector [1], use a monolithic (the whole sub-heap must be collected at once) "stop-the-world" copying collector to manage the young generation. For the old generation, the ParallelOld uses a parallel "stop the world" garbage collector with compaction. This approach is usually the most efficient way to maximise the time spent doing application work 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 compaction (usually a function of the size of the Java heap and the number and size of live objects in the old generation) and the monolithic "stop the world" collections. CMS, on the other hand, was developed in response to a growing number of applications that demanded a collector with lower worst-case pause 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. CMS 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. 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 times.

The Garbage-First [6] 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. For the young generation, similar to the previous collectors, G1 employs a parallel "stop the world" copying collector. 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 means that on every old generation collection, all objects in the subset of regions being collected are relocated to unused regions. 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.

Remembered Set. As objects get moved to the old generation, newly created objects in the young generation 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. Instead, a remembered set data structure is used to track outside (old generation) references to objects in the young generation. This, however, presents a performance overhead, as a write barrier is triggered, to update the remembered set, every time a field of an object is updated or a new reference is stored. For G1, 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 an additional 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 [9] when compared to previous collectors.

2.2 Concurrent Collectors

The Z Garbage Collector, also known as ZGC, and Shenandoah, are experimental scalable low-latency collectors built to handle heaps varying from relatively small to potentially multi-terabytes sizes. Compared to G1, both ZGC and Shenandoah are also region-based collectors; however, they are not generational. Instead, they are unigenerational collectors who aim not to exceed 10ms pauses even when increasing the heap or live-set size, by achieving concurrent compaction. ZGC does so through the use of 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. 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, which support the garbage collector correct implementation of concurrent compaction.

Shenandoah, however, 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, 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. 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, 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. Shenandoah uses both read and write barriers.

3 RELATED WORK

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. [8] was the first to present a profiling framework to choose a single most-suitable GC. Soman et al. [19] 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. [18] 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. [22] analyses the correlation between big data applications' memory usage patterns and the collectors' memory usage to obtain findings regarding GC inefficiencies. Yu et al. [23] 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. [25] 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. [17] 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 1: Comparison between the presented papers

Paper	Garbage Collectors	Benchmarks	Performance Metrics
Xu et al. [22]	Parallel Scavenge Con. Mark Sweep Garbage-First	Spark Framework	Memory Usage
Yu et al. [23]	Parallel Scavenge	JOlden Dacapo SPECjvm2008 Spark Framework Giraph Framework	Latency
P. Pufek et al. [17]	Serial Parallel Scavenge Conc. Mark and Sweep Garbage-First	Dacapo	Latency
W. Zhao et al. [25]	Garbage-First	Dacapo SPECjvm98 pjbb2005	Memory Usage Latency Execution time
R. Fitzgerald et al. [8]	Null Copying Gen. Copying	SPECjvm98 MISCJAVA IMPACT DOCTOR	Execution time
S. Soman et al. [19]	Semispace Copying Mark Sweep Gen. Mark Sweep Gen. Semispace Non-gen. Semispace	SPECjvm98 SPECjbb2000 JOlden JavaGrande	Execution time
J. Singer et al. [18]	Copy Mark Sweep Gen. Mark Sweep Gen. Copying Mark Sweep Mark Compact Semispace	SPECjvm98 SPECjbb2000 Dacapo JOlden	Execution time

Table 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

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 2: DaCapo Benchmarks Summary

earlier studies (see Table 1) by using 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 [8, 18, 19], latency [17, 23], or memory usage [22]. 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 academic-oriented JikesRVM.

4 METHODOLOGY

In this section, we start by describing the benchmark suite and workloads used to evaluate the various garbage collectors (i.e., ParallelOld, CMS, G1, Shenandoah and ZGC) presented in Section 2. A combination of real-world and synthetic benchmarks and workloads is used to approximate the results to real-world scenarios. 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 w.r.t 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 to expose the overhead associated.

4.1 DaCapo Benchmark Suite

The DaCapo [3] 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 [17, 23, 25]. A summary of all the workloads present in the DaCapo suite is presented in Table 2.

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 where downtime or data loss is unacceptable (i.e., there is a strong emphasis on latency in these applications). 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.

4.2.1 Cassandra. Apache Cassandra [15] 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. 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 [4] (YCSB) is used to benchmark Cassandra utilising the various garbage collectors.

4.2.2 Lucene. Lucene [16] 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. 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.

4.2.3 GraphChi. GraphChi [14] 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 [13], 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, wherein each iteration, a new group of vertexes is loaded and processed. GraphChi continues to iterate until all vertexes and edges in 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 2.2, 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 (as described in Section 2.1). 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. 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. In Figure 1, we show the life cycle of our micro-benchmark. A detailed description of the micro-benchmark implementation is present in the full document.

5 EVALUATION

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 (see Section 2), more specifically the ParallelOld, CMS , G1, ZGC, and Shenandoah.
(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 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 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.

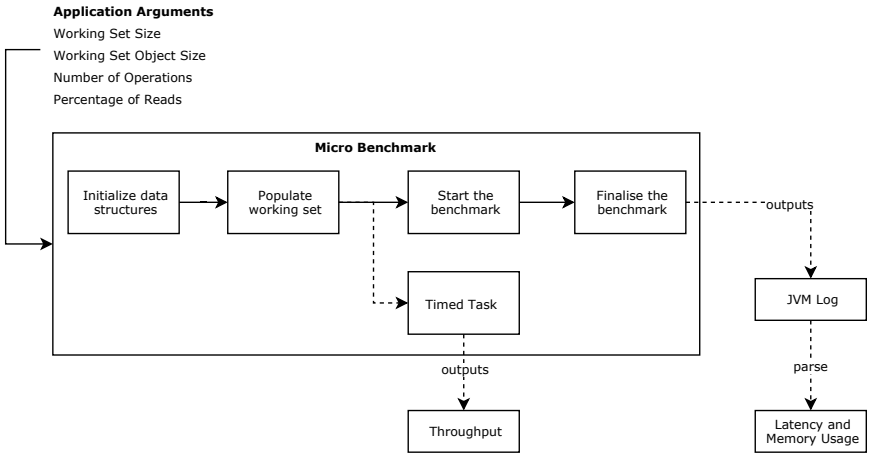

Figure 1: High-level view of our micro benchmark

5.2 Results

5.2.1 Micro Benchmark. 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).

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. 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 2, shows for each garbage collector the application throughput for our workload as we increase the percentage of read operations. The main finding regarding throughput is that 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. 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 lessening the impact on throughput.

As for the concurrent collectors (i.e., Shenandoah and ZGC), these present significantly lower throughput than the generational

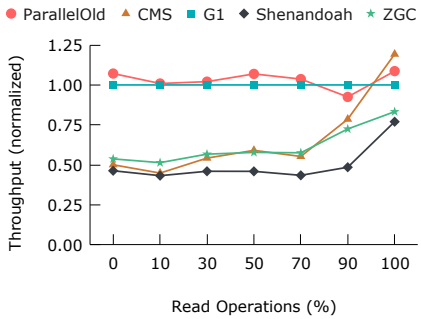

Figure 2: Throughput Results for the Micro Benchmark

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. 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.

With regards to memory usage, when comparing the concurrent (i.e., Shenandoah and ZGC) and generational collectors (except G1), the first one shows 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 2.2 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

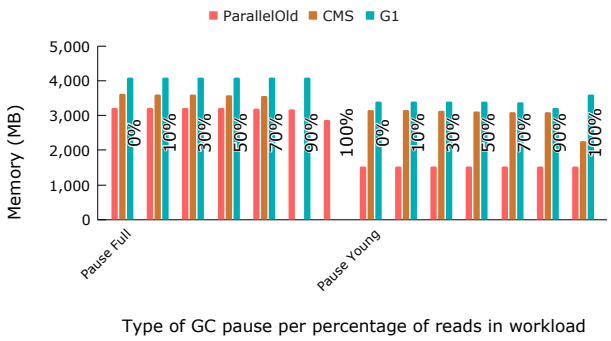

Figure 3: Average Threshold of Memory before GC Pauses per Percentage of Reads in Workload

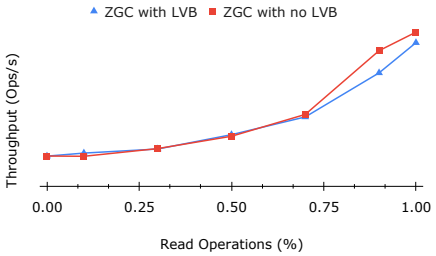

Figure 4: Throughput Results comparing ZGC with LVB and ZGC without LVB

collection so that memory may be made available for objects to be promoted. The mostly-concurrent mark-and-sweep 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 3, 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.

Concurrency Barriers. 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. To measure ZGC read barriers, named load-value barriers (LVB), we 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

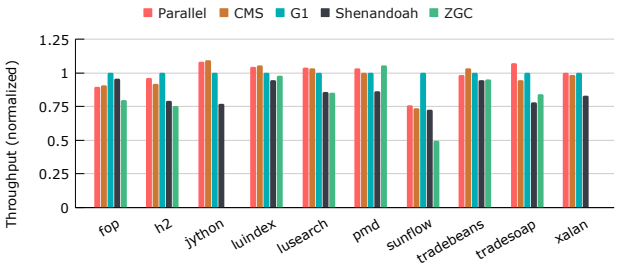

Figure 5: Throughput for the DaCapo Benchmark

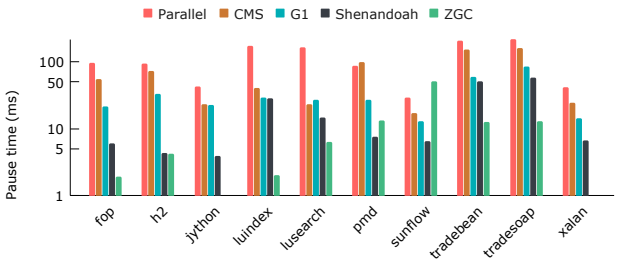

Figure 6: Worst Pause Times for the DaCapo Benchmark

on every read operation). Figure 4 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.

5.2.2 DaCapo 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. 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 for most sub-benchmarks. Figure 5 shows the results for the throughput of the DaCapo Benchmark Suite, normalized to the current default JDK garbage collector G1.

This finding is in agreement with our evaluation with ZGC and Shenandoah (see Figure 4), 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, as show in Figure 6. Regarding memory

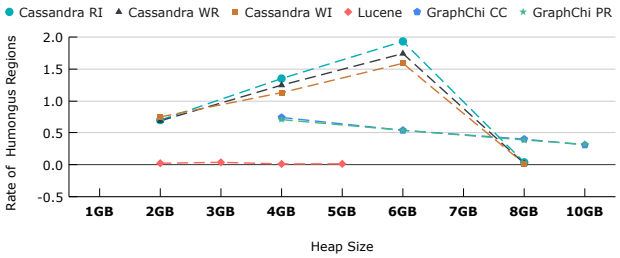

Figure 7: Rate of G1 Humongous Regions per non-Humongous Region for the Big Data Platforms Workloads

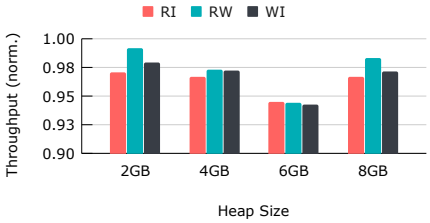

Figure 8: Application Throughput for G1 targetting 50ms pause times (normalized to G1 targetting 200ms pause times)

footprint and latency, in our evaluation with DaCapo, Shenandoah shows a higher sensitivity to smaller heap sizes than ZGC.

5.2.3 Storage 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.

Cassandra and Lucene. The workloads used to benchmark Cassandra were presented in Section 4.2.1, 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. For Lucene we were only able to evaluate Lucene with 2, 3, 4, and 5GB total. 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. 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.

Latency-oriented The results presented with Cassandra and Lucene, both 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. 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 7) and lack of G1 specific tuning (G1 targets by default 200ms pause times). Figure 7 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.

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 for the Cassandra Benchmark. 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 (as shown in Figure 7). Lucene shows the same behaviour as the heap size increases for all the generational 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 7 (the number of G1 humongous regions never surpasses 3% the number of old regions).For the worst GC pause times, 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 for the Lucene workload.

When performing the same evaluation with Cassandra but changing G1 to target pause times similar 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 8, where G1 shows a reduction in application throughput of 3% for smaller heap sizes and up to 6% in larger heap sizes. Figures 8 shows G1 application throughput for each Cassandra workload with G1 targeting 50ms pause times. All values are

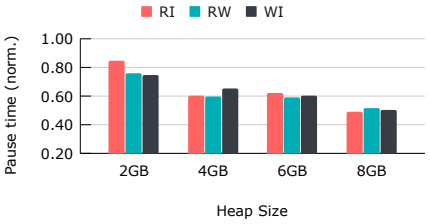

Figure 9: GC pause times for G1 targetting 50ms pause times (normalized to G1 targetting 200ms pause times)

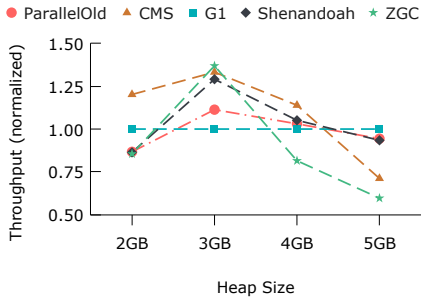

Figure 10: Application Throughput (normalized to G1) for Lucene Workload

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.

Throughput-oriented If we want to maximize the throughput metric, for the storage platforms, 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 (see Figure 10) 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 while maintaining all pauses under 10ms. Figure 11 shows the throughput growth of all collectors for the different evaluated heap sizes for Cassandra RW workload) The other Cassandra workloads (WI and RI) show a similar throughput plot.

5.2.4 Processing Platforms. For processing platforms, the results with GraphChi shows significant similarities with our micro-benchmark. Both workloads allocate small objects that are mostly guaranteed to

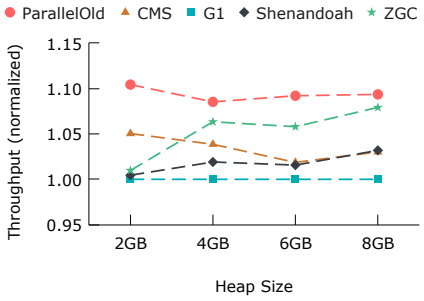

Figure 11: Application Throughput (normalized to G1) for Cassandra RW Workload

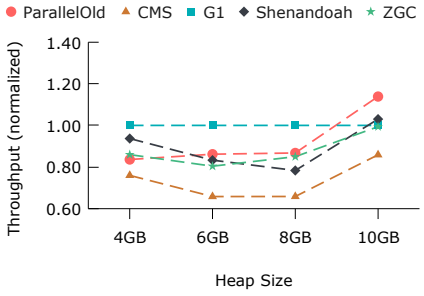

Figure 12: Application Throughput (normalized to G1) for GraphChi CC Workload

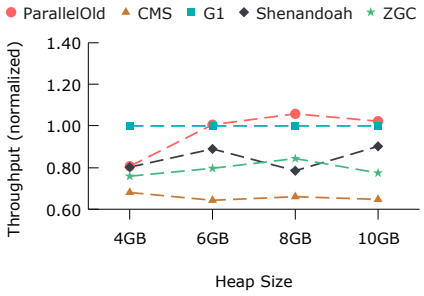

Figure 13: Application Throughput (normalized to G1) for GraphChi PR Workload

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 (as shown in Figures 2, 12 and 13).

Graphchi. GraphChi 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 4.2.3. Regarding the heap size available to both workloads

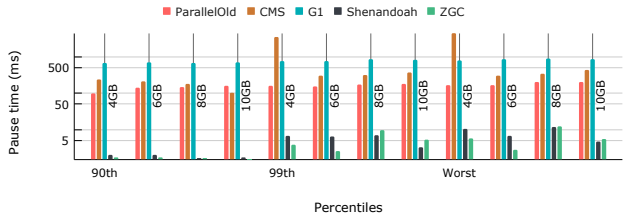

Figure 14: Pause Time Percentiles (ms) for GraphChi CC Workload

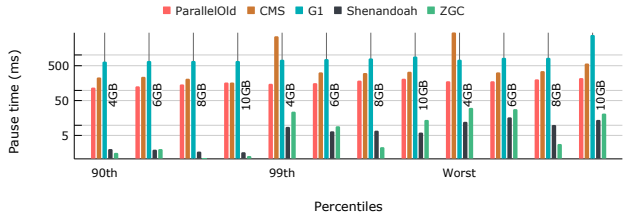

Figure 15: Pause Time Percentiles (ms) for GraphChi PR Workload

(i.e., CC and PR), we benchmark GraphChi with 4, 6, 8 and 10GB heap sizes for all garbage collectors.

Latency-oriented According to our results with GraphChi, for processing platforms, 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.

Throughput-oriented 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. Figures 14 and 15 show on a logarithmic scale, the latency profile for each garbage collector as the heap size increases for the ConnectedComponents and PageRank workloads, respectively.

6 CONCLUSIONS

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).

Answering our primary goal of giving hints on which garbage collector is best suited to fit an application's performance targets, we concluded that current (classic) garbage collectors are still de facto most suited collectors for smaller heap sizes. 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.

REFERENCES

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

[2] [n. d.]. Parallel GC. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/parallel.html

[3] [n. d.]. The DaCapo Benchmark Suite. http://dacapobench.sourceforge.net/

[4] [n. d.]. The Yahoo Cloud Serving Benchmark. https://research.yahoo.com/news/yahoo-cloud-serving-benchmark/

[5] Min Chen, Shiwen Mao, and Yunhao Liu. 2014. Big data: A survey. *Mobile networks and applications* 19, 2 (2014), 171–209.

[6] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management*. ACM, 37–48.

[7] L Peter Deutsch and Daniel G Bobrow. 1976. An efficient, incremental, automatic garbage collector. *Commun. ACM* 19, 9 (1976), 522–526.

[8] Robert Fitzgerald and David Tarditi. 2001. The case for profile-directed selection of garbage collectors. *ACM SIGPLAN Notices* 36, 1 (2001), 111–120.

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

[10] J Hunt and F Long. 1998. Java's reliability: An analysis of software defects in Java. *IEE Proceedings-Software* 145, 2 (1998), 41–50.

[11] Richard E Jones and Chris Ryder. 2006. Garbage collection should be lifetime aware. *Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2006)* (2006).

[12] Richard E Jones and Chris Ryder. 2008. A study of Java object demographics. In *Proceedings of the 7th international symposium on Memory management*. ACM, 121–130.

[13] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th international conference on World wide web*. AcM, 591–600.

[14] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. 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)*. 31–46.

[15] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[16] Michael McCandless, Erik Hatcher, and Otis Gospodnetic. 2010. *Lucene in action: covers Apache Lucene 3.0.* Manning Publications Co.

[17] P Pufek, Hrvoje Grgic, and Branko Mihaljevic. 2019. 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, 1677–1682.

[18] Jeremy Singer, Gavin Brown, Ian Watson, and John Cavazos. 2007. Intelligent selection of application-specific garbage collectors. In *Proceedings of the 6th international symposium on Memory management*. ACM, 91–102.

[19] Sunil Soman, Chandra Krintz, and David F Bacon. 2004. Dynamic selection of application-specific garbage collectors. In *Proceedings of the 4th international symposium on Memory management*. ACM, 49–60.

[20] David Ungar. 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM Sigplan notices* 19, 5 (1984), 157–167.

[21] Tom White. 2012. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.".
[22] Lijie Xu, Tian Guo, Wensheng Dou, Wei Wang, and Jun Wei. 2019. An experimental evaluation of garbage collectors on big data applications. *Proceedings of the VLDB Endowment* 12, 5 (2019), 570–583.
[23] Yang Yu, Tianyang Lei, Weihua Zhang, Haibo Chen, and Binyu Zang. 2016. Performance analysis and optimization of full garbage collection in memory-hungry environments. *ACM SIGPLAN Notices* 51, 7 (2016), 123–130.
[24] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.
[25] Wenyu Zhao and Stephen M Blackburn. 2020. Deconstructing the garbage-first collector. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 15–29.