

# TeraHeap for G1

## Efficient Caching for latency-sensitive applications

*Maria Charalambous*

Thesis submitted in partial fulfillment of the requirements for the  
*Masters' of Science degree in Computer Science and Engineering*

University of Crete  
School of Sciences and Engineering  
Computer Science Department  
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Prof. *Polyvios Pratikakis*

---

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

The work has been supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).




UNIVERSITY OF CRETE  
COMPUTER SCIENCE DEPARTMENT

**Importing TeraHeap in G1 GC: Efficient Spark Caching for  
latency-sensitive applications**

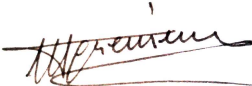
Thesis submitted by  
**Maria Charalambous**  
in partial fulfillment of the requirements for the  
Masters' of Science degree in Computer Science

THESIS APPROVAL


Author:

  
\_\_\_\_\_  
Maria Charalambous

Committee approvals:

  
\_\_\_\_\_  
Polyvios Pratikakis  
Associate Professor, Thesis Supervisor

  
\_\_\_\_\_  
Angelos Bilas  
Professor, Committee Member

  
\_\_\_\_\_  
Kostas Magoutis  
Associate Professor, Committee Member

Departmental approval:

  
\_\_\_\_\_  
Polyvios Pratikakis  
Associate Professor, Director of Graduate Studies

Heraklion, March 2023



## Abstract

Big data analytic frameworks like Apache Spark, handle the vast amount of data by moving objects outside the JVM managed heap (off-heap) onto a fast storage device. However, this strategy leads to high serialization/deserialization (S/D) costs and high garbage collection (GC) overhead, when off-heap objects are relocated back into the managed heap for processing. TeraHeap is a mechanism that manages to eliminate these overheads, by extending the JVM to use a second, high-capacity heap (H2) that is memory-mapped over a fast storage device and coexists alongside the regular heap (H1). TeraHeap eliminates the S/D cost with the use of memory-mapped I/O, and reduces the GC cost by avoiding GC scans over the secondary heap. TeraHeap achieves this by (1) marking candidate objects for placement in the H2 and indicating when to move them, (2) tracking live objects in the H1 that are referenced from H2, (3) reclaiming dead objects in H2. Originally TeraHeap was implemented in the Parallel Scavenge Collector, where large GC pauses are allowed because the main concern is the application's throughput. However, this does not perform well with real-time applications, due to its long pauses. Garbage-First (G1) Collector is for latency-sensitive applications, where the GC pauses are small and they meet a soft real-time goal with high probability while achieving high throughput.

In this thesis, we imported the TeraHeap mechanism in G1 GC. We aim to solve the off-heap problem of big data, in latency-sensitive applications that need quick responses without long GC pauses. Importing TeraHeap in G1 introduces unique challenges not encountered by Parallel Scavenge, highlighting the design differences between the two collectors. These challenges encompass (1) concurrent heap marking alongside the application threads, (2) G1's use of evacuation rather than compaction for small pauses during heap collection, and (3) the incremental collection approach applied to the old generation. Our evaluation shows that for the same DRAM size, TeraHeap improves performance by up to 72% compared to native Spark. However, there is still room for further work in refining this import process, given its demonstrated complexity and non-trivial nature.



## Περίληψη

Στα frameworks ανάλυσης μεγάλων όγκων δεδομένων, όπως το Apache Spark, χειρίζονται τον τεράστιο όγκο δεδομένων μετακινώντας αντικείμενα εκτός του διαχειριζόμενου σωρού JVM (off-heap) σε μια συσκευή γρήγορης αποθήκευσης. Ωστόσο, αυτή η στρατηγική οδηγεί σε υψηλά κόστη σειριοποίησης/αποσειριοποίησης (S/D) και συλλογής σκουπιδιών (GC), όταν τα αντικείμενα εκτός σωρού μεταφέρονται πίσω στον διαχειριζόμενο σωρό για επεξεργασία. Το TeraHeap είναι ένας μηχανισμός που καταφέρνει να εξαλείψει αυτά τα κόστη, επεκτείνοντας το JVM ώστε να χρησιμοποιεί ένα δεύτερο σωρό, υψηλής χωρητικότητας (H2) που είναι χαρτογραφημένη στη μνήμη μέσω μιας γρήγορης συσκευής αποθήκευσης και συνυπάρχει παράλληλα με τον κανονικό σωρό (H1). Το TeraHeap εξαλείφει το κόστος S/D με τη χρήση E/E με χαρτογράφηση μνήμης και μειώνει το κόστος GC, αποφεύγοντας τις σαρώσεις GC πάνω από τον δευτερεύοντα σωρό. Το TeraHeap το επιτυγχάνει αυτό (1) επισημαίνοντας υποψήφια αντικείμενα για τοποθέτηση στο H2 και υποδεικνύοντας πότε πρέπει να μετακινηθούν, (2) εντοπίζοντας ζωντανά αντικείμενα στο H1 που αναφέρονται από το H2, (3) ανακτώντας νεκρά αντικείμενα του H2 σωρού. Αρχικά, το TeraHeap υλοποιήθηκε στον Parallel Scavenge Collector, όπου επιτρέπονται μεγάλες παύσεις GC επειδή το κύριο μέλημα είναι η απόδοση της εφαρμογής. Ωστόσο, αυτό δεν αποδίδει καλά με εφαρμογές σε πραγματικού χρόνου, λόγω των μεγάλων παύσεων. Ο Garbage-First (G1) collector είναι για εφαρμογές ευαίσθητες στις καθυστερήσεις, όπου οι παύσεις GC είναι μικρές και προσπαθούν να κυμαίνονται κάτω από ένα όριο πραγματικού χρόνου, ενώ επιτυγχάνουν ταυτόχρονα υψηλή απόδοση.

Σε αυτή τη διατριβή, εισαγάγαμε τον μηχανισμό TeraHeap στο G1 GC. Στόχος μας είναι να λύσουμε το πρόβλημα του μεγάλου όγκου δεδομένων, σε εφαρμογές ευαίσθητες σε καθυστέρηση που χρειάζονται γρήγορες απαντήσεις χωρίς μεγάλες παύσεις GC. Η εισαγωγή του TeraHeap στο G1 εισάγει μοναδικές προκλήσεις που δεν αντιμετωπίστηκαν στον Parallel Scavenge, τονίζοντας τις σχεδιαστικές διαφορές μεταξύ των δύο συλλεκτών. Αυτές οι προκλήσεις περιλαμβάνουν (1) ταυτόχρονο μαρκάρισμα του σωρού concurrently με τα νήματα της εφαρμογής, (2) ο G1 χρησιμοποιεί τη τεχνική evacuation αντί του compaction για τις μικρές παύσεις συλλογής του σωρού και (3) η σταδιακή συλλογή που εφαρμόζεται στην παλιά γενιά. Η αξιολόγησή μας δείχνει ότι για το ίδιο μέγεθος DRAM, το TeraHeap βελτιώνει την απόδοση έως και 72% σε σύγκριση με το εγγενές Spark. Ωστόσο, υπάρχει ακόμη περιθώριο για περαιτέρω εργασία στην εισαγωγή του μηχανισμού αυτού στο G1, δεδομένου της πολυπλοκότητας και του μη τετριμμένου χαρακτήρα του.





## Acknowledgments

First and foremost, I extend my sincere appreciation to Prof. Polyvios Pratikakis, my supervisor, whose passion for his work has been a constant source of inspiration. I am always amazed by his knowledge and his high level of critical thinking. His guidance and unreserved supervision were invaluable in my academic pursuits. I am truly fortunate to have had the opportunity to learn and grow under his mentorship.

I would like to also express my gratitude to Foivos Zakkak for his insightful contributions and the generous amount of time that he dedicated during our meetings, earnestly striving to assist us. His commitment and expertise have significantly enriched our discussions, contributing to the depth and quality of our research endeavors.

To Iacovos Kolokasis, I owe a debt of gratitude for his invaluable assistance. Always there when I needed guidance, answering my questions with patience and clarity. His willingness to address every query, no matter how small, has been instrumental in my learning process. His support has helped me overcome obstacles, giving me strength during difficult times.

My heartfelt thanks go to my family: my mother Theodosia, my father Charis, and my sister Mikaella, for their unwavering faith in me and unconditional support. Their belief in my abilities has been the driving force behind my academic achievements, and without them, this journey would not have been possible.

Moreover, I would like to express my deepest appreciation to my friends Marianna and Aikaterini, whose companionship and support have been constant throughout this academic odyssey. Their presence has added warmth to the challenges and successes we've shared. Our joyous moments will always be remembered.

In a category of her own, my feline companion Maya deserves special mention. Through countless nights of studying, her comforting presence and gentle purring on my lap provided the energy and motivation needed to push forward.



*to my family*



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Background</b>	<b>5</b>
2.1	Apache Spark . . . . .	5
2.2	TeraHeap . . . . .	6
2.3	Generational Collectors . . . . .	8
2.4	Garbage-First Collector . . . . .	8
2.4.1	Heap Layout . . . . .	9
2.4.2	Collections . . . . .	9
2.4.3	Concurrent Marking . . . . .	10
2.4.4	Humongous objects . . . . .	11
2.4.5	Write Barriers . . . . .	12
2.5	Remember Sets . . . . .	12
<b>3</b>	<b>Design and Implementation</b>	<b>15</b>
3.1	Identifying and moving Candidate objects to H2 . . . . .	15
3.2	Reclamation of H2 Objects . . . . .	16
3.3	Tracking Backward References (H2 to H1) . . . . .	17
3.4	Remember Sets and Post-Write Barrier . . . . .	18
3.5	Concurrent Marking Cycle . . . . .	20
3.6	Evacuations . . . . .	23
<b>4</b>	<b>Evaluation</b>	<b>27</b>
4.1	Native Spark Configuration . . . . .	27
4.2	Spark Configuration for TeraHeap . . . . .	28
4.3	Tuning G1 GC . . . . .	28
4.4	Workloads and datasets . . . . .	30
4.5	Time breakdown . . . . .	31
4.6	Results . . . . .	32
<b>5</b>	<b>Related work</b>	<b>37</b>
5.1	Off-heap caching . . . . .	37
5.2	Region-based memory management . . . . .	38
5.3	Minimizing S/D overhead . . . . .	38

5.4	TeraHeap contribution . . . . .	39
<b>6</b>	<b>Future work</b>	<b>41</b>
6.1	Full GC . . . . .	41
6.2	Pause Time Estimation . . . . .	41
6.3	Remember Sets . . . . .	42
6.4	Asynchronous H2 transfers . . . . .	42
6.5	Multi-threaded Allocator . . . . .	43
<b>7</b>	<b>Conclusion</b>	<b>45</b>
	<b>Bibliography</b>	<b>47</b>

# List of Tables

4.1	Flags used for tuning G1 GC, to avoid full collections[1, 2]. . . . .	30
4.2	Configuration of each workload for native Spark and TeraHeap integrated Spark. . . . .	31





# List of Figures

2.1	(a) Native Spark Off-heap caching via S/D, (b) TeraHeap integrated Spark, on-heap caching over a memory-mapped fast storage device	7
2.2	A sample of a mixed collection.	10
2.3	G1 garbage collection cycle. The image was taken from this[3] article	11
2.4	Representation of G1 heap with humongous objects.	12
3.1	TeraHeap design overview in G1 GC. Both heaps are region-based and have their own card table	19
4.1	Performance of native and TeraHeap integrated Spark in the graph-based workloads	32
4.2	Performance of native and TeraHeap integrated Spark in the machine learning workloads	34
4.3	These are the findings presented in [4] for the TeraHeap implementation in the Parallel Scavenge Collector. They closely resemble the findings we presented for TeraHeap in G1 GC.	36



# Chapter 1

## Introduction

Big data analytic frameworks like Apache Spark [5], are specifically crafted for processing massive datasets. These frameworks face the challenge of processing and caching vast amounts of data that may exceed the capacity of the available heap [6]. To tackle the issues of growing datasets and limited DRAM capacity, a common approach is to move objects outside the managed heap to an off-heap storage device, like an NVMe SSD. On the other hand, this practice has some complexities as well. It introduces high serialization/deserialization (S/D) overhead when transferring data between the memory and the device. Because the data are in their serialized form when residing in the off-heap device, the frameworks can not compute directly over them. Therefore, these data need to be brought back to the managed heap for further processing. Moving such a large volume of objects back to the managed heap heightens the memory pressure. Many big data frameworks are written in Java, in which memory management is an automatic process that is managed by the Java Virtual Machine (JVM). In managed language environments, such as JVM, high memory pressure in the managed heap incurs excessive garbage collection (GC) overhead.

Moreover, lots of the objects in big data, stay alive throughout many commutation stages, displaying long lifetimes [7, 8, 9]. Therefore the significant number of long-lived objects places further memory pressure on the managed heap, leading to frequent garbage collections with low yield. Garbage collections are stop-the-world (STW) operations, where application threads are paused until the collection process is finished. Therefore, the frequency of these events can affect the application's performance and responsiveness.

TeraHeap [4, 10] is a system that eliminates S/D and GC overheads for a large portion of the data in managed big data analytics frameworks. TeraHeap avoids the need for S/D by keeping all cached data on-heap but off-memory, using memory-mapped I/O. To achieve this, it extends the JVM to use a second, high-capacity heap (H2) that resides on a memory-mapped fast storage device and is exclusively used for cached data. This secondary heap coexists alongside the regular heap (H1), while maintaining the illusion of a single unified JVM heap.

TeraHeap provides a hint-based interface that uses key-object opportunism [11] and enables frameworks to mark objects and indicate when to move them in H2. During GC, TeraHeap starts from root key-objects and dynamically identifies the objects to move to H2. It also reduces the GC overhead by avoiding costly GC scans over objects in H2. Because the GC scans in H2 are fenced, TeraHeap needs to address the following: (1) how to reclaim dead objects in H2 without GC scans, and (2) how to track backward references (H2 to H1) with low GC cost and I/O overhead.

TeraHeap was first implemented in the Parallel Scavenge Garbage Collector (GC). It has been shown that with the same DRAM size, TeraHeap improves performance by up to 73% compared to native Spark, and also it consumes up to  $4.6\times$  less DRAM by doing so. Implementing TeraHeap under the Parallel Scavenge GC, makes it well-suited for big data applications that do batch processing, due to its emphasis on achieving high throughput. In batch processing, large volumes of data are processed in parallel, where longer pause times during garbage collection are acceptable. Parallel Scavenge GC is designed to maximize overall system efficiency, by parallelizing the overall performance of batch-processing applications. Different GC algorithm addresses specific performance needs, providing flexibility for Java developers to choose the one that aligns with the demands of their applications.

In this work, we will import the TeraHeap mechanism to the Garbage First Garbage Collector (G1 GC) [12, 13]. G1 GC strikes a balance between throughput and low-latency requirements. This makes G1 GC particularly suitable for applications where minimizing pause times is a top priority. We aim for the benefits of the TeraHeap, to be passed on to the latency-sensitive application through the G1 GC. Compared with STW collectors, G1 allows users to specify a soft real-time goal for GC pauses and meets this goal with high probability. Catering to scenarios where low-latency performance is crucial, such as interactive or real-time applications, G1 GC is the preferred garbage collector.

Since Java 9 G1 is the default Garbage Collector, leading to widespread default usage across many applications. For instance, LinkedIn employs the G1 Collector, covering around 80% of its application landscape [14]. Presto, which is a distributed query engine that supports much of the SQL analytics workload at Facebook is using G1 GC as well [15]. Presto is known for its ability to perform fast interactive queries on large datasets. Moreover, Spark while collaborating with major enterprises, has stated that they frequently encounter concerns associated with garbage collection during the Spark execution [16, 17]. The choice of a GC algorithm can have dramatic effects on application performance and can even influence application implementation choices. Therefore they recommend G1 GC with some tuning for applications demanding real-time responsiveness.

In the initial implementation, TeraHeap was integrated with the Parallel Scavenge garbage collector. Importing TeraHeap into the G1 GC, however, presented a non-trivial challenge due to the inherent differences between G1 and Parallel Scavenge. G1 employs an evacuation technique in its collections, contrasting with the compaction technique utilized by Parallel Scavenge. A compaction must identify

all the live objects and their new locations, to proceed with the copying. In Parallel Scavenge this operation is single-threaded. On the other hand, evacuations are multi-threaded, and live objects can be copied to their new location as soon as they are identified. Additionally, G1 incorporates a Concurrent Marking for identifying live objects in the old generation, which traverse the heap concurrently with the application. Also, the space reclamation in the old generation is incremental. This stands in contrast to the batch-like approach of Parallel Scavenge. The adaptation of TeraHeap to G1 required addressing these distinct characteristics to ensure compatibility and effective performance within the G1 garbage collection framework. In this work, G1 was extended to do the following

- Identify the candidate objects to be moved in H2, while marking the managed heap concurrently with the application.
- Move objects to H2, while incrementally collecting the old generation
- Reclaim dead objects in H2 without any GC scans
- Tracks backward references (H2 to H1) to find the live objects in the managed heap (H1) and not reclaim them during a garbage collection

As the initial work of TeraHeap, we will also use the Spark framework which is extended to use the TeraHeap mechanism through its hint-based interface, without requiring modifications to the applications running on top of them. Our modification took place within the HotSpot Java Virtual Machine of OpenJDK 17 [18].

We implement an early prototype approach, for the TeraHeap importation in G1 GC. Our focus has been primarily on making the evacuation pauses and the Concurrent Marking to utilize the secondary heap (H2). While this marks an essential first step, there remains more work ahead to comprehensively address other aspects of the integration process. Our evaluation shows that for the same DRAM size, TeraHeap improves performance by up to 72% compared to native Spark.



## Chapter 2

# Background

This section provides some background related to Apache Spark, TeraHeap, and the G1 Garbage Collector.

### 2.1 Apache Spark

Serialization in Spark [5] is the process of converting a Java object, which resides in the computer's memory (managed heap), into a format that is suitable for storing it in memory or to an off-heap device. It also allows for the data to be transferred through the network and shared across multiple JVMs. The serialization transforms the object into a compact sequence of bytes. On the other hand, deserialization is the reverse process, where the byte stream is used to reconstruct the Java object. Serialization is useful for scenarios where you need to persistently save the state of an object or transmit it efficiently across the network. However, it comes with the trade-off of introducing additional complexity and overhead during execution.

Spark represents a large collection of data as a Resilient Distributed Dataset (RDD) [19]. RDDs are divided into smaller chunks called partitions. Each partition can be processed independently on different machines within a cluster. This division allows Spark to perform computations in a distributed and parallel fashion, making the processing more efficient and scalable.

The driver is the main program that defines the Spark application, while the executors are the worker nodes that carry out the tasks of the application in a distributed fashion. The Spark driver specifies a sequence of transformations and actions on RDDs and these tasks are submitted to executors. Each transformation generates a new RDD based on the existing one.

RDDs are lineage-based, meaning they keep track of the sequence of transformations applied to their base data. Spark operates in a lazy evaluation mode, where transformations on RDDs are not immediately executed. Therefore when you perform an action on an RDD, Spark needs to compute the data associated with that RDD based on its lineage, and those intermediate results need to be

recomputed each time. This also allows Spark to recompute lost data in case of node failures.

The recomputation can be avoided by caching or persisting intermediate RDDs. This is done through the use of *persist()* and *cache()* operations. Those intermediate results can be stored in memory (on-heap) or storage (off-heap), in a serialized or deserialized form.

The *cache()* operation means that an RDD is stored on-heap in a deserialized form. The *persist()* operation stores the RDD in a user-defined storage level. Spark's storage levels [20] are meant to provide different trade-offs between memory usage (how much memory out of the JVM heap is used) and CPU efficiency (the added cost of the serialization-deserialization process). The following are some storage levels that Spark supports for persisted RDDs.

**MEMORY\_ONLY** Stores RDDs as deserialized Java objects in the JVM heap. That is the most CPU-efficient option.

**MEMORY\_AND\_DISK** RDDs' partitions that fit in memory are stored in their deserialized form in the JVM heap. Partitions that do not fit in memory are serialized and stored on disk. Old partitions can be evicted from memory based on the LRU algorithm, to reclaim some space on the JVM heap for new partitions to accommodate it. This is a good compromise between memory and CPU efficiency.

**DISK\_ONLY** RDDs are stored only on disk in their serialized form. This is the most memory-efficient storage level, but it is also the slowest.

## 2.2 TeraHeap

Big data analytics frameworks process and cache massive data volumes that may exceed the managed heap. Therefore, frameworks temporarily move long-lived objects outside the managed heap (off-heap) on a fast storage device. However, this approach faces two challenges. It increases the serialization/ deserialization (S/D) cost and it heightened the memory pressure when off-heap objects are moved back to the managed heap for processing. Moving a large volume of off-heap objects back to the managed heap also increases the GC cost.

TeraHeap is a mechanism that eliminates these problems. It eliminates the S/D overhead, by extending the JVM to use a second high-capacity heap (H2) that is memory-mapped over a fast storage device [21, 22] and coexists alongside the regular heap (H1). Figure 2.1 displays the differences between the common approaches and TeraHeap. Common approaches cannot avoid the S/D cost when attempting to move objects off-heap. However, TeraHeap uses the Memory-mapped I/O to craft the illusion of a single unified heap within the JVM, allowing the utilization of the MEMORY\_ONLY storage level. This enables the objects to be stored in their deserialized form in H2. Therefore Spark engages in on-heap caching, where



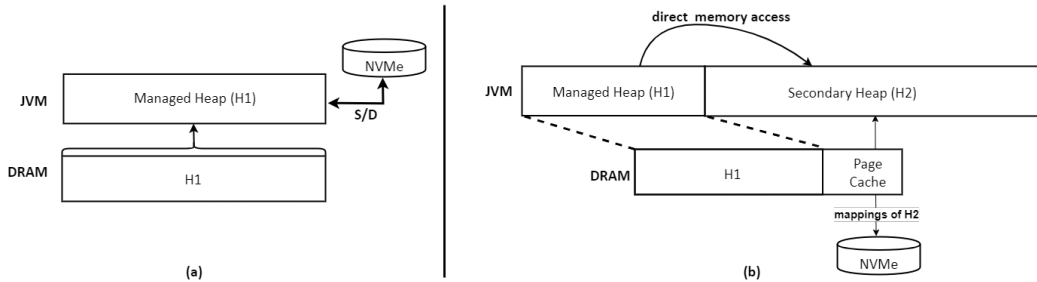


Figure 2.1: (a) Native Spark Off-heap caching via S/D, (b) TeraHeap integrated Spark, on-heap caching over a memory-mapped fast storage device

TeraHeap strategically moves objects off-DRAM over a fast storage device. This allows for the JVM to directly access the deserialized objects in H2 without the added cost of S/D.

Moreover, TeraHeap during garbage collection, tries to move all persisted objects in H2, and once they are moved they are never brought back to the managed heap, as direct access from the device is feasible. This places less strain on H1, minimizing the memory pressure of the managed heap and as a natural consequence, it contributes to a less frequent garbage collection. The GC cost is further reduced by fencing the garbage collector from scanning the H2 heap to avoid excessive I/O. As shown in the Figure 2.1, with the same DRAM size, TeraHeap consumes less DRAM for the JVM-managed heap compared to the common approach. This efficiency stems from TeraHeap’s ability to minimize the memory pressure on the managed heap. TeraHeap utilizes the remaining DRAM for page cache purposes.

Frameworks use TeraHeap through its hint-based interface based on key-object opportunism [11]. It enables the frameworks to mark candidate objects to populate H2 and indicates when to move them off-DRAM. The hint-based interface works at the framework level and is transparent to the applications written on top of such frameworks.

TeraHeap addresses three main challenges, that we took into account during its integration into G1 GC.

- **Identifying and moving candidate objects to H2:** it starts from root key-objects that are candidate objects to populate H2 and dynamically identifies their transitive closure. Afterward, it indicates when to move them off the managed heap.
- **Reclaim H2 dead objects:** During a garbage collection (GC) the heap object graph is traversed. The GC scans over H2 are fenced, to avoid expensive device I/O. Therefore, TeraHeap reclaims dead objects in H2 without scanning over it during a GC.
- **Tracking backward references (H2 to H1):** Fencing GC scans in H2, requires TeraHeap to further track down all the backward references (H2 to

H1) to prevent the garbage collector from reclaiming live objects in H1 that are referenced from live objects in H2.

Chapter 3 discusses the methodology employed to tackle these challenges, considering the distinctive nature of G1 GC.

## 2.3 Generational Collectors

Garbage collectors in JVM have the rationale that most objects are short-lived, meaning that they become unreachable soon after their creation. For this reason, the managed heap is divided into two areas, the young generation and the old generation. Based on the object's age, it may reside in the young space if it's relatively new, or in the old space if it is mature enough and it's considered to be a long-lived object. After every garbage collection, the age of an object is increased.

The young generation is further divided into Eden and Survivor spaces. Eden space holds all the new allocations that have been made by the mutator threads. When Eden fills up, a garbage collection is invoked and the objects that were in Eden space, are moved into the Survivor space. An object stays in the Survivor space until it becomes of age, and by each collection it survives it's again reallocated in the Survivor space. When it reaches a certain aging threshold, it is promoted to the old generation.

Based on the assumption that most objects die young, the young generation is collected more frequently, because it is assumed to have more garbage. Thus there are different types of collections, the ones that reclaim space only in the young generation, and the ones that reclaim space from the young and the old as well, but they are not as frequent.

## 2.4 Garbage-First Collector

Garbage-First (G1) Collector is a generational collector. G1 GC aims to strike a balance between latency and throughput. It attempts to achieve high throughput by meeting a pause time goal during its collections, with high probability [12]. All collections are stop-the-world (STW) operations, meaning the application threads are halted until the whole operation is completed. An objective of G1 is that it should rarely need to do a full collection over the entire heap, which is a slow and expensive STW operation. Alternatively, it does not collect the old generation in a batch, but incrementally with small collections that meet the pause time goal. But if these collections do not reclaim the old space fast enough a fallback to a full GC will occur. Therefore G1 tries to minimize the duration of the STW pauses, and this makes it ideal for latency-sensitive applications. When a fast response from the application is expected, we want the garbage collection interference to take as little time as possible. Thus we want to import TeraHeap to G1, for the

big data applications that are latency-sensitive to benefit from TeraHeaps' dual heap management.

### 2.4.1 Heap Layout

Generational collectors split the heap into 3 big sections: Eden, Survivor, and Old spaces. G1 differs from that by partitioning the heap into a set of equal-sized heap regions, each a contiguous range of virtual memory [23]. Their size must be a power of two (between 1MB - 32 MB) and the goal is to have no more than 2048 regions [24]. Each region is assigned to a space, therefore spaces are considered as logical sets of regions. The percentage of heap allocation for each space is not fixed, it can be dynamically adjusted after each collection based on the performance of the previous collection.

If an object is bigger than the G1 region size, then it is considered humongous and is allocated in a continuous set of regions.

### 2.4.2 Collections

With each collection, JVM tries to free as much space as possible, while ensuring that there are enough free regions. There are three main collections in G1 GC

- Young collections, where only the young regions are collected. These are the most frequent ones, as G1 is based on the hypothesis that most objects die young.
- Mixed collections, where all the young regions are collected along with a few candidate old regions. They collect the old generation incrementally while following the “most garbage first” principle, hence the name Garbage-First. They collect a part of the old generation, where the most amount of garbage lies.
- Full collection, in which the entire heap is collected, i.e. both young and old generations.

Young and mixed collections are trying to meet a soft real-time goal, by following the evacuation technique [12, 25] to reclaim garbage. For the evacuation pauses, G1 has the concept of a collection set (CSet) [12, 3], which is a list of regions that are being scheduled for collection. During an evacuation, a collection set (CSet) of regions is chosen and it should be able to be collected within the pause time limit. Figure 2.2 shows an example of a young evacuation. The circled regions are the ones included in the CSet. The live data inside the CSet are copied into a new set of regions, either survivor or old, based on each object's age. At the end of the evacuation, the CSet regions are freed and garbage is reclaimed. An evacuation is triggered when the Eden space fills up, and no room is left for new allocations.

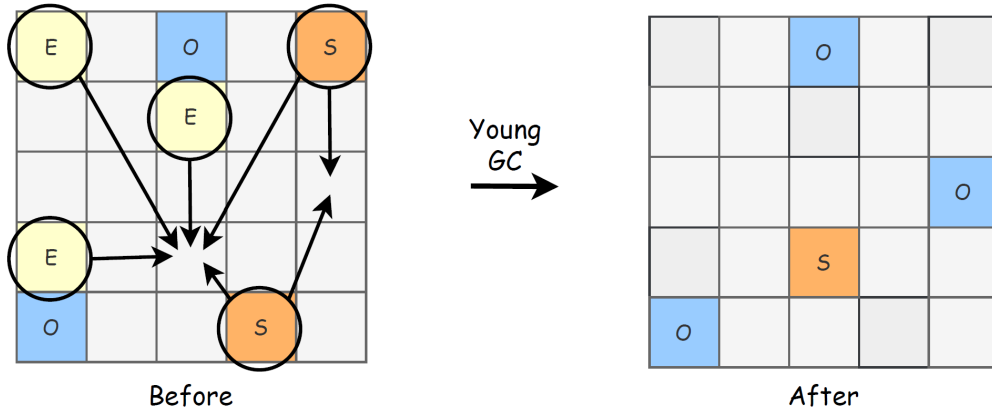


Figure 2.2: A sample of a mixed collection.

On the other hand, Full collections follow the compaction technique, where all the live objects on the heap are marked-and-swept together [26]. There is no need for a collection set here, nor the soft real-time goal is met. This is a costly GC and it is triggered only when there is too much fragmentation in the old generation, and there is not enough space to promote objects. All collections are parallel and STW operations. In this work, we focus on making the evacuation mechanism aware of the secondary heap. Consequently, we control all experimental runs so that no Full GC is triggered (section 4.2). Porting the Full GC phase to TeraHeap is orthogonal to this work.

### 2.4.3 Concurrent Marking

Upon start-up, G1 performs only Young collections and as time goes by, more and more objects are promoted to the old generation. When the old generation occupancy reaches a certain threshold [3], G1 schedules a Concurrent marking cycle. This cycle happens concurrently with the application and tries to determine the live objects in the old generation. We extended it so that during the old generation liveness analysis [24], it would also find the transitive closures of the objects that are meant to be moved in H2. Thus it does not only mark objects as live, but also flags them for potential relocation to the secondary heap. When the concurrent marking cycle completes, G1 knows how much live data and how many candidate H2 data each old region has. Then G1, instead of performing young collections, starts to perform mixed collections [3]. Mixed collections collect a part of the old generation incrementally, based on the “most garbage first” principle, which uses the information from the liveness analysis conducted during the Concurrent Marking. When a sufficient number of old regions are collected, G1 reverts to performing Young collections again. How many mixed collections are performed can be tuned by numerous flags and thresholds (section 4.2). The transitive closures of the root key-objects are found during the Concurrent Marking, and the subsequent

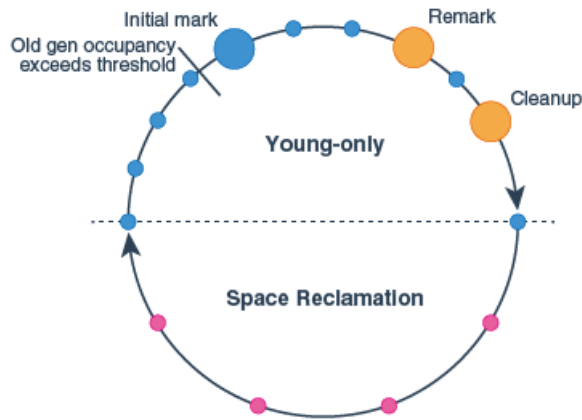


Figure 2.3: G1 garbage collection cycle. The image was taken from this[3] article

mixed collections transfer the tagged objects from H1, to H2 incrementally.

In later sections, we will see that the Concurrent Marking Cycle consists of five different phases, three of which are STW. These are the Initial Mark, Remark and CleanUp phases. Figure 2.3 displays the G1 garbage collection cycle. During the Young-only phase, only young collections can be initiated. They collect the young generation and promote objects in the old generation. When the old generation occupancy reaches a certain threshold the Concurrent Marking begins. In between the concurrent phases of the Marking young collections can occur. When the Marking has been completed, a last young collection is initiated and then the reclamation phase begins. In this phase, multiple mixed collections are incrementally reclaiming space in the old generation. When G1 determines that it has reclaimed enough old space, and further reclamation would be inefficient, it stops the mixed collections. Then the cycle restarts again with the young-only phase. Full collections can interrupt this cycle at any time, and restarting it again. It's worth noting that the Concurrent Marking can only be triggered during the Young-only phase.

#### 2.4.4 Humongous objects

Objects that are more than half a region's size, are considered to be humongous. A humongous allocation represents only one single-large object, and therefore it must be allocated into a contiguous set of regions [27]. This can cause significant fragmentation. Humongous objects are allocated directly to the old generation and are handled differently. Those objects are never included in the CSet of an evacuation pause, because they are too expensive to evacuate (they may exit the pause time goal). Therefore we may include them in the transitive closure of an object, but they are never actually moved to H2 during the evacuations. A humongous allocation can potentially trigger a Concurrent Marking cycle or a Full collection. Figure 2.4 shows a representation of the heap, with humongous

objects.

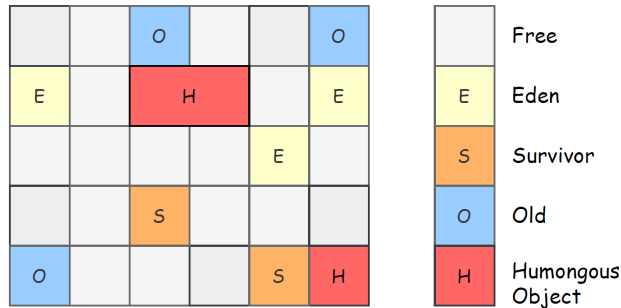


Figure 2.4: Representation of G1 heap with humongous objects.

### 2.4.5 Write Barriers

A write barrier is a code block that is emitted into the program by the compiler when a store operation occurs [28, 29]. It’s a mechanism that handles modification to the object graph and ensures that generational invariants are maintained. G1 has two write barriers; a pre-write barrier, and a post-write barrier, where the external code is inserted immediately before, or after the store operation respectively [12]. As we progress through this work, in later sections we will explore their necessity in garbage collection and how TeraHeap requires a more elaborate and expensive post-write barrier.

## 2.5 Remember Sets

Because G1 employs a region-based approach, during garbage collection it is crucial to track cross-region references. For that reason, G1 GC has independent Remembered Sets (RSets) per region, to track incoming references from other regions. Therefore only the roots and the region’s RSet must be scanned for references into that region, instead of the whole heap which would be inefficient and increase pause times.

To maintain RSets, G1 uses a post-write barrier [12, 30, 29] to record changes in the heap. The card table divides each region into smaller units called cards. When objects are modified, the corresponding cards are marked in the card table, through the post-write barrier.

The marked cards are later scanned to identify outgoing references and update the RSets of the corresponding regions. The processing of these cards, along with RSet updates, occurs concurrently with the application by refinement threads [31, 32, 30, 29] and at the beginning of each collection.

References originating from H2 are excluded from the RSets. To identify the backward references (H2 to H1) we use a different approach (section 3.2), which

requires a more elaborate and expensive post-write barrier.





## Chapter 3

# Design and Implementation

TeraHeap manages to eliminate the S/D cost with the use of memory-mapped I/O, and the GC cost is eliminated by fencing the garbage collection scans. In this chapter, we will discuss the three main challenges of the TeraHeap mechanism [4], which are (a) identifying and moving candidate objects to H2, (b) reclaim H2 dead objects, and (c) tracking backward references (H2 to H1). Importing TeraHeap in G1 GC indicates that the evacuation pauses and the Concurrent Marking traversals should be fenced from scanning H2. We will also discuss the methodology employed to tackle these challenges, considering the distinctive nature of G1 GC.

### 3.1 Identifying and moving Candidate objects to H2

Big data frameworks move specific objects outside the managed heap on an off-heap fast storage device. More specifically, Spark moves off-heap intermediate results (RDDs) organized into partitions. A partition is a group of objects with a single-entry root reference [33]. Its objects are considered to be long-lived and they are reused across computational stages and have similar lifetimes. Each time such an object needs to be reused, the whole partition from the off-heap device is deserialized and brought back into the managed heap for further processing.

TeraHeap provides a simple hint-based interface, that uses key-object opportunism [11] and enables Spark to tag those objects to be moved in H2. As the JVM has the illusion of a single unified heap, H2 is considered an on-heap caching that is mapped over a fast storage device. This allows for immediate access to deserialized objects without the need for S/D. Therefore objects that are moved in H2, are never relocated back into the managed heap (H1), which minimizes the memory pressure of H1. Consequentially, the frequency of the garbage collections on the managed heap is reduced.

Spark users explicitly annotate objects that need to be moved off-heap with the *persist()* call. Using JNI calls during the *persist()* operation, TeraHeap tags the object as a candidate to be moved in H2 [4]. This tag is essentially an extra word in the Java object header, which tells the JVM that this object is a candidate

for placement in the H2. This is the already existing mechanism of TeraHeap.

We extended the Marking process, to find the transitive closures of the objects that were tagged for placement in H2, through the *persist()* call. During the Concurrent Marking, the object graph is traversed. If it encounters a root key-object selected for H2 placement, it tries to identify the objects in its transitive closure and tag them for relocation in H2 as well. After the concurrent marking, a series of mixed collections are to be followed. The tagged objects that were found throughout the concurrent marking, are moved from H1 to H2 during the incremental mixed GCs.

## 3.2 Reclamation of H2 Objects

Data-intensive processing frameworks like Spark, often employ direct I/O. The data are fetched directly from the off-heap device into the managed heap. Since the page cache is bypassed, this can lead to a reduction in page faults associated with cache misses. TeraHeap on the other hand, has extended the JVM to use a secondary heap (H2) over a fast storage device via memory-mapped I/O (mmio). This can include I/O wait due to page faults when accessing the H2 backing device.

Therefore H2 is never garbage collected during a GC, to avoid traversing the H2 heap as a whole. If that were to happen, due to the nature of the graph traversal, which often may suffer from random accesses and poor page locality, then this may trigger a page fault for every reference in the graph that it would be followed. Moreover, the liveness analysis and the compaction of H2 will incur high I/O traffic due to excessive read and write operations.

For these problems to be avoided, TeraHeap reduces the GC cost by fencing the garbage collector from scanning H2 objects. Upon encountering a reference from H1 to H2, the collector is fenced from crossing into H2. Spark creates objects that can be grouped in sets of similar and long lifetimes. TeraHeap leverages this, to free pace in H2 without the need for GC scans. Space reclamation in H2 is achieved by organizing H2 in virtual memory as a region-based heap. Each H2 region hosts object groups with similar lifetimes, which enabled us to reclaim whole H2-regions and their objects in bulk.

Thus TeraHeap must ensure that while reclaiming an H2 region, none of the objects in that region is referenced from live objects in H1 or H2. To find such H2-regions, TeraHeap should tracks cross-region (H2 to H2) and forwarding (H1 to H2) references without scanning the H2.

For this purpose, TeraHeap holds some metadata per H2-region: the dependency list, and the live bit [4]. This functionality is integrated into the original TeraHeap mechanism, which is encapsulated in the 'allocator'. The 'allocator' is responsible for the management of the H2 heap. To us, the 'allocator' operates as a black box, and our interaction with it is solely through its interface functions when importing it into G1 GC. We used these metadata and imported them as follows:

### Dependency list

- The dependency list keeps track of the cross-region references in H2. Each H2 region has its own dependency list, which holds all the H2 regions referenced by its outgoing references.
- Newly transferred objects in H2, may also have cross-region references. After their evacuation in H2, their once forwarding references now become cross-region references. Therefore during their evacuation, we also check if they have references in existing H2 regions to update their dependency list.
- During runtime a reference in H2 may get updated. This will be logged through the extended post-write barrier (section 3.2) and during the next GC, if there is a new cross-region reference found the dependency list will be updated.

### Live bit

- Each H2 region has its own life bit, which indicates its readability from other objects.
- The garbage collector clears all the live bits at the beginning of the Concurrent Marking.
- During the Concurrent Marking heap traversal, for every forward reference found, the live bit of the referenced H2 region is set. The dependency list of this H2 region is traversed and for all the H2 cross-regions in the list, the live bit is set as well.
- At the end of the Concurrent Marking, any H2 region not marked as live, means that it is not reachable from any objects in H1 or H2. Therefore at the end of the marking, these H2 regions are reclaimed in bulk.

## 3.3 Tracking Backward References (H2 to H1)

In each garbage collection, the live objects of the collected area (CSet) are found, and the rest are reclaimed. Fencing GC scans in H2, requires further tracking down the references from H2 to H1 in order to identify all live objects within the collection area. To accomplish this, we implemented an extended card table over H2, that is optimized for fast storage devices. It's a byte array located in DRAM, where each entry corresponds to a fixed-size H2 card segment (similar to vanilla JVM). At the beginning of each collection, specific card segments are scanned, to identify all the live objects in the CSet that are referenced from H2.

Each card table entry can take one of the following values:

- **clean:** no backward references, the card segment is either empty or contains only cross-region references.
- **dirty:** there was a pointer update inside this segment by a mutator thread.
- **oldGen:** there are references only to the old generation.
- **youngGen:** there are references to the young generation, and there may or may not be some to the old as well.

To track the liveness coming from H2 during a normal young GC, we scan the cards with values of dirty and youngGen. For a mixed GC, we also scan the oldGen cards as well. Moreover, the first phase of the Concurrent Marking is piggybacked on a Young GC which also does an initial marking (section 3.4). During this initial marking, all the old objects that are directly reachable from the roots should be marked as live. Therefore during the start of a Concurrent Marking, we need to scan all the backward references; youngGen and dirty to find the live objects that need to be evaluated during the young collection, and the oldGen to find the old objects that need to be marked as live. During the evacuations, we adjust the backward references to point to the new locations of the evacuated H1 objects.

To keep the H2 card table updated, it requires a more extended post-write barrier. When a mutator thread makes a pointer store in the secondary heap, the post-write barrier will be triggered and the corresponding H2 card entry will be labeled as dirty. Later on, during the next garbage collection, all the H2 dirty cards are scanned and their value is updated. If all the backward references inside a card segment exclusively reference to objects in the old generation, then the card value is set to oldGen. If there is at least one backward reference in the young generation, then the card is set to youngGen. If no backward references are found, the card value will be set to clean. Also if a new cross-region reference is found during the dirty card scanning, the dependency list of the contained H2 region will be updated. Scanning H2 card table is multithreaded, therefore the GC threads can scan its card segments in parallel.

### 3.4 Remember Sets and Post-Write Barrier

G1 during an evacuation pause, should find all live objects in the CSet. The CSet can contain young regions and maybe some old ones as well. To identify the live objects in the CSet, G1 must firstly find the objects that are directly reachable from the JVM roots and the old generation, thereafter everything reachable from those is found during the evacuation process. Instead of scanning the whole heap, G1 has the concept of individual Remember Sets (RSet) [12]. Each region possesses its own RSet, encompassing potential locations in the old generation, that might contain references within that specific region. Essentially, the RSet stores incoming references from the old generation. Therefore, G1 in the root set of an evacuation, includes the JVM roots and the RSets of the regions that form the CSet.

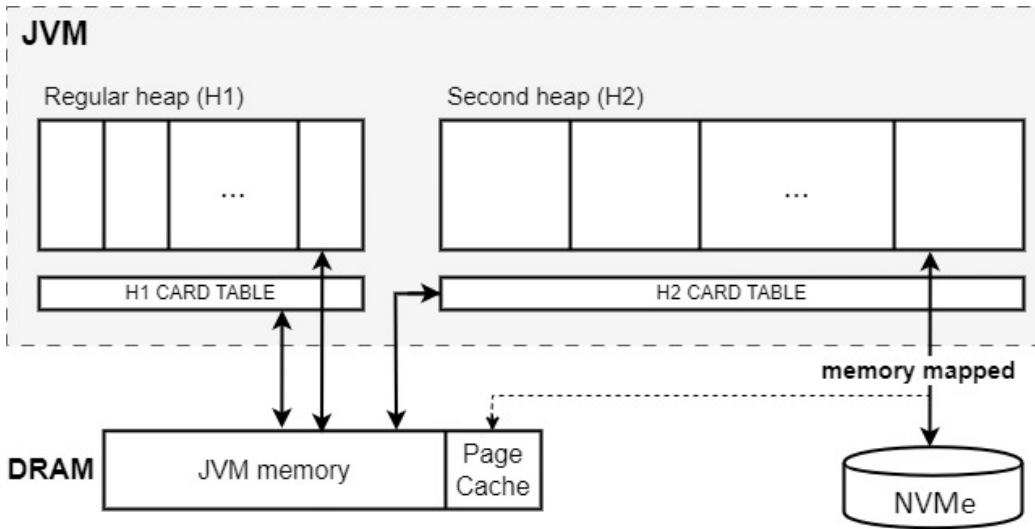


Figure 3.1: TeraHeap design overview in G1 GC. Both heaps are region-based and have their own card table

References originating from H2 are excluded from the RSets. To identify live objects within the CSet that are referenced from H2, the H2 card table is employed. As a result, the final root set also encompasses the H2 card segments that require scanning for this particular type of collection.

To maintain RSets, G1 uses a post-write barrier [12, 30, 29] to record changes in the heap, along with the use of a card table over the managed heap (H1). G1 partitions the heap into equal-sized areas, referred to as card segments, which are smaller than the regions. The card table is a byte array, where each of its card entries corresponds to an H1 card segment [31, 32]. When an application thread performs a store within an H1 card segment, the post-write barrier marks its card entry as dirty. A dirty card entry indicates that the associated card segment needs scanning to identify outgoing references and update the RSets of the regions it points to. The processing of these dirty cards, along with RSet updates, occurs concurrently with the application by refinement threads [31, 32, 30, 29] and at the beginning of each collection.

Importing TeraHeap to G1, requires a more extended post-write barrier to monitor updates in both heaps. These updates may arise from interpreted or JIT-compiled methods using the C1 and C2 JVM compilers. To discern the heap (H1 or H2) in which the pointer store occurs, the JVM must determine the corresponding card table and mark the relevant card as dirty. To achieve this, an additional range check has been introduced in the post-write barrier, enabling the selection of the appropriate card table. Consequently, the post-write barriers in both the template-based interpreter and JIT compilers have been extended to generate assembly code that incorporates these essential checks.

Figure 3.1 shows the design of TeraHeap implementation in G1 GC. JVM is

extended to use a second, high-capacity managed heap (H2) over a fast storage device that coexists with the regular managed heap (H1). H1 resides within the DRAM, whereas H2 is memory-mapped over a fast storage device, such as NVMe. Both heaps are region-based and they use independent card tables that are located in DRAM. The post-write barrier was extended to choose between these two card tables and update the appropriate one, based on which heap the store operation has been made.

### 3.5 Concurrent Marking Cycle

In every Young collection, the old generation's occupancy is constantly calculated based on the promoted objects. When the occupancy reaches a certain threshold, called IHOP (Initiating Heap Occupancy Percent) [3], a Concurrent Marking is triggered at the end of the young GC.

The marking cycle traverses the whole heap and marks all the live objects in the old generation, for the subsequent mixed collections to reclaim them incrementally. We extended it to find root key-objects that were tagged through the TeraHeap interface (via the *persist* operation) and calculate their transitive closure. This tag, is called *tera flag* and it's an extra word in the Java object header. When the *tera flag* of an object is enabled, then it's a candidate for placement in H2. This *tera flag* lets the following mixed collections know to evacuate this object in the secondary heap, instead of H1.

For the Concurrent Marking Cycle, G1 follows the principle of the snapshot-at-the-beggining (SATB) [12, 31]. It's like taking a snapshot of the heap at the start of the marking cycle. Therefore The objects identified as garbage are only the ones that were dead when the initial snapshot was taken. Moreover, all the newly allocated objects that appear during the marking, are considered by default to be live. Therefore, the set of live objects found during the Concurrent Marking is composed of the live objects in the snapshot, and the objects that were allocated after the snapshot was taken.

SATB algorithms need a pre-write barrier that is triggered before each pointer store, to record and mark objects that are part of the logical snapshot. When the barrier is triggered, the value of the pointer field is stored in a SATB buffer, before it is overwritten. These old values are considered to be live and they are traversed, in order to maintain the heap image that is in the snapshot taken at the beginning.

At the outside of the SATB marking, a snapshot of the heap is taken. Then the JVM roots and the young generation in the snapshot are originally scanned. This scan aims to identify and mark directly reachable objects in the old generation. Subsequently, following these identified live objects, the object graph of the old generation in the snapshot is traversed, to mark the remaining live objects. To accomplish this task, the SATB marking is broken down into five phases, some of which are STW and others are concurrent.

- **Initial mark phase (STW):** This phase is piggybacked on a Young GC. While the root set is scanned, G1 additionally marks as live all the root reachable objects in the old generation.
- **Root region scanning phase (Concurrent):** Root regions are the regions that the prior young GC just evacuated into. These can be old or survivor regions. The root regions are scanned for direct references to the old generation and mark them as live. This phase must be completed before the next Young GC can start. After this phase, all the old objects that are directly referenced from the young generation and the roots, have been marked as live.
- **Concurrent marking phase (Concurrent):** Traverse the old generation concurrently with the application threads, to find and mark all live objects. This is done with the use of a tricoloring algorithm [34]. This phase can be interrupted by young collections.
- **Remark phase (STW):** Drain the remaining SATB buffers to trace and mark their objects if they are unvisited.
- **Cleanup phase (STW):** At the end of the marking, G1 knows how much live data each old region has. In this phase, all the old regions that are full of garbage are freed. For the rest old regions, an estimated 'price' for their evacuation is computed. This metric is called gc-efficiency and is based on the reclaimable space the old region has, over the estimated time it needs to be reclaimed. The efficient old regions are included in the candidate region set. This set holds the old regions that are candidates for mixed collection. G1 sorts these regions, cheapest first, resulting in a ranking of old regions that indicate their desirability to be included in a mixed collection.

The intuition is that, at the end of the Concurrent Marking, we would have a candidate-set of old regions, sorted based on their efficiency. From this candidate-set, mixed collections will choose each time the most efficient old regions to include in their collection set (CSet). Every mixed GC that would follow, will include as many old regions, as the pause time allows. The mixed collections are stopped when the space reclamation of the old generation is determined to be costly and inefficient [24], or when the candidate-set is empty. It's worth noting, that only the old regions that existed at the time of the snapshot can be included in the candidate-set for the incremental collections.

To import TeraHeap we need to also compute the H2 candidate data that each old region has and account it to their gc-efficiency computation. Thus we will have three statistics for each old region: (a) liveness ratio, which is the total live data in the region, (b) h2-liveness ratio, which is a subset of the liveness ratio and counts for the amount of live data that are tagged to be moved in H2, (c) and the gc-efficiency.

The concept is that regions with more H2 live data will have a higher possibility of being chosen for collection [12]. Leading to a more relieved managed heap where its memory pressure won't cause frequent garbage collections. Thus we account the H2 live data as reclaimable space and try to get them higher in the sorting ranking for them to be chosen for collection. To achieve this we extended all the Concurrent Marking phases to (a) enable the `tera` flag of the objects that are included in the transitive closure, (b) increase the h2-liveness ratio of an old region if there is such an object in it, and (c) fence the marking upon finding a forwarding reference (H1 to H2), as to not traverse the H2 heap.

The first phase of the marking is also extended to clear the H2 regions' live bit. Throughout the whole marking, upon encountering a forwarding reference, we set the live bit of the H2 region, and the live bit of its H2 cross-regions found through its dependency list. During the final phase, H2 regions that don't have their live bit set (they are full of garbage) are reclaimed. Lastly the gc-efficiency is evaluated for every old region and the candidate region set is built. Originally G1, used the following equation to calculate the gc-efficiency for every region:

$$\text{Reclaimable Bytes} / (\text{RSet scan time} + \text{Copy time})$$

The reclaimable space, is the amount of garbage and the regions' unallocated space that is left unused. The denominator represents the time required for the region to be reclaimed, and it is comprised of the estimated time it needs to (1) scan the region's RSet, and (2) copy the region's live data into another region. Note that a region may have a small amount of live data, yet still have low estimated efficiency because of a large remembered set. G1 then sorts these regions based on their estimated efficiency in descending order. The cost of collecting an old region may change over time so this initial sorting is considered approximate.

We alter the gc-efficiency equation as follows:

$$(\text{Reclaimable bytes} + \text{H2 bytes}) / (\text{RSet scan time} + \text{Copy to H1 time})$$

As previously mentioned, H2 data (h2 liveness ratio) is accounted as reclaimable space, and their copy time to H2 is not included in the calculation. We only estimated the time it needs to evacuate objects into H1. This choice was motivated by the strategic objective of prioritizing the ranking of the regions that contain more H2 data. This amortization decision was taken based on the fact that the H2 data are only copied once and they are never moved from H2. Thus minimizing the impact on the overall efficiency.

Furthermore, it is important to mention that not the whole transitive closure of a root key-object is found, only a subset of it. That's because only the old generation in the snapshot is traversed. Any subtree of the object graph extending back into the young generation or to newly allocated objects, those remain untraversed.

In more detail, the transitive closure of an object starts to form when the root key-object is found. We know if an object should be tagged as an H2 candidate,



only if its parent object was tagged as well. Therefore if we lose track of the parent object, we also lose track of its whole subtree that should be included in the closure. G1 only tries to mark the old generation. For that reason alone, it does not follow the old to young references. Moreover, the newly allocated ones are not traversed, as they are not included in the snapshot and therefore considered live by default. Taking into account the preceding statements, we don't include in the transitive closure the objects' subtrees that have been spawned into the young generation or the newly allocated ones, because G1 does not traverse those areas. Therefore we lose track of these subtrees.

As far as the SATB is concerned, we did not extend the pre-write barrier because there was no need to. The processing of the SATB buffers and the concurrent marking happen interchangeably. Through the processing of a SATB buffer, all the overwritten values that it holds, are marked as live and traversed, as to maintain the initial logical snapshot. Those objects may already be visited by the Concurrent Marking traversal.

Thus if an object is in the SATB buffer and waits for processing, it may already be marked (1) either before its insertion in the buffer (2) or after its insertion through another objects referencing to it. In both cases, it will also have its *tera* flag enabled if needed because it would be traversed through its parent object. Otherwise, the object was not visited before, therefore it will be traversed earlier than its parent, during the SATB processing. Therefore we do not know if it should be included in any transitive closure to enable its *tera* flag, because we don't know yet this information about its parent either. Later on, when its parent will finally be traversed through the Concurrent Marking, G1 will see that this subtree is already visited and will not traverse it again. In consequence, an object's subtree may also be excluded from the transitive closure if the object has been modified and processed through a SATB buffer.

### 3.6 Evacuations

When Eden space fills up, an allocation failure will occur and an evacuation will be performed [12]. An evacuation is a garbage collection technique, where a set of regions (CSet) is chosen to be collected [12]. Those regions should be able to be collected within the pause time goal. G1 has two types of evacuations, the young and mixed collections. We extended those pauses to be aware of the secondary heap, and refrain them from reclaiming H1 objects that are still referenced by live H2 objects. Mixed collections were also extended to transfer H2-tagged objects off-DRAM into H2, while incrementally collecting the old generation. Evacuations are multi-threaded. As soon as a live object is identified, (1) it is copied to its new location, (2) the new address is communicated to the object referencing it, and (3) the new address is also stored within the old object's header for other objects to discover. With the last two steps, G1 ensures that pointers are appropriately adjusted, maintaining the integrity of the object graph. Traversing the evacuated

object's references, the directly reachable objects are found to be live and the evacuation process continues. After the copying has been completed, the regions in the CSet are reclaimed and freed.

To identify the live objects inside the CSet, the object graph is traversed starting from the roots. The 'root set' contains the JVM roots, the RSets of the regions in the CSet and the H2 cards that need to be scanned. Objects are evacuated to a new set of regions, either in H1 or if we are on a mixed GC, in H2.

As previously mentioned, the TeraHeap implementation is encapsulated within the 'allocator,' the key entity responsible for managing the H2 heap. The 'allocator' was originally built to work along with the Parallel Scavenge collector. In Parallel Scavenge, TeraHeap moves objects in H2 during full collections, which are single-threaded by default. Therefore when the 'allocator' is asked to give the next available address in H2, it can only handle one response at a time. The evacuations in G1 though are multi-threaded. Thus we surrounded this action with locks. When many threads are trying to transfer objects in H2 at the same time, they will have to wait to get the new location that they can evacuate into. Otherwise, there may be problems like two threads trying to copy two different objects to the same H2-address, resulting in a corrupted heap. In more detail, the two evacuations are extended as follows.

**Young Collections** During young collections, we are not moving objects in H2. But the following adjustments were made (1) fence the collector from scanning objects in H2 and (2) prevent reclamation of H1 live objects that are referenced from H2 (tracking of backward references). For the first task, G1 distinguishes if a reference is from H1 or H2, by using a reference range check and manages to fence the traversal. For the second task, we scan the H2 card table, but only the cards with the state `youngGen` and `dirty`. The backward references are adjusted to point to the new location of the evacuated objects in H1. Since the H2 dirty cards are scanned, their state is updated to `clean/youngGen/oldGen`.

**Mixed Collections** Mixed collections are incrementally reclaiming a part of the old generation. Each mixed collection collects the whole young generation and selects a few old regions out of the sorted candidate region set. If the evacuation manages to complete before the pause time limit, then there is a possibility that G1 can collect even more space in the old generation. If the remaining pause time allows it, G1 has an optional CSet that contains the next most efficient old regions from the candidate set, and tries to collect them as well. The old generation is collected incrementally through multiple mixed collections. Once G1 determines that it has reclaimed a sufficient amount of old space, and deems further old generation collection inefficient, it transitions back to performing young collections.

Mixed collections have the same attributes as the young collections, with the only difference being that during the H2 card table scanning, we also scan the

oldGen cards. Moreover, we extended this evacuation to move objects in the H2 as well. If an object has its `tera` flag enabled then G1 knows that its relocation must be in the secondary heap.



# Chapter 4

## Evaluation

We implement TeraHeap in G1 GC of OpenJDK17 and evaluate it with 8 widely used applications in Spark. Our benchmarking experiments were conducted on an Intel Xeon machine with 32 CPUs of E5-2630 v3, each running at a clock speed of 2,40 GH. It has 256 GB of RAM and it operates on the CentOS Linux 7 platform with x86\_64 architecture. The machine also has an NVMe of 1,6 TB that we will use as a fast storage device.

We use Spark v3.3.0 with Kryo Serializer [35], a state-of-the-art highly optimized S/D Library for Java, recommended by Spark. For each instance of Spark, we use one executor with 8 mutator threads. To capture the effect of large datasets and limited DRAM capacity, we used *cgroups* to restrict the available DRAM for all processes in a single instance of Spark.

We conducted benchmarks by running both native Spark and TeraHeap-integrated Spark, comparing their performance under the same DRAM sizes. A specific budget from the *cgroup* DRAM was allocated for the managed heap (H1), with the remaining DRAM devoted to system-related purposes, such as Spark drivers and page cache I/O. For every benchmark, we divided the DRAM differently between native and TeraHeap runs.

### 4.1 Native Spark Configuration

We use native Spark to compare it with TeraHeap and see the effects that we have on the Spark application performances. Benchmarks with native Spark use `MEMORY_AND_DISK` as storage level. Therefore the executor’s memory (H1 heap) is placed in DRAM, and persisted RDDs are placed either on the on-heap cache (H1) if there is any space left, or they are serialized in the off-heap cache over the NVMe. Standard practices [36, 37] have used 70% - 80% of DRAM capacity for the JVM heap. For this reason, we also set the managed heap (H1) to be between 70% - 80% of DRAM, and report the best results.

Native Spark employs direct I/O for storing and loading data on the off-heap space, bypassing the page cache. Thus, the rest of the DRAM that is not used for

H1, is exclusively utilized by the Spark drivers.

## 4.2 Spark Configuration for TeraHeap

TeraHeap’s configuration is comparable to native Spark, in that we configure both to use similar resources. We configure TeraHeap to allocate the managed heap (H1) on DRAM and memory-mapped the secondary heap (H2) over a file in the NVMe device [21, 22]. The H2 heap is mapped to the JVM virtual address space where the application can directly access the data without any S/D.

With TeraHeap-integrated Spark, the I/O mapping allows us to use the MEMORY\_ONLY option for storage level. This enables TeraHeap to have direct access to the deserialized objects in off-heap storage device (H2). Spark operates without awareness of any specific device, as the operating system takes control of the I/O processes. Because H2 is mapped onto the device, this can include I/O wait due to page faults when accessing H2. Thus, the remaining DRAM here is used not only by Spark drivers but for page cache as well to reduce the I/O wait. The division of DRAM here is hand-tuned and we report the best results.

## 4.3 Tuning G1 GC

G1 collects the heap in two ways: evacuation pauses and full GC. Evacuations are small pauses that collect only a part of the heap, while the full GC collects the whole heap resulting in a larger pause. In our implementation, we target streaming and real-time applications requiring small pauses. Thus, we implemented TeraHeap only within the evacuations. Implementing TeraHeap within the full GC is outside the scope of this thesis and left to future work.

Notably, if the secondary heap is not empty, running the native implementations of a full collection would lead to a corrupted heap, because it cannot track backward references (H2 to H1). The garbage collector must not reclaim H1 live objects that are referenced by live H2 objects. Therefore in our experiments, we tuned G1 GC to avoid full collections, by using the flags listed in Table 4.1. Also, we run all our benchmarks with 8 GC threads to parallelize the collection process, by setting the flag `-XX:ParallelGCThreads=8`.

G1 results in a full GC when the old generation is heavily fragmented, and there is not enough old space to promote objects into, or for humongous allocations; since humongous objects are directly allocated in the old generation. To eliminate the full GC occurrences by fine-tuning G1 GC, firstly it is important to understand under what conditions G1 may trigger such an event. Full collections can occur unpredictably, including during a Concurrent Marking Cycle or while mixed collections are happening. Our goal is to proactively prevent both of these scenarios. The initial three flags in Table 4.1 are used for the tuning of the Concurrent Marking Cycle, the following four flags are for the mixed collections tuning and the last one is for the humongous objects.

G1 initiates the start of a Concurrent Marking Cycle, when the IHOP threshold is met, which is the occupancy percent of the old generation. By default that is 45%, but if the application keeps allocating and promoting objects at a high rate, then the old generation occupancy may reach a much higher level (like 90%), before even the Concurrent Marking finishes. In that case, G1 will trigger a full GC and the Concurrent Marking will be aborted. To avoid this scenario we lowered the IHOP percent to 10%, thus we ensure that G1 will initiate the start of the Concurrent Marking earlier, before the old generation occupancy reaches high levels to the point of initiating a full GC.

If Concurrent Marking Cycles are starting early enough, but are taking a lot of time to finish then there is also the possibility of G1 initiating a full GC. That's because the subsequent mixed collections are delayed, therefore the old generation is not timely reclaimed. Increasing the concurrent marking threads will help the marking process to finish faster. We have configured these threads to the count of 8.

After the Marking is finished, mixed collections follow, incrementally reclaiming space in the old generation. If the application's promotion rate is higher than the rate at which we reclaim memory in the old generation, then a full GC can be initiated. That's because mixed collections did not reclaim enough space in the old generation quickly enough. How much old space they are able to reclaim, is tuned by numerous flags. Therefore, we have fine-tuned G1 GC to ensure that mixed collections can collect as much of the old generation as they can, in a single operation emulating the behavior of a full collection. Thus all the candidate old regions found during the Concurrent Marking will be collected in a batch, reclaiming enough old space in time, for a full GC not to occur. As shown in Table 4.1 we have also tuned the pause time goal to a higher target, as the mixed collections can only reclaim space under this time limit.

Moreover, one of the main contributors to significant heap fragmentation in G1, is the presence of humongous objects. As explained in Section 2.4.3, these are bigger than half of the G1 region size and they are directly allocated to the old generation in a continuous set of regions. These regions can accommodate only one object, which is the humongous object. Therefore the last region of such an allocation is unused, resulting in a waste of heap space. When many long-lived humongous objects exist, G1 exhibits significant heap fragmentation in the old generation, resulting in frequent full collections. We minimize humongous objects, by making the G1 regions larger. More specifically we set the region's size to 32 MB as it is the maximum region size allowed by G1.

Humongous objects are never moved because it is too expensive to do so, they are only reclaimed when they are deemed to be dead. Thus, such objects are never included in the CSet of TeraHeap's mixed collections and even if they are candidates for H2 placement they are never moved off-heap. By minimizing the amount of objects G1 considers as humongous, we also enable TeraHeap's mixed collections to transfer more objects off-heap, reducing the memory pressure of the managed heap even more. Less memory pressure means less likelihood of initiating

-XX:InitiatingHeapOccupancyPercent=10	Sets the old generation occupancy threshold that triggers a marking cycle. The default is 45%
-XX:-G1UseAdaptiveIHOP	Turn off this behavior of G1, of adapting the IHOP percent
-XX:ConcGCThreads=8	Sets the number of parallel marking threads
-XX:MaxGCPauseMillis=50000	Sets a target value for the desired maximum pause time. The default value is 200 milliseconds.
-XX:G1OldCSetRegionThresholdPercent=100	Sets an upper limit on the number of old regions to be included in the CSet of a mixed collection. The default is 10 percent of the Java heap
-XX:G1HeapWastePercent=0	Sets the percentage of heap that you are willing to waste. If the total reclaimable bytes from the old generation is less than this threshold, then G1 reverts back to young collections.
-XX:G1MixedGCLiveThresholdPercent=100	Sets the maximum liveness of an old region to be included in a mixed collection. If there are no such old regions, G1 reverts back to young collections. The default is 65%
-XX:G1HeapRegionSize=32m	Sets the size of a G1 region. The value can be a power of two and can range from 1MB to 32MB.

Table 4.1: Flags used for tuning G1 GC, to avoid full collections[1, 2].

a full GC.

## 4.4 Workloads and datasets

For our experiments with Spark, we use the eight memory-intensive workloads from the Spark Bench suite [38], used in the Parallel Scavenge implementation of TeraHeap [4]. These workloads include five graph-based workloads from GraphX [39]: Page Rank (PR), Connected Component (CC), Shortest Path (SSSP), SVDPlusPlus (SVD) and Triangle Count (TR), and three machine learning workloads from MLLib [40]: Linear Regression (LinR), Logistic Regression (LogR) and Support Vector Machine (SVM).

Considering the volume of cached data associated with each workload, we sought to determine a suitable DRAM size that cannot accommodate all of them, simulating the effect of big data applications. The reported DRAM size is also influenced by the benchmarks' ability to progress within these constraints. The Table 4.2 contains information about each workload, concerning their dataset size and the DRAM division for the execution of TeraHeap and native Spark. For the native Spark, the managed heap (H1) occupies approximately 70% - 80% of the DRAM capacity. The reported H1 size reflects the configuration where no out-of-memory errors were encountered during execution. TeraHeap's managed heap



Benchmark	Dataset (GB)	DRAM (GB)	H1 Native (GB)	H1 TeraHeap (GB)
<b>PR</b>		10	8	5
<b>CC</b>	3 GB	14	11	6
<b>SSSP</b>		10	8	8
<b>TR</b>	192 MB	10	8	8
<b>SVD</b>		9	7	6
<b>LogR</b>		7	5	2
<b>LinR</b>	24 GB	7	5	2
<b>SVM</b>		11	9	2

Table 4.2: Configuration of each workload for native Spark and TeraHeap integrated Spark.

(H1) size is hand-tuned and we reported the best results that the application could progress in, without initiating a full collection.

## 4.5 Time breakdown

Each experiment was conducted five times and we report the average end-to-end execution time. The execution time is broken down into four components: other time, S/D time, young GC time, mixed GC time, and full GC time.

Other time encompasses mutator threads time, concurrent processing (Concurrent Marking and refinement processes) and the I/O overhead caused by the cache misses. In TeraHeap, the off-heap cache (H2) is memory-mapped onto the device, thus the other time also includes I/O wait due to page faults. S/D time includes both shuffle and caching S/D time. In TeraHeap though, all S/D time is attributed only to shuffling, as the cached objects are stored in their deserialized form due to the use of the MEMORY\_ONLY storage level. The JVM reports the time spent for each GC through a log file [41].

S/D overhead takes place within the mutator threads. For its estimation, we use a sampling profiler [42] to collect execution samples from the stack trace of these threads. We grouped all the samples for the paths that originate from the top-level writeObject() and readObject() methods of the KryoSerializationStream and KryoDeserializationStream classes. These samples include both shuffle and caching paths of Spark. Then we calculate the ratio of S/D samples to the total samples provided by the profiler as an estimation of the time spent in S/D. This information is plotted separately in our execution time breakdowns. The profiler operates with a 10 ms sampling interval, ensuring minimal overhead. That is the same technique used in the original TeraHeap implementation [4].

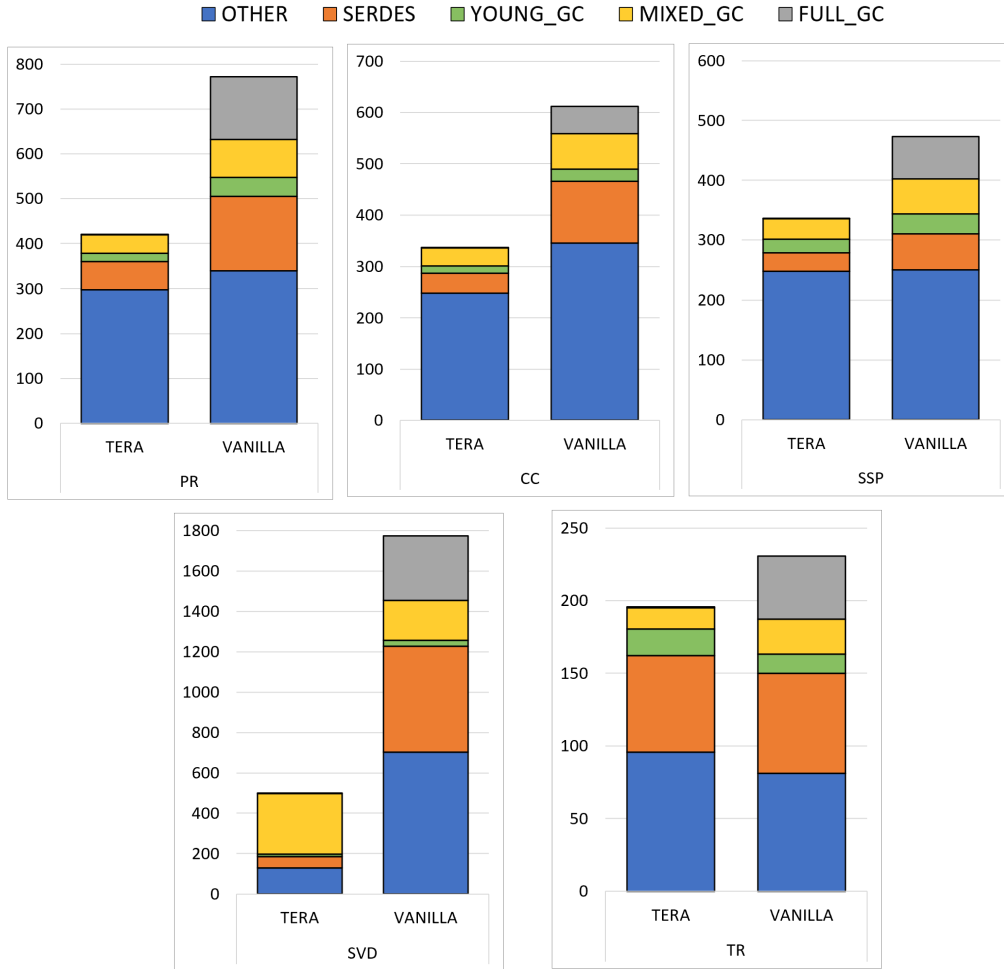


Figure 4.1: Performance of native and TeraHeap integrated Spark in the graph-based workloads

## 4.6 Results

Figure 4.1 and Figure 4.2 show our performance results of TeraHeap and native Spark. We found that using the same DRAM size, TeraHeap reduces execution time between 15% (SVM) and 72% (SVD) compared to native Spark. Also, TeraHeap provides better performance. The improvement in GC overhead reaches up to 78%. This overhead primarily arises due to the occupation of the heap by cached objects in Spark, leading to more frequent GC triggers. On the other hand, TeraHeap transfers objects to H2 which places less strain on H1. Moreover, TeraHeap reduces S/D cost between 3% (TR) and 92% (LogR), as it allows for direct access to the deserialized objects in H2. The outcomes of our evaluations are illustrated and analyzed in this section.

Note that in several benchmarks, the Other time differs between the native and

TeraHeap implementations. This difference can be attributed to a combination of locality effects by TeraHeap object reordering, and delays caused by page fault I/O. Specifically, TeraHeap gradually moves all the persisted RDDs in H2 according to their transitive closure, shaping its page locality. Objects that are of the same RDD are grouped together, ensuring their close proximity in the device (H2). By allowing direct access and computation over H2 objects, TeraHeap optimizes data operations with a favorable cache locality, improving the mutator threads time (Other time). When native Spark exhibits poor cache locality, our charts will reflect the TeraHeap locality effect through a reduction in TeraHeap’s Other time. However, if native Spark demonstrates a good cache locality like TeraHeap, then the impact of I/O page faults will cause an increase in TeraHeap’s Other time. We observed both of these scenarios in our evaluation measurements.

In the case of PR, CC, SSSP, and SVD the page locality of TeraHeap aligns well with the specific memory access patterns and usage scenarios. Figure 4.1 displays the performance of those workloads. TeraHeap outperforms native Spark, exhibiting less cache misses and achieving a notable reduction in Other time. In these graph-based workloads, we note an improvement in the GC cost of 78%, 67%, 65% and 43% in PR, CC, SSSP, and SVD, respectively. TeraHeap transferred lots of the objects in H2, without the need to bring them back into the managed heap like native. Thus the frequency of garbage collection and also their duration has been reduced, compared to native Spark. Resulting in an improvement to the overall GC overhead.

Moreover, in the SVD workload, there is a notable improvement in the S/D cost. This improvement highlights the fact that the SVD workload caches a significant number of RDDs. Native Spark incurs a significant cache S/D overhead, as it undergoes serialization each time it offloads persisted RDDs to the device and deserialization each time it brings them back on-heap for processing. The limited size of the managed heap and the large volume of cached data intensify the recurrent movement of cache data to and from the managed heap, leading to heightened S/D cache overhead. However, TeraHeap doesn’t have S/D cost for caching the objects. TeraHeap stores the persisted data in their deserialized form in H2, and once transferred to the device they are never moved. Therefore it only incurs the S/D costs for shuffling. This applies to all the other workloads as well, but in the SVD it was particularly pronounced. Additionally, the S/D overhead on the PR, CC, SSSP, and SVD workloads sees reductions of 62%, 68%, 49% and 89%, respectively.

As for the TR workload, its cached data fits on the on-heap cache. Smaller heap sizes led to an out-of-memory error. In native Spark, the primary S/D overhead stems from the shuffling process. Given that native doesn’t place a lot of data off-heap because they can fit on the managed heap, it doesn’t have a lot of S/D caching overhead. Both native Spark and TeraHeap incur shuffling overhead, but since TeraHeap doesn’t introduce extra caching overhead, we expect to see only a small difference in the S/D times. Figure 4.1 shows a reduction in the S/D cost by 3% and for the GC cost an improvement of 59%. TeraHeap is similar to native,

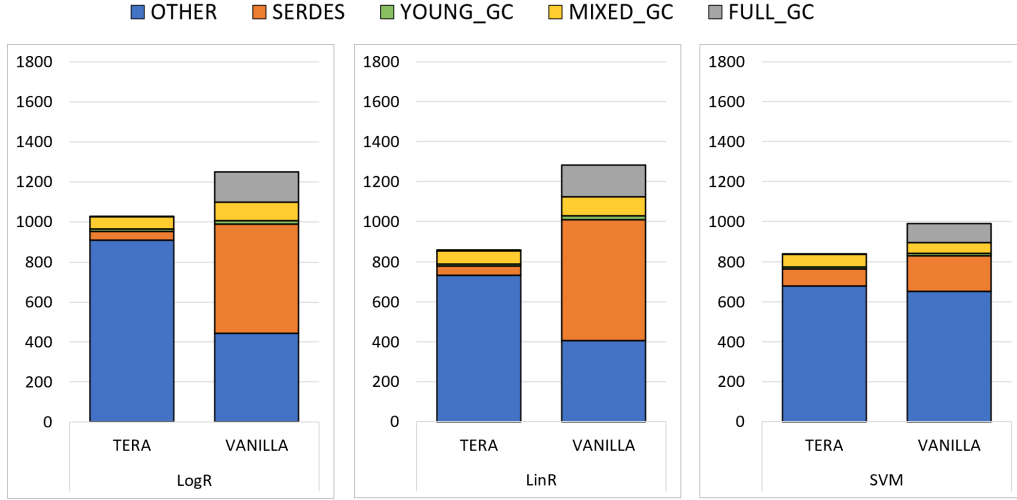


Figure 4.2: Performance of native and TeraHeap integrated Spark in the machine learning workloads

but due to the major page faults occurring, there is a slightly larger other time. During the mixed collections of the TeraHeap, 73% of the cached data have been transferred to H2, leading to potential page faults upon access. However, native Spark experiences minimal off-heap transfers, as indicated by the small deviation of the S/D cost. Therefore most of the cached data in native Spark resides on-heap, which minimizes additional I/O overhead.

In the machine learning (ML) workloads, the training phase involves 100 iterations, during which streaming access is performed on the cached RDDs. Given the small size of the page cache and the largeness of the dataset, this scenario induces numerous major page faults in TeraHeap. As a consequence, data retrieval directly from the storage device becomes necessary during each computational iteration, which comes with an I/O overhead. However, native Spark avoids the page fault penalty by bypassing the page cache. Nevertheless, it loses this advantage through the caching overhead, as it serializes and deserializes the RDDs while it brings them back and forth into the managed heap. Figure 4.2 displays these ML workloads and their performance. The other time with TeraHeap is increased by 45%, 51% and 4% in LinR, LogR, and SVM, respectively, compared to native Spark due to the page faults occurring. Therefore, these workloads benefit more from a larger page cache. Moreover, in LinR and LogR there is an improvement on the GC overhead of 72% and in S/D 92%. As for the SVM workload, there is an improvement of 54% in the GC cost, and 52% in the S/D cost.

Throughout all our experiments, we observed a notable reduction in both the S/D cost and the GC cost. The cached RDDs were efficiently transferred to the secondary heap (H2), relieving the managed heap (H1). Thus, the managed heap

in TeraHeap experiences less memory pressure compared to native Spark. Consequently, the frequency of the collections has been diluted, and the number of garbage collections occurring has dropped by up to 83%. Moreover, TeraHeap doesn't have to scan long-lived cached objects during its garbage collections, as they have been moved in H2. In contrast, native Spark because of the on-heap cache, includes such objects in the garbage collection scans, introducing an additional overhead.

While we did not completely eliminate humongous objects, their count has been reduced between 0 and 2 GB at most. This signifies that in big data applications, there might be instances where objects exceed 16 MB, which is half the size of a G1 region. Moreover, we estimated that 1% - 3% of the GC time is devoted to the scanning phase of the H2 card table, to find backward references.

With native Spark, when the application wants to access an element of a cached RDD partition and is not on-heap, then it's a cache miss. For each cache miss, Spark brings back into the managed heap the whole partition, not just the element the application was trying to access. Thus the memory pressure is heightened due to the cached data on-heap. Thus we note that TeraHeap has a reduction in GC frequency by up to 83%, and the pause times duration are shorten by up to 82%. It's worth noting that the native Spark was not able to avoid full collections, even with the fine-tuning of G1 GC. It has a greater need for full collections contrary to TeraHeap, because of its high memory pressure. During the TeraHeap runs, we noticed that almost all of them necessitated an initial full garbage collection, caused by metaspace allocation failure, which is the space in JVM that holds class metadata. These full collections were triggered before any mixed collections could take place, ensuring that the H2 heap remained empty, and thus, the full collections did not risk creating a corrupt heap. Subsequently, TeraHeap did not need to initiate another full garbage collection. This observation highlights that, despite having a smaller DRAM heap (H1) in certain workloads as shown in Table 4.2, TeraHeap successfully mitigates memory pressure. Thus, TeraHeap also helps G1 GC to stay true to its promise and reduce the costly fallbacks to a full GC.

Our results align closely with those reported in the original TeraHeap paper that uses the Parallel Scavenge Collector [4]. Figure 4.3 displays those results. Note that TH is the TeraHeap performance and Spark-SD is the native. Moreover, the benchmark LR is the linear regression and LgR the logistic regression, whereas ours were called LinR and LogR, respectively.

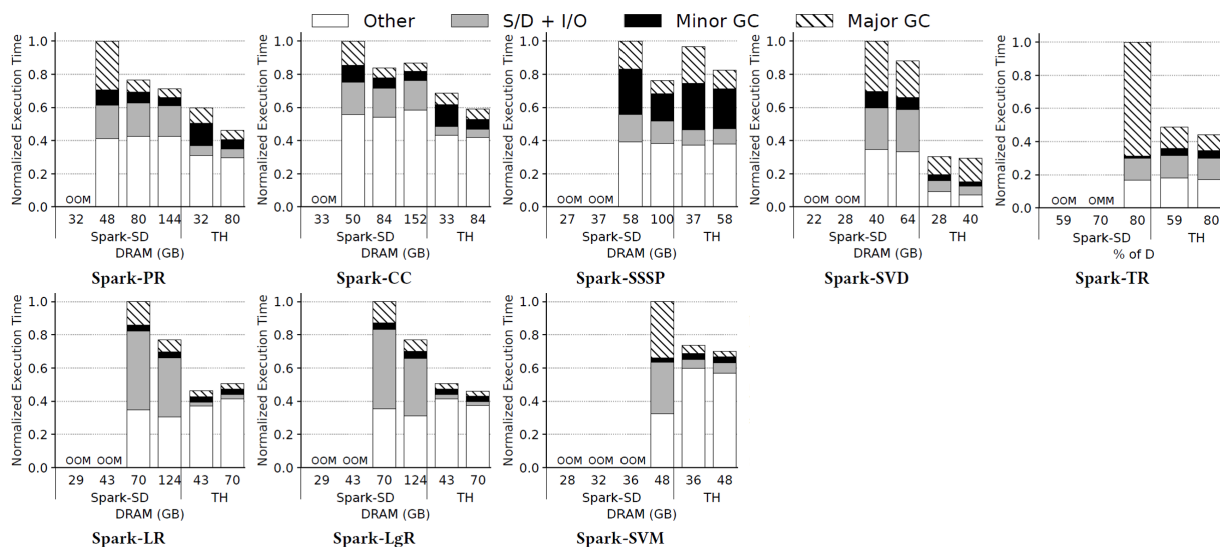


Figure 4.3: These are the findings presented in [4] for the TeraHeap implementation in the Parallel Scavenge Collector. They closely resemble the findings we presented for TeraHeap in G1 GC.

# Chapter 5

## Related work

Related work falls in four categories: (1) off-heap cache over a fast storage device, (2) region-based memory management for big data systems, (3) minimizing S/D overhead that comes with off-heap processing, and (4) why we chose TeraHeap compared to the other works

Managed big data analytics frameworks face challenges when dealing with growing datasets, typically requiring an amount of data that do not always fit in the managed heap. To address this, frameworks often move objects off-heap on a fast storage device. Unfortunately, this introduces high S/D costs and memory pressure when bringing off-heap objects back to the managed heap for processing. Therefore managing objects off-heap presents limitations, including increased S/D overhead and high GC cost, when objects are brought back to the managed heap for further processing.

### 5.1 Off-heap caching

Recent researches focus on NVM or NVMe SSD devices for storing cached data outside the managed heap (off-heap).

Panthera [43] introduced a semantic-aware GC tailored for big data analytics over hybrid memories. This mechanism allocates the new generation in DRAM and splits the old generation into DRAM and NVM, optimizing data object placement through static analysis. However, it increases the GC overhead by scanning and compacting objects on the managed NVM heap, which costs more than collecting the DRAM heap.

TMO [44] monitors application DRAM usage and transparently offloads cold data to an NVMe SSD device. It provides direct access to the off-heap device, without the need for S/D. Nevertheless, it cannot avoid slow GC scans over the device.

Other works [45, 46, 47] target NVM or NVMe SSD for storing managed heaps beyond DRAM, but unlike these works, TeraHeap eliminates slow GC traversals over the fast storage device. It archives this by fencing the garbage collector from

scanning the off-heap data.

## 5.2 Region-based memory management

Broom [48] demonstrates that big-data systems generate objects with predefined lifetimes. By using region-based memory management, Broom efficiently locates objects that reside in shared regions, leading to improved garbage collection (GC) times. However, Broom employs region annotations, necessitating the refactoring of application source code, which adds some additional complexity to the developer.

Facade [49] transforms programmer-specified classes for off-heap allocation. It separates the application objects between the managed heap which is garbage-collected, and the off-heap which is region-based, and reclaims data at the end of each iteration. The developer though, should identify these “boundary classes” and annotate their code, as to specify when they can be freed from off-heap.

YakGC [50] on the other hand, finds objects with similar lifespans as defined by the application, and allocates them in an epoch on a second region-based heap, effectively reducing garbage collection (GC) time. However, it also adds an extra burden on developers, while it requires the annotation of epochs.

In contrast to previous approaches, TeraHeap requires adding hints only at the framework layer. The hint-based interface operates seamlessly at the framework level, remaining entirely transparent to the developers of the applications.

## 5.3 Minimizing S/D overhead

Transferring objects from the managed heap to a fast storage device (off-heap) poses a challenge as frameworks usually can not perform direct computations over them. Consequently, there is a need to relocate these objects back to the managed heap for processing, causing high memory pressure. This reallocation process incurs significant serialization/deserialization (S/D) overhead, particularly for applications employing complex data structures [51, 52, 53]. There are some systems though that support off-heap computation over byte arrays with primitive types [54], but not over arbitrary objects. Other works [55, 56, 57] manage to reduce the S/D cost but they require custom hardware extensions and maybe some modifications to the programming model. Also, some recent works [52, 58, 53] show that by reducing the number of objects copied across buffers, they can also reduce the S/D cost. However, none of these works address the GC cost problem that comes along with the memory pressure on the managed heap.

Attempts like Skyway [51] and SSDStreamer [59] minimize the S/D overhead but do not effectively handle the GC overhead. For instance, Skyway transfers objects directly through the network, and SSDStreamer which is an SSD-based caching system uses DRAM as a stream buffer for SSD devices.



Moreover, there are also several other libraries [60, 61, 35] that are attempting to enhance the S/D efficiency, but also fall short in mitigating the substantial garbage collection (GC) costs associated with big data frameworks.

In this landscape, TeraHeap stands out as the first solution to eliminate both GC and S/D for a substantial portion of objects in big data analytics frameworks. It manages to minimize the S/D by providing direct access to the off-heap, by mapping it [21, 22] over a fast storage device.

## 5.4 TeraHeap contribution

TeraHeap [4], originally designed for the Parallel Scavenge Garbage Collector (GC), has emerged as a powerful solution to the challenges faced by big data analytics frameworks like Spark [5] and Giraph [62]. Specifically, it addresses the common issues of high serialization/deserialization (S/D) costs and the high memory pressure when H2 objects are moved back to the heap for processing while remaining transparent to the end user. TeraHeap, for the same DRAM size, demonstrates significant performance gains including up to a 73% improvement and a 4.6 $\times$  reduction in DRAM consumption, compared to native Spark. Also, it improves up to 28% and needs 1.2 $\times$  less DRAM capacity than native Giraph. Finally, it outperforms Panthera [43], a state-of-the-art garbage collector for hybrid memories, by up to 69%.

Our work builds upon these prior efforts, importing the proven advantages of TeraHeap into G1 GC, which is well-suited for applications with low latency requirements. The integration aims to extend the benefits of TeraHeap to latency-sensitive applications within the big data domain, combining the strengths of TeraHeap’s innovative mechanisms with the efficiency and adaptability of the G1 Garbage Collector.



# Chapter 6

## Future work

### 6.1 Full GC

In the pursuit of optimizing the G1 garbage collector, future work aims for the awareness of the Full GC regarding the secondary heap. Additionally Full GC would be extended to identify the transitive closures of the root key-objects and transfer them in H2.

As the Full GC does not need to meet the real pause time goal, humongous objects can be moved in H2 as well, if they are included in a transitive closure. Those are large objects allocated in continuous regions, where nothing else can be allocated in the regions in which they reside. This can cause fragmentation issues in the managed heap, as they may have significant space left over in their last region, ultimately mitigating the risk of out-of-memory (OOM) errors [27]. Humongous objects are never moved, thus by transferring them off the managed heap, the frequency of garbage collections is expected to decrease.

Furthermore, we could run latency-sensitive application and see the effects it has on latency, without setting the pause time limit and all this tuning we have made. Consequently, the integration of TeraHeap into the full garbage collection of G1 not only addresses the challenges posed by humongous objects but also helps G1 GC to maintain small pauses under the targeted pause time goal, without many fallbacks to a full collection. Upon the completion of this task, we could be able to measure the latency improvements of the applications.

### 6.2 Pause Time Estimation

To meet the pause time goal in the evacuation process, the collection set is carefully chosen to ensure it can be collected within the available time frame [12]. For young collections, the entire CSet consists of young regions. Predicting the number of such regions in advance is essential for meeting the desired pause time goal. Thus G1 keeps track of the real pause times, to dynamically adjust the size of the young generation after each evacuation. This leads to a natural period between

evacuation pauses.

In mixed collections, G1 may include additional old regions if pause time allows. The regions are chosen based on the garbage-first policy. However, the selection of regions ceases when the "best" remaining old region surpasses the specified pause time limit. For this purpose, G1 must estimate the time it needs for an old region to be evacuated, as to check if it can be reclaimed without exceeding the pause time limit.

Therefore G1 needs to estimate the pause time as close to reality as possible, to stay true to his promise of meeting the soft real-time goal. For future work, this estimation could be expanded to consider the scanning of H2 cards and the evacuations in the secondary heap, which are more expensive because of the device. Currently, we only take into account the objects that will be transferred in H2, and we consider them as reclaimable space. These help to promote more objects in H2 but the inaccuracy of the predicted time results in mixed collections exceeding their pause time limit. In our evaluation, this did not happen as we have set the pause time target to a greater limit.

### 6.3 Remember Sets

As of now the backward references (H2 to H1) are identified through the H2 card table scanning. H1 though has remember sets that keep track of incoming pointers for all its regions. Remember sets have different levels of precision [32, 30], and a potential future work would be to see how we could incorporate references from H2 into the remember sets as well. This analysis will shed light on the trade-offs between the two methods and guide the selection of the most effective strategy for handling H2 references.

### 6.4 Asynchronous H2 transfers

During the mixed collections, we transfer objects in H2 via *memcpy* which is a synchronous blocking operation. Therefore, for each object we evacuate in H2, we will be accessing H2 again to adjust its pointers as many times as the amount of its references. Each time one of its referenced objects is evacuated, we will be accessing H2 to update its reference and point to the new location of the evacuated object.

As a part of future work, we envision transforming this process into an asynchronous operation. For example, a way to approach this is as follows. While traversing the CSet object graph, the objects for placement in H2 are identified but not moved. Their new H2 locations are determined and stored within their headers. Even if the H2 objects are not moved yet, their pointers are adjusted while residing in the managed heap. Once all relocations in H1 have been successfully completed, and all the H2 objects have their new location stored in their header, then we can proceed to the relocation process in H2. This can ensure that

all the *tera*-flagged objects have their pointers adjusted before moving them in H2. Therefore there is no need to access the secondary heap for pointer stores, but only for evacuations. To expedite this, H2 objects could be copied in batches asynchronously, introducing efficiency to the overall operation.

## 6.5 Multi-threaded Allocator

“Allocator” is the encapsulated implementation of TeraHeap. To import TeraHeap in G1 GC, we use the allocator interface to interact with the secondary heap. The allocator was originally designed to facilitate single-threaded operations. However, the garbage collections in G1 are multi-threaded. Therefore an interesting direction for future work would be to make the allocator support multi-threaded requests.

This enhancement could eliminate the need for locks when interacting with the allocator’s interface. By doing so, contention is reduced, which can lead to better responsiveness, especially when multiple threads need to make progress without having to deal with lock-related delays.



## Chapter 7

# Conclusion

With the exponential growth of datasets, big data frameworks like Apache Spark demand for a larger heap size. In managed language environments like JVM, large heaps incur excessive GC overhead. Therefore, frameworks avoid using large heaps and resort to expensive off-heap S/D when managing large datasets. However, the repeated S/D creates significant CPU overhead that cannot currently be reduced without increasing GC overhead. Such overheads can be eliminated using TeraHeap, which extends the JVM heap over a fast storage device. TeraHeap is an on-heap cache mechanism, provides direct access over cached data, eliminating both GC and S/D cache overheads. TeraHeap was originally implemented in the Parallel Scavenge Collector for batch-processing applications, and it has shown significant improvements in the applications' performance. In our work, we imported TeraHeap mechanism into the Garbage-First (G1) Collector. G1 has a great balance between throughput and low-latency requirements. The garbage collection pauses under G1, are trying to meet a soft real-time goal with high probability. Thus, we aim for the benefits of TeraHeap, to be passed on to the latency-sensitive application through G1 GC. Our results show that TeraHeap improves performance by up to 72% compared to native Spark. Moreover, the S/D and GC overheads are reduced by up to 92% and 78%, respectively.





# Bibliography

- [1] Monica Beckwith. Tips for tuning the garbage first garbage collector. <https://www.infoq.com/articles/tuning-tips-G1-GC/>.
- [2] Krishnakumar Nallasamy. G1gc new terms and tuning flags. <https://dzone.com/articles/g1gcgarbage-first-garbage-collector-tuning-flags-1>.
- [3] Oracle. Java platform, standard edition hotspot virtual machine garbage collection tuning guide. <https://docs.oracle.com/javase/10/gctuning/garbage-first-garbage-collector.htm>.
- [4] Shoaib Akram Christos Kozanitis Anastasios Papagiannis Foivos S. Zakkak Polyvios Pratikakis Angelos Bilas Iacovos G. Kolokasis, Giannos Evdorou. Teraheap: Reducing memory pressure in managed big data frameworks. <https://dl.acm.org/doi/abs/10.1145/3582016.3582045>, 2023.
- [5] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *In Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. USENIX Association.
- [6] Erci Xu, Mohit Saxena, and Lawrence Chiu. Neutrino: Revisiting memory caching for iterative data analytics. In *In Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'16, pages 16–20, Berkeley, CA, USA, 2016. USENIX Association.
- [7] José Simão Luis Veiga Rodrigo Bruno, Duarte Patricio and Paulo Ferreira. Runtime object lifetime profiler for latency sensitive big data applications. <https://dl.acm.org/doi/10.1145/3302424.3303988>.
- [8] Ting Cao John Zigman Haris Volos Onur Mutlu Fang Lv Xiaobing Feng Chenxi Wang, Huimin Cui and Guoqing Harry Xu. Panthera: holistic memory management for big data processing over hybrid memories. <https://doi.org/10.1145/3314221.3314650>.
- [9] Lijie Xu, Tian Guo, Wensheng Dou, Wei Wang, and Jun Wei. An experimental evaluation of garbage collectors on big data applications. <https://doi.org/10.14778/3303753.3303762>, 2019.

- [10] Iacovos G. Kolokasis. Teracache: Ecient spark caching over fast storage devices. [https://elocus.lib.uoc.gr/dlib/5/5/1/attached-metadata-dlib-1617101997-352719-7563/Thesis\\_Kolokasis\\_Iakwvos.pdf](https://elocus.lib.uoc.gr/dlib/5/5/1/attached-metadata-dlib-1617101997-352719-7563/Thesis_Kolokasis_Iakwvos.pdf).
- [11] Barry Hayes. Using key object opportunism to collect old objects. <https://dl.acm.org/doi/10.1145/117954.117957>.
- [12] Steve Heller Tony Printezis David Detlefs, Christine Flood. Garbage-first garbage collection. <https://dl.acm.org/doi/10.1145/1029873.1029879>.
- [13] Stephen M. Blackburn Wenyu Zhao. Deconstructing the garbage-first collector. <https://dl.acm.org/doi/10.1145/3381052.3381320>.
- [14] Jesse Jie. Linkedin’s journey to java 11. <https://engineering.linkedin.com/blog/2022/linkedin-s-journey-to-java-11>.
- [15] Dain Sundstrom David Phillips Wenlei Xie Yutian Sun Nezhil Yigitbasi Haozhun Jin Eric Hwang Nileema Shingte Christopher Berner Raghav Sethi, Martin Traverso. Presto: Sql on everything. <https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8731547&tag=1>.
- [16] Daoyuan Wang and Jie Huang. Tuning java garbage collection for apache spark applications. <https://www.databricks.com/blog/2015/05/28/tuning-java-garbage-collection-for-spark-applications.html>.
- [17] Yongjian Liao Deleli Mesay Adinew, Zhou Shijie. Spark performance optimization analysis in memory tuning on gc overhead for big data analytics. <https://dl.acm.org/doi/pdf/10.1145/3375998.3376039>.
- [18] Microsoft. Reasons to move to java 11 and beyond. <https://learn.microsoft.com/en-us/java/openjdk/reasons-to-move-to-java-11>.
- [19] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI’12*, page 2, USA, 2012. USENIX Association.
- [20] Spark Code Hub. Spark storage levels uncovered: A comprehensive guide to data persistence in apache spark. <https://www.sparkcodehub.com/spark-storage-levels>.
- [21] Polyvios Pratikakis Iacovos G. Kolokasis, Anastasios Papagiannis and Angelos Bilas. Say goodbye to off-heap caches! on-heap caches using memory-mapped i/o. <https://www.usenix.org/conference/hotstorage20/presentation/kolokasis>.

- [22] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustris, Manolis Marazakis, and Angelos Bilas. Optimizing Memory-mapped I/O for Fast Storage Devices. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '20, USA, July 2020. USENIX Association.
- [23] Philipp Lengauer. Understanding the g1 garbage collector – java 9. <https://www.dynatrace.com/news/blog/understanding-g1-garbage-collector-java-9/>.
- [24] Matt Robson. Part 1: Introduction to the g1 garbage collector. <https://www.redhat.com/en/blog/part-1-introduction-g1-garbage-collector>.
- [25] Monica Beckwith. Garbage first garbage collector. [https://www.youtube.com/watch?v=Io8hEdm6haw&ab\\_channel=Devovx](https://www.youtube.com/watch?v=Io8hEdm6haw&ab_channel=Devovx).
- [26] Akhil. Understanding jvm garbage collection (gc mark , sweep and compact basics) – part 2. <https://abiasforaction.net/understanding-jvm-garbage-collection-part-2/>.
- [27] Stephanie Crater. What’s the deal with humongous objects in java? <https://devblogs.microsoft.com/java/whats-the-deal-with-humongous-objects-in-java/>.
- [28] JEB Moss AL Hosking and D Stefanovic. A comparative performance evaluation of write barrier implementation. [https://www.researchgate.net/publication/234787457\\_A\\_Comparative\\_Performance\\_Evaluation\\_of\\_Write\\_Barrier\\_Implementation](https://www.researchgate.net/publication/234787457_A_Comparative_Performance_Evaluation_of_Write_Barrier_Implementation).
- [29] Jevgēnijs Protopopovs. Throughput barrier exploration for the garbage-first collector. [https://www.protopopov.lv/static/files/masters\\_thesis.pdf](https://www.protopopov.lv/static/files/masters_thesis.pdf).
- [30] William D. Clinger David Detlefs, Ross Knippel and Matthias Jacob. Concurrent remembered set refinement in generational garbage collection. [https://www.researchgate.net/publication/220817732\\_Concurrent\\_Remembered\\_Set\\_Refinement\\_in\\_Generational\\_Garbage\\_Collection](https://www.researchgate.net/publication/220817732_Concurrent_Remembered_Set_Refinement_in_Generational_Garbage_Collection).
- [31] Simone Bordet. G1 garbage collector details and tuning. [https://www.youtube.com/watch?v=Gee7QfoY8ys&ab\\_channel=VoxxedDays](https://www.youtube.com/watch?v=Gee7QfoY8ys&ab_channel=VoxxedDays).
- [32] ANDREAS SJÖBERG. Evaluating and improving remembered sets in the hotspot g1 garbage collector. <http://www.diva-portal.se/smash/get/diva2:754515/FULLTEXT01.pdf>.
- [33] Salman Salloum Tamer Z Emara Mohammad Sultan Mahmud, Joshua Zhexue Huang and Kuanishbay Sadatdiyev. A survey of data partitioning and sampling methods to support big data analysis. <https://ieeexplore.ieee.org/document/9007871>.

- [34] Sourav Choudhary. Exploring the inner workings of garbage collection in golang : Tricolor mark and sweep. <https://medium.com/@souravchoudhary0306/exploring-the-inner-workings-of-garbage-collection-in-golang-tricolor-mark-and-sweep-e10eae164a12>.
- [35] soteric Software. Kryo. <https://github.com/EsotericSoftware/kryo>.
- [36] Jonathan Dowland. Overhauling memory tuning in openjdk containers updates. <https://developers.redhat.com/articles/2023/03/07/overhauling-memory-tuning-openjdk-containers-updates#>.
- [37] Microsoft. Containerize your java applications. <https://learn.microsoft.com/en-us/azure/developer/java/containers/overview>.
- [38] Yandong Wang Li Zhang Min Li, Jian Tan and Valentina Salapura. Spark-bench: a spark benchmarking suite characterizing large-scale in-memory data analytics. <https://link.springer.com/article/10.1007/s10586-016-0723-1#Sec3>.
- [39] Apache Spark. Graphx is apache spark's api for graphs and graph-parallel computation. <https://spark.apache.org/graphx/>.
- [40] Apache Spark. Mllib is apache spark's scalable machine learning library. <https://spark.apache.org/mllib/>.
- [41] Matt Robson. Collecting and reading g1 garbage collector logs - part 2. <https://www.redhat.com/en/blog/collecting-and-reading-g1-garbage-collector-logs-part-2>.
- [42] Andrei Pangin. Async-profiler. <https://github.com/jvm-profiling-tools/async-profiler>.
- [43] Ting Cao John Zigman Haris Volos Onur Mutlu Fang Lv Xiaobing Feng Chenxi Wang, Huimin Cui and Guoqing Harry Xu. Panthera: holistic memory management for big data processing over hybrid memories. <https://dl.acm.org/doi/10.1145/3314221.3314650>.
- [44] Dan Schatzberg Leon Yang Hao Wang Blaise Sanouillet Bikash Sharma Tejun Heo Mayank Jain Chunqiang Tang Johannes Weiner, Niket Agarwal and Dimitrios Skarlatos. Tmo: Transparent memory offloading in datacenters. <https://dl.acm.org/doi/10.1145/3503222.3507731>.
- [45] Kathryn McKinley Shoaib Akram, Jennifer Sartor and Lieven Eeckhout. Crystal gazer: Profile-driven write-rationing garbage collection for hybrid memories. <https://doi.org/10.1145/3322205.3311080>.
- [46] Kathryn S. McKinley Shoaib Akram, Jennifer B. Sartor and Lieven Eeckhout. Write-rationing garbage collection for hybrid memories. <https://doi.org/10.1145/3192366.3192392>.

- [47] Haibo Chen Yanfei Yang, Mingyu Wu and Binyu Zang. Performance gap for copy-based garbage collectors atop non-volatile memory. <https://doi.org/10.1145/3447786.3456246>.
- [48] Malte Schwarzkopf Kapil Vaswani Dimitrios Vytiniotis Ganesan Ramalingan Derek Murray Steven Hand Ionel Gog, Jana Giceva and Michael Isard. Broom: sweeping out garbage collection from big data systems. <https://dl.acm.org/doi/10.5555/2831090.2831092>.
- [49] Yingyi Bu Lu Fang Jianfei Hu Khanh Nguyen, Kai Wang and Guoqing Xu. Facade: A compiler and runtime for (almost) object-bounded big data applications. <https://doi.org/10.1145/2694344.2694345>.
- [50] Guoqing Xu Brian Demsky Shan Lu Sanazsadat Alamian Khanh Nguyen, Lu Fang and Onur Mutlu. Yak: a high-performance big-data-friendly garbage collector. <https://dl.acm.org/doi/10.5555/3026877.3026905>.
- [51] Christian Navasca Guoqing Xu Brian Demsky Khanh Nguyen, Lu Fang and Shan Lu. Skyway: Connecting managed heaps in distributed big data systems. <https://dl.acm.org/doi/10.1145/3173162.3173200>.
- [52] Matei Zaharia Deepti Raghavan, Philip Levis and Irene Zhang. Breakfast of champions: towards zero-copy serialization with nic scatter-gather. <https://dl.acm.org/doi/10.1145/3458336.3465287>.
- [53] Jonathan Stone Changhoon Kim Rajit Manohar Adam Wolnikowski, Stephen Ibanez and Robert Soulé. <https://dl.acm.org/doi/10.1145/3458336.3465283>.
- [54] Apache Software Foundation. Apache arrow: A cross-language development platform for in-memory data. <https://arrow.apache.org/>.
- [55] Sunmin Jeong Jun Heo Hoon Shin Tae Jun Ham Jaeyoung Jang, Sung Jun Jung and Jae W. Lee. A specialized architecture for object serialization with applications to big data analytics. <https://doi.org/10.1109/ISCA45697.2020.00036>.
- [56] YangWeng Qing Yang Dongyang Li, FeiWu and Changsheng Xie. Hods: Hardware object deserialization inside ssd storage. <https://doi.org/10.1109/FCCM.2018.00033>.
- [57] Hussein Kassir Mark Sutherland Zilu Tian Mario Paulo Drumond Babak Falsafi Arash Pourhabibi, Siddharth Gupta and Christoph Koch. Optimus prime: Accelerating data transformation in servers. <https://doi.org/10.1145/3373376.3378501>.
- [58] Gustavo Alonso Konstantin Taranov, Rodrigo Bruno and Torsten Hoefler. Naos: Serialization-free rdma networking in java. <https://www.usenix.org/conference/atc21/presentation/taranov>.

- [59] Jeonghun Gong Wenjing Jin Shine Kim Jaeyoung Jang Tae Jun Ham Jinkyu Jeong Jonghyun Bae, Hakbeom Jang and JaeW. Ssdstreamer: Specializing i/o stack for large-scale machine learning. <https://doi.org/10.1109/MM.2019.2930497>.
- [60] Apache Software Foundation. Apache thrift. <https://thrift.apache.org/>.
- [61] Google. Protocol buffers. <https://developers.google.com/protocol-buffers/docs/javatutorial>.
- [62] Ibrahim Abdelaziz Sherif Sakr, Faisal Moeen Orakzai and Zuhair Khayyat. Large-scale graph processing using apache giraph. <https://link.springer.com/book/10.1007/978-3-319-47431-1>.