TÉCNICO LISBOA

Garbage Collection Optimization for JVM running Big Data Workloads

Duarte Patrício

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Examination Committee

Chairman:	Prof. Doutor. António Rito Silva
Supervisor:	Prof. Doutor. Luís Manuel Antunes Veiga
Member:	Prof. Doutor. António Paulo Teles de Menezes Correia Leitão

October 2015

Acknowledgements

First, and foremost, I would like to give a special thanks to Luís Veiga, who was my thesis supervisor, for all the technical insight, the learning experience, patience and all the help provided.

Then, I would like to thank Miguel Belo for always listening to my progress, for suggesting approaches and for accompanying me on this journey.

Finally, the most important special thanks goes to my family and my girlfriend, who were always patient and never ceased to give me support and strength to carry on, during the good and bad times.

16th of October, Lisbon
Duarte Patrício

–To my family and Cândida

Abstract

Java Big Data applications have increased popularity in the past decade. The increase of Web-based services and of the computing power started an era of reducing large and complex data sets into useful data. The computing of large data sets has strict demands for the scalability of a system in order to increase performance.

Java has been the programmer's choice when implementing Big Data applications due to its cross-platform capabilities and managed runtime. However, it incurs memory overhead, typically required by a managed runtime. This overhead is negligible for small server applications but not for large scale data processing. The Java Virtual Machine (JVM) is the main bottleneck for all Java programs. If a program misbehaves, the JVM takes appropriate action. Large scale data analysis programs can misbehave in terms of the memory bloat and object dispersion. The problem is in the huge amount of objects and collections of objects that these programs allocate. But, the JVM does not handle this problem specifically because it is more general-purpose oriented.

This thesis's work presents a solution to deal with the memory bloat that large objects cause on the Java heap, for the HotSpot JVM. The solution consists on separating the Java heap space to give room to large objects that can be considered part of the memory bloat problem. Since the Java heap requires a garbage collector to collect unused object references and return the freed memory, this work also presents an extension to the default garbage collector in the HotSpot VM, the Parallel Scavenge, to collect on such heap and promote old objects accordingly.

It is shown that this solution provides increased cache, TLB and page locality, giving the possibility to increase the application performance in the long run.

Resumo

Tem-se visto um crescimento na popularidade das applicações de Big Data escritas em Java desde a década passada. O aumento de serviços Web, e a capacidade de computação actual, iniciou uma época de começar a reduzir conjuntos de dados complexos e em grade escala. A computação destes conjuntos de dados deve ser extremamente escalável para que a *performance* seja visível.

A linguagem Java tem sido a escolha preferida para implementar aplicações Big Data. Isso deve-se ao facto do Java ser uma linguagem de multi-plataforma e cujos programas são geridos durante a execução. No entanto, programas em Java incorrem muito uso da memória devido à necessidade do gestor do programa ter metadados sobre todos os objectos. Este aumento do uso da memória é negligível em pequenas applicações de servidor, mas não é negligível para o processamento de conjuntos de dados enormes. A máquina virtual do Java (Java Virtual Machine or JVM) é a principal restrição em todos os programas em Java. Se um programa tiver um comportamento incorrecto, então a JVM deve tomar medidas para o corrigir. Normalmente, programas que lidam com o processamento de conjuntos de dados em grande escala têm tendência para ter comportamentos incorrectos no que toca ao enchente de memória que causam no espaço de alocação (Java heap). Este comportamento deve-se à alocação de uma enorme quantidade de objectos e colecções de objectos.

O trabalho nesta tese apresenta uma solução para lidar com a enchente de memória causada por objectos de grade dimensão, na HotSpot JVM. A solução consiste em separar o espaço de alocação (Java heap) para dar espaço suficiente, e em separado, a objectos de grande dimensão, que são normalmente os causadores dos problemas de enchente na memória. Como a Java heap necessita de um colecionador de lixo (garbage collector) que colecciona objectos sem uso e retorna a memória libertada por estes, este trabalho também apresenta uma extensão ao coleccionador por defeito da HotSpot JVM, o Parallel Scavenge, para que possa promover objectos para as novas separações da Java heap e coleccionar nesta conhecendo os seus limites.

Mostra-se que esta solução pode providenciar melhor uso da applicação (throughput) quando os espaços para objectos de grande dimensão contêm objectos que vivem bastante tempo, e cujos objectos que referencia estão perto e na mesma região de memória, devido ao aumento da localidade que reduz o número de *misses* da cache e da TLB e o número de *page faults* da page table.

Palavras Chave

Reciclagem automática de memória
Máquina Virtual Java
Megadados
Anti-enchimento do espaço de alocação
Gestão de memória do Java

Keywords

Garbage Collection
Java Virtual Machine
Big Data
Bloat-aware Heap
Java Memory Management

Index

1 Introduction **1**

1.1 Overview . . . 1

1.2 Problem Statement . . . 2

1.3 Extended motivation and Roadmap . . . 2

1.4 Research Proposal . . . 3

1.5 Contributions . . . 4

1.6 Structure of the thesis . . . 4

2 Related Work **5**

2.1 Big Data Platforms . . . 5

2.1.1 Key-Value Databases . . . 6

2.1.1.1 Voldemort . . . 6

2.1.2 Column-oriented Databases . . . 6

2.1.2.1 HBase . . . 7

2.1.2.2 Cassandra . . . 7

2.1.2.3 Druid . . . 8

2.1.3 Batch-Data Processing . . . 9

2.1.3.1 Hadoop Map-Reduce . . . 9

2.1.4 Stream Processing . . . 10

2.1.4.1 S4 (Simple Scalable Streaming System) . . . 10

2.1.4.2 Storm . . . 10

2.1.5 Deployment support and coordination frameworks . . . 11

2.1.5.1 ZooKeeper . . . 11

2.1.5.2 OpenStack . . . 12

2.2 Garbage collection . . . 12
2.2.1 Barriers . . . 13
2.2.1.1 Read Barrier . . . 13
2.2.1.2 Write Barrier . . . 13
2.2.2 Non-parallel algorithms . . . 14
2.2.2.1 Mark-and-Sweep . . . 14
2.2.2.2 Copying GC . . . 14
2.2.2.3 Incremental GC . . . 15
2.2.2.4 Generational GC . . . 15
2.2.3 Concurrent collection . . . 16
2.2.3.1 Mostly-Parallel . . . 17
2.2.3.2 Real-time concurrent . . . 17
2.2.3.3 Pauseless . . . 18
2.2.3.4 Continuously Concurrent Compacting Collector (C4) . . . 18
2.2.4 Parallel algorithms . . . 19
2.2.4.1 Immix . . . 20
2.2.4.2 Parallel Scavenge . . . 20
2.2.4.3 NUMA-Aware Parallel Scavenge . . . 21
2.2.4.4 NUMA-Aware Dominant-thread-based copying . . . 22
2.3 Object Layout and Locality . . . 22
2.3.1 Java Memory Layout. . . . 22
2.3.2 Locality Optimization Schemes . . . 23

3 Architecture **27**

3.1 HotSpot Architecture Overview . . . 27
3.2 Garbage collection — Parallel Scavenge . . . 29
3.2.1 Young collection . . . 31
3.2.2 Full collection . . . 32
3.2.2.1 Marking phase . . . 33

3.2.2.2 Summary phase . . . 34
3.2.2.3 Compact phase . . . 35
3.2.2.4 Cleanup phase . . . 37
3.3 From Parallel Scavenge to Bloat-Aware PS . . . 37
3.3.1 Partitioning the old space . . . 38
3.3.2 Fast Klass information . . . 39
3.3.3 Adjusting the space . . . 41
3.4 Collecting on bloat-aware Parallel Scavenge . . . 42
3.4.1 Bloat-aware young collection . . . 42
3.4.2 Bloat-aware full collection . . . 45

4 Implementation **51**
4.1 Partitioning the old-space . . . 51
4.2 Fast Klass information . . . 53
4.2.1 Header region mark . . . 54
4.2.2 Hashing Klass pointer . . . 56
4.3 Bloat-aware young collection . . . 56
4.3.1 Old-To-Young Root tasks . . . 56
4.3.2 Promoting by type . . . 58
4.4 Bloat-aware full collection . . . 60
4.4.1 Decision information . . . 60
4.4.2 Targeting regions . . . 61
4.4.3 Dealing with region stealing . . . 62
4.4.4 Broadcast top-pointers . . . 62

5 Evaluation **65**
5.1 Overview . . . 65
5.2 Choosing Better Locality . . . 66
5.3 Evaluating Object Locality . . . 67

5.3.1 Header word locality 68
5.3.2 Hash Klass pointer locality 69
5.4 Performance benchmark suite 72

6 Conclusion **75**
6.1 Concluding remarks 75
6.2 Future Work 76

List of Figures

3.1 Parallel Scavenge heap layout . . . 30
3.2 A space's layout . . . 30
3.3 Young collection marking . . . 31
3.4 Young collection promotion . . . 32
3.5 Full collection — overview of regions and live objects . . . 33
3.6 Summary phase — Dense-prefix-end and current top pointer . . . 34
3.7 Summary phase — Summarizing the From-space . . . 35
3.8 Compact phase — Filling regions on the old space . . . 36
3.9 Final view of the summary phase and final effective result after compaction . . . 37
3.10 The partitioned bloat-aware old generation . . . 38
3.11 Layout of an array object header with an added word for BDA-region indexing . . . 40
3.12 An array of entries with BDA-region identifiers . . . 41
3.13 Object-start-array fetch of invalid addresses . . . 45
3.14 Allocation of a variable number of PLAB over BDA-regions . . . 46
3.15 Compaction — fetching and filling of regions . . . 48

5.1 H2 — Used space using the hash of the Klass pointer on an array of 800 (a), 1000 (b) and 1200 (c) entries, using the given decision thresholds . . . 66
5.2 Tradebeans — Used space using the hash of the Klass pointer on an array of 800 (a), 1000 (b) and 1200 (c) entries, using the give decision thresholds . . . 67
5.3 Tradesoap — Used space using the hash of the Klass pointer on an array of 800 (a), 1000 (b) and 1200 (c) entries, using the give decision thresholds . . . 67
5.4 Header Region word — H2 — Object locality (in heap words) in each BDA-region 69
5.5 Header Region word — Tradebeans — Object locality (in heap words) in each BDA-region . . . 69

5.6 Header Region word — Tradesoap — Object locality (in heap words) in each BDA-region 70

5.7 Hash of the Klass pointer — H2 — Object locality (in heap words) in each BDA-region 70

5.8 Hash of the Klass pointer — Tradebeans — Object locality (in heap words) in each BDA-region 71

5.9 Hash of the Klass pointer — Tradesoap — Object locality (in heap words) in each BDA-region 71

5.10 Execution times over the previous benchmarks in respect to the bloat-aware PS . 72

5.11 Locality improvement over the L1 Data Cache, the DTLB and the Page Table . . . 73

List of Tables

2.1 Barriers used in GC (for those applicable) 14

2.2 Summary of GC Algorithm capabilities 25

1 Introduction

In order to succeed, we must first believe that we can. — *Nikos Kazantzakis (1883–1957)*

1.1 Overview

The amount of data flowing across the Internet has been growing exponentially. This is due to various factors, but much is given to the continuous increase of the computational power and the widespread use of the Internet network. The *Big Data* paradigm, isn't new. Doug Laney of the former META Group described the 3Vs (velocity, variety and volume) as a challenge for the future (Laney 2001) and, in recent years, it has been a hot topic in the database and distributed computing areas. There is also great interest in Big Data technologies for fields of research. The Apache Foundation and several other businesses, such as Twitter, Facebook and LinkedIn have already contributed with open-source applications currently empowering businesses and research.

Many of the Big Data applications in use today are written in the Java language. This is due to flexible object-oriented design, quick release cycle, portability and worldwide support. However, Java is a language that at runtime is managed by its core engine that is the Java Virtual Machine. And, in order to efficiently manage the Java program, the Java Virtual Machine (hereafter simply referred to as JVM) incurs some memory overhead on the system because it keeps additional information about the object in its header. Although state-of-the-art JVMs are dynamically efficient, there is still no assertive solution to deal with memory hungry programs such as Big Data applications.

Java facilitates application development because it relies on the JVM for its allocations and to free unused memory. Unused memory is reclaimed by the garbage collector (the GC) which tries to be as less intrusive to the actual program execution as it can. Likewise, the allocation is made by modules aware of the garbage collector algorithm internals, to save extra synchronization mechanisms. But, the current mechanisms for allocation and collection are still not aware of the increased memory footprint under Big Data. Thus, a study of the underlying JVM mechanisms is essential to scale Big Data applications and achieve continuous performance and availability. The mechanisms of the JVM that could mitigate these problems include the memory placement (allocation) and garbage collection schemes.

1.2 Problem Statement

Java Big Data technologies have high demands, in terms of performance and scalability, and depend on the JVM to execute. It is known that some of this technologies will cause severe bloat in memory and, consequently, performance degradation. A study from the University of California (Bu et al. 2013) stated that the **low-packing factor** of the memory and the **large volumes of objects and references** are the key downgrading factors for Java Big Data applications. This **bloat-unawareness** in memory degrades the performance and possibly the availability of the service.

One way to deal with the **bloat-unawareness** of the JVM is to implement a new design, such as suggested in (Bu et al. 2013). However, this would require the reprogramming of much of the code base and, ultimately, that would solve only part of the problem for a specific application. The other part is the inherent overhead of a managed language, since it requires metadata for tracking the objects within the JVM.

At the JVM level, the only information it has of the application that is running are the classes it uses and the amount of memory required to allocate its objects. This information, if handled properly, can help navigate to the memory bloat problem and adjust the allocation space according to the application needs, thus dealing with the **low-packing factor**. Therefore, making the Java heap to become *bloat-aware* and deal with the **low-packing factor** is the main motivation for the work now presented.

1.3 Extended motivation and Roadmap

Computation over large datasets is becoming increasingly common. This led to the introduction of the Big Data paradigm which is spreading fast in businesses and research activities. However, large datasets may produce memory bloat thus degrading performance and scalability.

The two culprits for the limited performance in some Big Data installations in memory restrained systems is the application itself and the JVM. Tuning the application code is impratical given the increasing number of applications. And, in fact, the JVM is the main bottleneck and is the common platform for all applications.

All objects go into the Java heap, a runtime structure of the JVM closely managed by the GC module. Management of the heap consists on dividing the virtual space, committing memory on the virtual space divisions, interpreting allocation calls and triggering a garbage collection when the spaces are exhausted. This shows that the GC module has great impact on the throughput of the application. Therefore, the GC module components require adaptation to become *bloat-aware*.

From the works of (Moon 1984; Wilson et al. 1991) it was shown that *bloat-awareness* can be achieved optimizing locality for the objects in the heap. In fact, Moon's work (Moon 1984) deals with a Large LISP system with *Mark&Sweep* and *Copying* algorithms, which is still the norm. There is a similarity when translating his work to a Big Data system. And, in (Wilson et al. 1991) reorganization techniques that improve locality were demonstrated. These showed that there is improvement on the traversal algorithm when objects of a certain type are given special treatment. On the other hand, the work in (Ilham & Murakami 2011) showed that co-locality schemes can grant cache, page and TLB hit improvements.

This work consists on optimizing the Sun's HotSpot JVM implementation of a Java heap to give special treatment to certain objects, considered to be possible candidates for causing memory bloat, and to garbage collect that heap. It tries to deal with the **low-packing factor** presented in (Bu et al. 2013) by approximating special objects to those it references, and is inspired by the works of (Moon 1984; Wilson et al. 1991; Ilham & Murakami 2011).

The first step on this thesis's report will be to show where to give room to special objects, how to effectively pack those together and which object types should be included in the first implementation. Next it is shown how to efficiently, i. e., not causing much latency, acquire special object information. It finishes on how to garbage collect with special objects in different memory regions.

1.4 Research Proposal

The JVM has its main allocation space on the heap managed by the VM thread (called the Collected Heap). On the heap, Thread Local Allocation Buffers (TLAB) are allocated in the eden space and assigned to each mutator (Java Thread) to avoid races against the other mutator threads at allocation time. On the other hand, the GC Threads operate on the generations based on the tasks assigned to them. "Heavy" objects can fill a whole TLAB, generating three scenarios: forcing the allocation of a new TLAB, placing the object on the shared heap or triggering a collection. Although there is support for a NUMA architecture to allocate more efficiently, by having chunks of mutable space assigned to nodes, there is no native support for the placement of objects that tend to be heavy on memory load.

The main goal of this work is to implement new allocation strategies for heavy objects in the Sun's HotSpot JVM. The new allocation module can be featured in the product version of the virtual machine and can be configured through certain argument options, such as promotion decision mechanisms. The allocation strategy allows packing dependant objects in separate regions of the heap, in turn reducing memory bloat over certain regions.

However, the separation of heavy, tendentially long lived, objects is still not enough to observe significant improvements in the throughput of the application. The special objects are still collected at the same time of the others and, although the traversing of the references is

reduced, long lived objects do not need to be collected as often. The ideal solution is to have a GC instance watching the memory regions of the special objects, concurrently executing with the application threads. This GC instance, in quorum with the others in the pool of watchers, could issue a full collection in parallel for the case when its memory region has high occupation of dead objects.

Deciding the special object types is also not a trivial task. It cannot be known at startup and by inference it is error-prone (an error would cause a mostly empty portion of the heap). There are two solutions to deal with dynamic set up of special region spaces: profile-based and by an API. A profile-based approach would decide based on the statistics of the allocation of large objects, whether at runtime or previously by benchmarking the application. On the other hand, an API approach would expose a programmable API that wrapped tendentially large objects, such as Java collections, in order to help the programmer specify what he knows to be potencial point of memory bloat.

1.5 Contributions

The main contributions of this thesis are:

- An extension for the Java heap which handles potencial large objects.
- A garbage collector extension to operate on a heap divided into segments, promoting objects to these segments based on a configurable decision criteria.
- Two approaches to save object's special status information.

1.6 Structure of the thesis

The remaining of this thesis report is organized in a sequence of chapters, ranging from previous work that inspired this to an evaluation methodology. It ends with some concluding remarks and future work. In Chapter 2, the relevant related work that inspired this thesis's approach is analysed. Next, in Chapter 3, the insight of this work's solution, the decisions taken, the designed architecture and the methodology in order to implement this solution is presented. Our implementation of the designed architecture is then presented in Chapter 4. To assess our implementation's advantages, Chapter 5 shows the evaluation of our architecture's decisions using benchmarks frequently used in the literature. For last, Chapter 6 concludes this report with some concluding remarks and a summary of the planned future work.

2 Related Work

> Start by doing what's necessary; then do what's possible; and suddenly you are doing the impossible — *Francis of Assisi (1181–1226)*

This chapter presents the most relevant work regarding this thesis's goals. There are two main subjects to cover: the currently available open-source Big Data platforms written in Java and, at the lower level, the mechanisms of garbage collection, allocation and object placement in the virtual machine. Higher focus is given to the NoSQL (Not Only SQL) database platforms and the Apache Hadoop project, being the latter one an umbrella project with various platforms. The chapter is then organized in a top-down view, first starting to describe the relevant Big Data platforms that could improve under this work's solution in Section 2.1, and then the lower-level mechanisms relevant to the decision of the platform that is the basis of this work in Section 2.2. In Section 2.3 some lower level details for optimization levels in respect to the available literature are shown.

2.1 Big Data Platforms

To make sense of the unstructured data flowing across the Web, more specific platforms were developed. Conventional, general purpose database stores, like **RDBMS**, **DBMS** or **ORDBMS**, do not quite fit for the storage of data in large quantities, due to their lack of scalability, throughput and dynamism. The storage technologies in large scale data analysis provide different mechanisms contrary to the tabular view of relational databases. These are commonly referred to as NoSQL databases (Not Only SQL), meaning that they may also provide SQL-like queries, but their main form of access is through lower-level query languages, usually specific to the application. NoSQL databases, divided into **Key-Value Stores**, **Column-oriented Stores**, **Document Stores** and **Graph Stores**, are, therefore, purpose oriented. On the other hand, distributed large scale data processing frameworks provide tools for reducing the data sets into useful information.

Databases and file systems are the integral part for storage technologies. Big Data requires that the databases and file systems powering the cluster to be high-throughput oriented. But, the throughput must also be scalable. Many Big Data technologies are designed for commodity hardware machines, trusting in the distributed computing design of the applications for

performance. Therefore, the design of the application is targeted for a specific workload, with strong implications in the type of storage it provides.

This thesis work gives, in the next paragraphs, an overview over the **Key-Value Stores** and **Column-oriented Stores**. It does not give importance to the filesystems used in Big Data, since most of them are not written in Java.

2.1.1 Key-Value Databases

Key-Value store databases have, as their fundamental data model, a map of their contents structured as key-value pairs. The key is often unique and is used as an identifier for the value, although its usage may differ. This kind of storage offers the advantage of quick lookups and fast loading onto memory.

2.1.1.1 Voldemort

Voldemort[1] is an open source key-value storage solution implemented and in use by LinkedIn (Sumbaly et al. 2012). It is based on Amazon's DynamoDB as in being a highly available huge distributed hash table (DHT). A Voldemort cluster contains multiple nodes where each physical host can run more than one node. Also, it allows key-value serialization and flexible storage engine type, just as Dynamo's. To achieve high availability and consistency, a cluster running Voldemort first sets a list of parameters: **replication factor**, **required reads** and **required writes**. The replication factor sets the number of nodes where a key-value pair is replicated on (**replication factor** parameter). On the other hand, the required reads and required writes parameters set the minimum number of nodes Voldemort reads or writes from in parallel, respectively. This is how consistency of the tuples is guaranteed. For replication across the cluster, Voldemort uses a DHT in a ring topology. The ring is shared across the nodes, hence every node is aware of the others in the cluster. At the time of replication a **preference list**, with all the nodes in the cluster, is computed and, through hashing the key of the tuple, it is replicated at the according nodes. The preference list allows the replication mechanism (the **routing** module) to select the preferred node in which to a allocate replicas of the tuples.

2.1.2 Column-oriented Databases

Column-oriented databases, contrary to the conventional **RDBMS**, store their data in columns rather than in rows. This provides faster lookups in disk when reading an large set of blocks. Thus, it is a popular data orientation in NoSQL datastores.

[1] http://www.project-voldemort.com/voldemort/

2.1.2.1 HBase

HBase is the distributed NoSQL database that powers the Hadoop stack. It is built on top of HDFS to provide what HDFS cannot, real-time read/write random access to very large datasets (White 2009). HBase is modeled after Google's "BigTable: A Distributed Storage System for Structured Data"[2] in that it is a distributed column-based store with versioning of the cells. In HBase, a **region** is a horizontal partition of a table consisting of a subset of rows. A **region** is defined from the first row to the last row plus an identifier (White 2009). Just like HDFS and the Map-Reduce framework of Hadoop, HBase also follows a master-slave pattern. The slave servers are the **regionservers** and the master coordinates the cluster, which is lightly loaded since it only assigns **regions** to **regionservers**.

2.1.2.2 Cassandra

Cassandra is a distributed, decentralized, fault-tolerant, column-oriented data store, initially developed by Facebook[3]. The main motivation for Cassandra was high write throughput and to treat failures as the norm instead of the exception (Lakshman & Malik 2010). Its data model is a distributed key/value map where the key is an identifier for the row and the value is a highly structured object. Being a NoSQL datastore Cassandra can be queried with the Cassandra Query Language (CQL), an alternative to the traditional SQL interface.

Cassandra is highly scalable. This characteristic is due to the demand of serving users all around the world through a handful of data servers scattered around the globe. To achieve this goal, it partitions its data across a DHT ring containing all nodes, where for each node a random value was assigned to determine its position in the ring. To obtain the data's position in the ring the data's key (the id for the row) is hashed and walks clockwise around the ring, to find the next node that has a larger position than the item's. This node becomes the coordinator node for the key. Although this method of distribution suffers from lack of non-uniformity of the data's load, Cassandra, periodically, analyzes the load information in the ring and distributes it accordingly. Also, the data's localization may be a problem. For example, an item should be in a server in Europe instead of an Asian one, but hashing does not permit it. But, since high write throughput is a necessary requirement, the value assigned to the item may be correspondent to the item's global location.

Regarding partition tolerance and availability, in accordance to the CAP theorem, are particularly important for the cluster. Cassandra manages replication by replicating each data item at N hosts, where N is a predetermined factor of replication, a responsibility of the coordinator. Each node that stores a replica of a certain data item must be aware of it. To achieve the latter,

[2] http://research.google.com/archive/bigtable.html
[3] https://www.facebook.com

Cassandra elects a leader using ZooKeeper (Hunt et al. 2010), and for every node that joins the cluster the leader informs the node what ranges of keys it is responsible for.

In addition to replication, local persistence is achieved in Cassandra using a commit log for write operations on a dedicated disk, to achieve maximum write on disk throughput. Commits to disk generate an index for fast lookup, based on a row key, and read operations go through a bloom filter to summarize the keys in the file, possibly leading to false positives though it is not problematic.

2.1.2.3 Druid

Druid is a scalable, highly available, shared-nothing, column-oriented and real-time analytical data store (Yang et al. 2014) designed as a backend to a SaaS deployment of Metamarkets.[4] A Druid cluster manages "batches" of immutable data, called "segments", resultant of the process of events. It manages the segments using multi-version concurrency control to maintain read operations consistent. The cluster's architecture is organized into nodes, spanning four specific roles, the **real-time nodes**, the **coordinator nodes**, the **historical nodes** and the **broker nodes**, coordinated by a ZooKeeper instance (Hunt et al. 2010).

Druid real-time nodes acquire streaming data, a process called **ingest**, persist the data to a MySQL database and, periodically, handoff batches of immutable data to history nodes, called "segments". The data indexed by the real-time nodes are immediately available for query, however they only concern events during a small window of time. Later, they handoff the data processed during that period of time, the "segment", to the historical nodes.

On the other hand, historical nodes load the batches of immutable segments, created by the real-time nodes, for query. Every time the nodes load or drop data, they announce it on ZooKeeper. These nodes use a local cache for loading the segments before serving to client requests. If a segment is not on the local cache of the node, then the node must download it from the deep storage. Since they depend of ZooKeeper to announce and retrieve the information of the data they must download, if it should fail they can no longer download additional segments located only on the deep storage. Although this constitutes a point of failure, since historical nodes depend only on themselves, and queries come from HTTP requests, they can still serve data available currently on node.

The broker nodes have the role of a router, routing client queries to the real-time or historical nodes. For faster access to segments, the broker nodes use an LRU cache with invalidation protocols. If a queried segment is not present in the cache, the node will forward the query to the correct historical or real-time node, according to the instructions from ZooKeeper. Every time a request is forwarded to an historical node, the results received will be cached on the

[4] https://metamarkets.com/

broker node. That does not happen to real-time nodes, because they serve that request only for a small amount of time.

Coordinator nodes first execute a leader-election process to elect a single node as the coordinator. Then, the coordinator load-balances the historical nodes, decide when they should drop outdated segments and when to replicate. The coordinator node runs periodically, depending on the ZooKeeper instance and the MySQL database to retrieve what segments should be served by the historical nodes, based on sets of rules. Outages from the MySQL database or from the ZooKeeper could break the system, since the coordinator could no longer assign or balance the historical nodes, but in such case, historical, broker and real-time nodes can still serve requests from their caches.

2.1.3 Batch-Data Processing

Big data and real-time web technologies process huge amounts of data at a time. That huge amount of data can be translated into a batch. Therefore, batch processing in Big Data involves the scheduling of a "job" as its workload.

2.1.3.1 Hadoop Map-Reduce

Map-Reduce is an abstract programming model to deal with the computation of large amounts of raw data (Dean & Ghemawat 2008). It takes care of the complexity behind the operations, like load-balancing, fault tolerance and parallelization. The programming model defines two primitives, **map** and **reduce**, available for the users to program. The **map** function has, as arguments, an input pair key/value and generates a set of intermediate key/value pairs. The **reduce** function takes the set of intermediate key/value pairs and generates a "reduced" set, meaning that outputs the intended key/value pair.

The Hadoop project[5] (White 2009), from Apache, consists of several software packages that constitute a reliable, highly available, scalable and distributed computing environment specific for Big Data jobs. It employs the map-reduce programming model in its jobs, in order to deal with the parallelization and load-balancing. The high-level mechanism for the Hadoop map-reduce is based on three main Java classes, the `JobClient`, the `JobTracker` and the `TaskTracker`. Hadoop processes its map-reduce runs as batch jobs, thus making it highly optimized for static data. The Hadoop software packages are written in Java, are open source and can be deployed on commodity hardware.

[5] https://hadoop.apache.org/

2.1.4 Stream Processing

Like batch-data processing, the stream processing can be used to process large amounts of data. However, unlike batch processing where the workload is an amount of data read from a kind of storage like a file system or a database, stream processing manages to process data coping with a real-time guarantee.

2.1.4.1 S4 (Simple Scalable Streaming System)

S4 is an Apache incubator project, initially developed by Yahoo[6] to create a streaming event system. Unlike batch processing where load-balancing is an important factor, S4 processes its streaming data as events. It keeps up with the event rate or slowly degrades it by eliminating events, a process called load-shedding (Neumeyer et al. 2010). S4 defines two basic entities, the **Processing Element** (PE) and the **Processing Node** (PN).

The PE is the basic unit of computation, an object instance that consumes events that correspond to its key and it may output events to other PEs. For a `WordCount` example, keys are the words of the text to be processed(Neumeyer et al. 2010) (consumed as an event).The PE is configured to consume events only for its type. Some PEs are **keyless**, meaning they have no key assigned yet, although they consume all events of the configured type before being assigned a key by the S4 cluster. A big number of PEs may be created during the processing of an event, but S4 lacks the ability to properly load-balance the events and to free the heap space they occupy when they're no longer needed.

The PNs are the containers of PEs, where they are responsible for listening, dispatching events and outputting other events. Thus, they are the nodes in the S4 cluster. Events are routed to the PN based on the hash of the key of the event. This allows the reception of a single event by several PNs, which ZooKeeper will coordinate. However, the key of a PE maps into just one of the PNs. The dispatcher in a PN invokes the PEs in the appropriate order and, in the case that a new PE must be created, a **PE prototype** object will be cloned, resident on the node.

2.1.4.2 Storm

Storm is another streaming processing system, in use by Twitter after acquiring BackType (Toshniwal et al. 2014). Its architecture consists of streams of **tuples** of two types, **sprouts** or **bolts**. Sprouts are much like S4's processing nodes (PN) where bolts are S4's processing elements (PE). However, unlike S4, bolts don't serve a specific type based on a key. Instead, they act like a container for a set of tasks running over an **executor**. Executors are what provides the intra-topology parallelism. Effectively, they are threads belonging to a JVM process in each

[6] http://incubator.apache.org/s4/

worker node. Storm supports a variety of partition configurations, ranging from a fully centralized solution with **global grouping** to a fully random solution with **shuffle grouping**. The whole coordination of the workers is maintained by ZooKeeper (Hunt et al. 2010). Each node runs a **supervisor** that can spawn other worker nodes dynamically. The supervisor decisions are based on the **master** node, called the Nimbus, which is responsible for the coordination of the topology - not of the cluster, since that is the ZooKeeper's job. Therefore, Storm works in a decentralized manner, with different possible configurations for it's topology of sprouts and bolts, with a central node managing their configuration. This ensures no extra load over the Nimbus, but with coordination achieved.

2.1.5 Deployment support and coordination frameworks

Big data applications are deployed over clusters, whether they are built over commodity hardware or over an expensive data center. Although, as Sakr *et. al.* (Sakr et al. 2011) stated, cloud services are increasingly providing better Quality-of-Service (QoS), including management of the scalability of the system (**elasticity**), availability of the resources (**resource pooling**) and high-speed network access. Thus, it is feasible to assume that the management of underlying cluster has importance on the overall experience. The following subsections relate two cheap solutions for cluster management, that can also integrate with each other.

2.1.5.1 ZooKeeper

ZooKeeper, a former Hadoop subproject, but now entirely on its own, is a centralized service for the coordination of processes under a distributed application system. ZooKeeper provides group messaging, a distributed lock service, shared registers, leader-election, linearizable writes and per client FIFO ordering (Hunt et al. 2010). To enable flexibility for application developers, ZooKeeper exposes an API for the implementation of new primitives, enabled by their **coordination kernel**.

ZooKeeper's coordination kernel is the service that enables the custom primitives that implement the various techniques to coordination. Hence, ZooKeeper acts as a library for powerful coordination primitives that can be shared among the community (White 2009). Primitives, like group membership, configuration management, rendezvous and several types of locks and barriers can be easily implemented over the **znodes**. ZNodes are in-memory data nodes in the ZooKeeper's data tree (Hunt et al. 2010). The znodes are created by the clients and follow a hierarchical directory view, thus giving the application (the client) the possibility to manage, independently, its znodes.

The architecture of ZooKeeper is based on three main components, the **Request Processor**, the **Atomic Broadcast** and the **Replicated Database** for high availability. Read requests to the service are immediately directed to the database. On the other hand, a write request

follows the ZooKeeper's pipeline by first going through the request processor, then through the atomic broadcast and finally written to the database. The need for an atomic broadcast is to provide quorum-consensus over the data to be written and total order of the messages sent. The database holds the state of ZooKeeper and its znodes, replicated across its servers.

2.1.5.2 OpenStack

OpenStack[7] provides an open source IaaS (Infrastructure as a service) on the cloud for the deployment of big data applications. As applications we name the before mentioned data stores (HBase, Cassandra and Druid), the job processors (MapReduce and S4), the tools for the coordination (ZooKeeper) and distributed file systems (HDFS). Since we're working at PaaS (Platform as a service) level, we believe that careful tuning of the supporting platform for these applications (the Java VM) may increase their performance.

2.2 Garbage collection

For high-level language (HLL) programs garbage collection brought means for programmers to forget, to a large extent, explicit memory management and continue to have a well performing system. It is common sense to consider explicit management of the memory as better performing compared to garbage collection. In fact, for languages that run on top of process VMs, like Java, it is not trivial to simulate explicit management for comparison. Hertz and Berger introduced an experimental methodology to compare garbage collection algorithms with explicit memory management (Hertz & Berger 2005). They concluded that the best performing garbage collector is competitive with explicit memory management when given the right amount of memory.

The garbage collection research field has seen several years of study and its importance is increasing, on par with the evolution of high-level languages like Java and C#. This has spanned a vast collection of algorithms where those used today are combinations of older algorithms, which can be stacked with application-specific profiles (Singer et al. 2007).

Algorithms for garbage collection are divided by categories. However, more recent algorithms employ hybrid solutions, a mix of different algorithms from different categories. In the following subsections, some relevant algorithms are described. They will be categorized by their parallel and concurrent characteristics. But first, an explanation of the existing **barriers** is given. Barriers are used by some algorithms to track the mutator's read or writes.

[7] http://www.openstack.org/software/

2.2.1 Barriers

In some GC algorithms, the read or writes of the mutator must be recorded. This is to give the garbage collector a chance to take appropriate action before the system becomes unstable. Thus barriers are a way for the collectors to know that a certain memory address is to be handled differently. Barriers can throw OS-level traps or invoke special routines for handling. Implementation of barriers may vary between algorithms, and they can be software-based or hardware-based, although there are two main categories, **read-barriers** and **write-barriers**. Table 2.1 shows, for the algorithms in Sections 2.2.2, 2.2.3 and 2.2.4 what kind of barrier an algorithm uses, based on the understanding of the literature.

2.2.1.1 Read Barrier

Read barrier is that which triggers whenever the mutator loads a reference. Baker used a read-barrier for its incremental collector (Baker Jr 1978) when copying a loaded reference to the to-space, Bacon *et. al.* for their **real-time GC with low overhead** (Bacon et al. 2003) and Azul Systems for their Pauseless and C4 garbage collector algorithms (Click et al. 2005; Tene et al. 2011). Two types of read-barriers were categorized in (Blackburn & Hosking 2004; Yang et al. 2013): **read conditional** and **read unconditional**. A conditional barrier obeys to a certain condition and unconditional barrier masks low-order address bits.

2.2.1.2 Write Barrier

A write barrier is triggered by the mutator when a new reference is stored or a field of an object is updated. There are several write barrier implementations, some targeted specifically for some categories of algorithms. Blackburn and Hosking (Blackburn & Hosking 2004), and their consequent work with Yang *et. al* (Yang et al. 2013), put on a comparison between write barrier implementations. The barrier implementations are categorized as **boundary**, **object**, **hybrid**, **zone** and **card**.

The **boundary** barrier records source addresses when both source and target addresses lie in a different region of an inter-region heap. It is particular to the generational collectors. The **object** barrier remembers objects based on a flag (a bit) in the object header. Hence, the barrier's slow-path only needs to flip the bit to avoid re-remembering the object. The **hybrid** barrier mixes object and boundary barriers, where **boundary** barrier is used for arrays (no need to check the bits of every object in the array) and **object** barrier for everything else. On the other hand, **zone** barrier assumes that the heap is split into aligned zones of 2^k-bytes and XORs the source and target address in order to remember those that cross regions of the heap. Last but not least, the **card** barrier is an unconditional barrier, that also assumes a heap divided into fixed 2^k-byte zones. It maps an entry in the **card table** (that can be implemented from a byte array) if a zone has been **dirtied**.

GC	x86						Dedicated H/W	
	Read	Write					Read	Write
	Conditional	Boundary	Object	Hybrid	Zone	Card		
Inc Copying	X							
Boehm MP			X					
Immix		X	X		X			
AppelRT		X						
Pauseless							X	
C4							X	X
PS						X		

Table 2.1: Barriers used in GC (for those applicable)

2.2.2 Non-parallel algorithms

This work defines the the non-parallel algorithms as the traditional, or classical, collectors because they are the base of the more recent algorithms. These are simply referred to as **Stop-the-World** (STW) algorithms, meaning that the mutator threads must be idle for the garbage collector to take place.

2.2.2.1 Mark-and-Sweep

Mark-and-Sweep was the first garbage collector, devised by John McCarthy in the 60s (McCarthy 1960). The collector marks all the reachable objects starting from the **root-set**. The root-set is composed of objects directly accessible by the mutator, like stack-frames, global variables and registers. After the marking phase, the unmarked objects are garbage and so they are collected ("sweeped"). The sweep phase proceeds through free lists that notify the mutator of the free memory slot. The algorithm is still used today, although tuned to work with the most recent architectures and systems. In its infant form, the mark-and-sweep collector has high overhead for very large heaps.

2.2.2.2 Copying GC

Copying garbage collection was proposed to deal with the overhead of the mark-and-sweep (Baker Jr 1978). Instead of traversing the whole heap, the copying collector divides the heap in two regions, the **to-space** and the **from-space**. This collector is not to be confused with multi-region collectors, because the regions are not used concurrently. It is instead a **semi-space** collector. At the time of collection the to-space is an empty space and the from-space is the current heap. The collector begins traversing the heap, starting from the root-set and copying the records it finds to the to-space. When copying, the collector aligns the objects with the

bump-pointer. The bump-pointer tracks the end of the last allocated object and the beginning of the next. It is incremented using the size of the next object, thus "bumping" its value. When there are no more objects to traverse the two regions are swapped, making the to-space (now with the surviving objects) the new from-space and vice versa. The to-space can then be flushed of all references it contains.

2.2.2.3 Incremental GC

Incremental garbage collection, first proposed by Henry Baker Jr, is based on the idea of spreading its work for a portion of the heap at a time (Baker Jr & Hewitt 1977). Between each phase the mutator program is allowed to execute, therefore providing small **latency** for systems that cannot not cope with long pauses. Incremental collection is commonly associated with a copying collector, that incrementally copies objects to the to-space. Although this algorithm brings various benefits, due to the synchronization features, such as waking up the mutator threads and locking the GC threads, the sum of the incremental phases may be longer than a continuous collection phase. This latency may yield lower throughput. In spite of that trade-off, incremental collection is a low-cost solution for systems that do not require real-time guarantees, e. g., web browsers.

2.2.2.4 Generational GC

Generational garbage collection is based on the *weak generational hypothesis*, that sates that most objects die young. It splits the heap regions turning each space into a generation. Typically, the generations created are two: the **young-space** (sometimes called the **nursery-space**) and the **old-space** (sometimes called the **mature-space**). The nursery is, generally, considerably smaller than the old-space. Each space is collected independently of one another. Allocation takes place on the nursery and the objects that survive a specific number of collections (defined as **age**) are copied to the old-space, a process called **promoting**.

The generational collector was first proposed by Lieberman in 1983 and implemented by Ungar in 1984 (Ungar 1984). Later, Appel raised some problems with the previous implementations, mainly due to the fact that before making an assignment to a field they had to consult a list of root-objects (Appel 1989). He proposed a generational collector, with fast allocation, which is often called Appel-style collector.

In each generation a specific algorithm can be employed and the frequency of collection can also vary. For example, consider that `t1` and `t2` : `t1` $\ll$ `t2` indicates the time elapsed between periodic collections for the nursery and the mature space, respectively. If a Copying GC algorithm operates on the nursery space, it will collect more often than a mark-and-sweep algorithm that operates on the mature space. There can be a indefinite number (limited only by the heap size) of generations. However, as Appel suggested, less regions maximize the size of

the newest space which reduces the chance of copying a young object to the mature space too early (Appel 1989).

The use of generations is one way of avoiding the "allocation wall", a case in which performance is limited by the rate of the new data allocated into main memory, and adding more nodes does not improve performance. Generational collectors can avoid this overhead by setting per-thread nurseries with sizes smaller than the L2 cache, increasing cache hit and reducing accesses to main memory (Zhao et al. 2009).

2.2.3 Concurrent collection

Concurrency is having one or multiple mutator threads running at the same time as the collector threads. Concurrent collection is also tightly coupled with the **real-time guarantee** term, which means that a collector has low latency.

Concurrent collectors try to reduce the pause time overhead of STW collectors. Although long pause times can be minimized with incremental collection, some systems require **minimal latency**, i. e., extremely short interruptions. Concurrent collection is an option, since it makes better use of the multicore architectures of today and can achieve very small interruptions, in the order of a few milliseconds.

The study of concurrent collection was initiated by Djikstra in 1978 (Dijkstra et al. 1978), when they proposed the **on-the-fly garbage collection**. In their initial paper they raised the synchronization and mutual-exclusion problems, and proposed the "tricolor" method. This method consists on coloring the nodes in the object graph such as: **black nodes** are for objects already marked-through (had all of their references marked), **gray nodes** are for marked objects but whose references are still **white nodes**, which are for objects not yet scanned (they are potential garbage). This method is specially useful in SMP (Shared Memory Architectures), although concurrency can also be achieved on uniprocessors — using context-switching while the mutator thread is waiting for external events (Appel et al. 1988).

There are several techniques to achieve concurrency, though all must follow a certain principle for the synchronization between the collector and the mutator. This prevents conflicts and additional overhead. For example, a concurrent copying collector cannot copy pages to the to-space and then have the original page modified after, or a concurrent mark-and-sweep cannot sweep newly written data allocated by the mutator but unknown to the collector.

A commonly used technique is **page-protected accesses** to the objects, whether using read-barriers or write-barriers (see Section 2.2.1). This method allows the collector program to trap and take appropriate action. The algorithms have their own ways of handling traps. An example of handling page-protection exceptions by inserting **forwarding pointers** on the previous memory address. Forwarding pointers are special pointers that indicate the real location of an object after it has been moved, a common case on copying-based concurrent collectors.

Another technique consists of using **virtual dirty bits** to mark pages that have been modified (written to) after they were scanned by the collector. This is similar to "tricolor" marking (Dijkstra et al. 1978). The virtual dirty bits form the basis of the original mostly-concurrent collector, proposed by Boehm *et. al.* (Boehm et al. 1991), where "mostly" means that the mutator suffers a short pause at the beginning — to deal with the root-set since it doesn't go through the write-barrier. They refer that their collector was neither perfectly concurrent nor precise, however it is the base of some recent generational algorithms (Detlefs & Printezis 2000).

2.2.3.1 Mostly-Parallel

Mostly-Parallel garbage collection, by Boehm *et. al.* (Boehm et al. 1991) (Boehm MP), is a **concurrent mark-and-sweep** collector. Their collector starts by clearing all virtual dirty bits, before the marking phase begins concurrently with the mutator. The collector marks all objects reachable from the root set, as with the traditional mark-and-sweep. However, the mutator writes will be reflected on the virtual dirty bits, meaning that those memory pages the mutator touched are dirty. When the collector stops marking, the mutator execution is halted, so that the collector can trace from the marked objects on the dirty pages. This guarantees that no newly written data is collected.

It is not perfectly concurrent, but its innovation is trading complete concurrency with **throughput** by allowing **root locations** — thread stacks, globals and registers — to be written without the write-barrier constraint. And, although it uses a conservative approach (its target was low-level languages like C) it certainly benefits managed languages due to the removal of its conservative counterpart.

2.2.3.2 Real-time concurrent

Real-time concurrent collection, by Appel *et. al.* (Appel et al. 1988) (Appel RT), is a concurrent copying garbage collector, allowing compaction of the copied objects but sacrificing heap space. It makes use of the hardware virtual capabilities, thus allowing to run on stock multiprocessors. In order to achieve concurrency, and avoid conflicts or memory faults, the algorithm uses virtual-memory page protections (a read-barrier, in fact). The steps that the collector takes are just a few more than the classic copying algorithm. One of the main differences is that it flips the two survivor spaces (to-space and from-space) in the first place, instead of in the end. This restricts the mutator to the to-space in its registers and only allocates new data into to-space. Objects in the unscanned area contain both to-space and from-space pointers. Objects in the scanned area contain to-space pointers only.

The collector sets a virtual-memory page protection to the unscanned area, and for every access by the mutator to a protected page a **page-access** trap is triggered. Whenever a trap is triggered, the collector handles it by copying the object, if it was on from-space, and forwarding

the necessary pointers. In the end, the collector unprotects the page and resumes the mutator on the faulting instruction (saved by the **program counter** (PC)).

The advantage of Appel's method is that, for the mutator, the page that it tried to access contained only to-space pointers. Hence, there's no synchronization work by the mutator. The only overhead that may arise is resultant of the speed that the collector can field that trap.

2.2.3.3 Pauseless

The Pauseless garbage collector, by Azul Systems, explores the possibility of read-barriers in the hardware (Click et al. 2005). Just like Appel's, it leverages its efforts on virtual-page protection. Thus, it achieves almost no overhead compared to concurrent GC on modern state-of-the-art JVMs. The approach used was to add a new privilege mode to the **translation-lookaside-buffer** (TLB), called the **GC-mode**, that can generate fast user-level traps. The drawback of this privilege mode is that it requires specific hardware to achieve full compatibility. However, it is possible to emulate it with some costs.

The base of the Pauseless-GC algorithm is a **concurrent compact mark-and-sweep**. It marks live objects, compacts the memory in-between pages (they are of 1 MB of size) and sweeps free the unmarked objects. The mark phase consists of having the mutator and collector threads mark from the root-set (to prevent idle threads, in the Pauseless-GC, the mutators also perform collector work). Since with a concurrent marking phase an object may be newly written after its parent was scanned, a read-barrier is used to generate a **Not-Marked-Through**-trap every time an unmarked reference is loaded by a mutator thread. **Not-Marked-Through** (NMT) is a special bit of the object header, used by the mutator whenever it loads a reference to check if the object has its fields marked through. If not, the NMT-trap handler will correct the fields pointers. After marking, the algorithm starts a **relocate** phase, compacting the objects within memory pages, frequently filling the 1MB pages with objects, and freeing the "empty" pages. In this phase, the read-barrier takes GC-traps when the mutator tries to read a protected page (a page with objects being copied). This is to prevent the mutator to access invalid words. The last phase, the **remap** phase, consists of forwarding pointers for the moved objects and freeing the protected pages (the previous locations of the moved objects).

2.2.3.4 Continuously Concurrent Compacting Collector (C4)

The C4 garbage collector is an improved version of the Pauseless-GC, but uses two generations to collect (Tene et al. 2011). Just like the Pauseless GC algorithm, the C4 uses a read-barrier, this time called the **Load-Value Barrier** (LVB), triggered based on two invariant conditions that assert that a loaded reference refers to a live object and that is correctly placed (there are no mutator-access locks). The LVB is a self-healing barrier in that the mutators themselves store a copy of the loaded reference after it has been "healed" (flipping Not-Marked-Through

bits to the correct one), a mechanism also executed by the Pauseless-GC (Click et al. 2005). The promotion of objects occurs during the Relocate and Remap phases. Since old-generation objects may have references to young-generation, a write barrier is used whenever references from the young-generation are stored on the old-generation, called the **Store-Value Barrier** (STB). This way the LVB never traps on objects marked by the STB, filtering out objects from the young-generation in the process of being promoted.

2.2.4 Parallel algorithms

The parallel algorithms are those that execute with GC threads in parallel during garbage collection, but that require stoppage of the mutator threads. The term "parallelism" is not to be confused with concurrency, which is addressed in Section 2.2.3.

Modern architectures ship with big heaps, therefore a single GC thread is bound to generate high overhead. A likely scenario for STW and incremental collectors would be long pause times, and for a concurrent collector would be a growth of garbage objects, meaning that the collector would be **late** over the collection period. The importance of the research of parallel collection increased, throughout the time, with the need to adapt collectors to multiprocessors. Ever since, several studies have been done to give parallel capabilities to the already existing algorithms, focusing on **Shared Memory Processors** (SMP) and **Non-Uniform Memory Access** (NUMA) architectures.

Of the studies regarding parallel capabilities over existing algorithms, some issues were raised. One of the issues, deeply related to multiprocessor architectures, is the **scalability** of the collectors. The scalability is the level of how much inversely proportional is the stop-the-world pause time in respect to the number of cores. One of the reasons the **scalability** may not be optimal is the poor load-balancing of the collection work, as noted in (Cheng & Blelloch 2001); for their algorithm, a `shared stack`, which serves as a hub of **gray objects** (from the "tricolor" marking scheme, presented in Section 2.2.3), is loaded by the `local stack` of each node. Other issues raised by the stop-the-world parallel algorithms are the lack of NUMA-awareness and the use of a lock for the parallel phase of the collection (Gidra et al. 2013). The lack of NUMA-awareness, as with scalability, is also related to poor load-balancing of the nodes — the collectors can, sometimes, overload the one node where most allocation takes place. On the other hand, the heavily contented lock is used to synchronize the access to objects by the collector threads. Mechanisms to deal with the lack of NUMA-awareness have been proposed (Gidra et al. 2013; Ogasawara 2009) and they will be explained in the following algorithms.

2.2.4.1 Immix

Immix is a space efficient mark-region garbage collector with contiguous allocation and fast collection, by Blackburn and McKinley (Blackburn & McKinley 2008). It deals with the inherent fragmentation, commonly resultant of non-moving collectors, by using **opportunistic evacuation**, that is, Immix only triggers evacuation if the target space has not been exhausted due to a previous evacuation or if the application has not pinned the object. It has a particularity which is is easy adaptation to different configurations of the heap. Thus it has the possibility of using various types of write-barrier.

Immix divides the heap into blocks and each block into fine-grained lines. Blocks act like virtual-memory pages and an object cannot span multiple blocks, though it can span multiple lines within a block. Thread-local allocators (TLA) request blocks and bump-allocates objects through the lines. Blocks can be requested as long as they are available, otherwise a collection is triggered.

The collection in Immix is very similar to mark-and-sweep, in the sense that it traces the live objects from the root-set and later sweeps the garbage objects. But, Immix's **recycling policy** defines three states for a block: **free**, **recyclable** or **unavailable**. A recyclable block is one that has at least one free line. The difference is that after sweeping the garbage objects, Immix returns entirely free blocks to a global pool and recyclable objects for the next allocation phase. The free blocks in the global pool will be the target of the **demand driven overflow allocator**, that allocates **medium** to **large** objects.

To defragment the heap, Immix triggers an opportunistic evacuation whenever it finds a candidate object. It evacuates the object to a new target block, giving priority to recyclable blocks, and leaves a forwarding pointer behind. The heuristics to find a candidate object are computed through histograms, that estimate the required space and the available space. Also, Immix can defragment in parallel using a **compare-and-swap** (CAS) approach to deal with the race of accesses.

Since Immix uses TLAs, that request their addressing space (blocks) from the global allocator and must be synchronized, the collector does not need any synchronization between threads during allocation. Also, the marking of the lines is done through the objects metadata, reducing the need for synchronization.

2.2.4.2 Parallel Scavenge

The Parallel Scavenge (PS) algorithm defines a parallel STW garbage collector, with generational, copying and compacting capabilities, employed on the HotSpot JVM[8]. It has no official information, but in (Gidra et al. 2013) its internals are described in detail. The Parallel Scavenge

[8]Information at: http://www.cs.princeton.edu/picasso/mats/HotspotOverview.pdf

collector separates the heap into two spaces, where each of these regions allocate the young and the old objects, respectively. A parallel copying collector operates on the heap collecting the young space. As for the old space, a two-phase mark-compact algorithm is used.

The young space is divided in three sub-spaces, the **eden** space and two **survivor** spaces, the **to-space** and the **from-space**. If the objects in each of these regions survive a certain number of collections, then they are promoted to the next region. For allocation in the eden space, the mutator allocates, for its own use, a memory chunk called the Thread-Local Allocation Buffer (TLAB). A collection is triggered when there's no space left on the current TLAB, and the young space cannot allocate more TLABs. Promoted objects, whether to the to-space or to the old space, are allocated from Promotion-Local Allocation Buffers (PLAB).

The Parallel Scavenge uses two monitor pairs, one to synchronize the mutator with the VM thread and another to synchronize the VM thread with the GC threads. The VM thread is responsible for the queue of GC tasks. The tasks include the **root-tasks**, the **Old-To-Young root tasks** and the **steal-tasks**. The GC threads pop tasks from the queue and traverse the object graph in a breadth-first search. For each object found in the object graph, they create a copy in the PLAB and install a forwarding pointer in the old object's header. To "trace-through" the object, the GC thread pushes all the references of the object fields to a reference queue, pops a reference and forwards, repeating the process. Idle GC threads steal tasks from the task queues belonging to the other non-idle threads. After all tasks have been processed, the VM thread stops the GC threads and wakes up the mutator threads.

2.2.4.3 NUMA-Aware Parallel Scavenge

NUMA-Aware Parallel Scavenge (NAPS) is the resulting collector from the study of Parallel Scavenge in (Gidra et al. 2013) to get NUMA capabilities. They identified the lack of NUMA-awareness and the use of the heavily contended lock, as the prime suspects for the overhead of Parallel Scavenge when employed over large NUMA heaps. For the contended lock (the monitors that give access to the task queue) they simply replaced it with a lock-free task-queue. Therefore, the GC threads pop tasks using a **compare-and-swap** (CAS) operation. On the other hand, to deal with the lack of NUMA-awareness, they proposed three heap layouts that leverages memory accesses and memory placement policies. The **interleaved spaces** policy maps pages from different nodes in a round-robin fashion, balancing memory accesses. The **fragmented spaces** policy divide the heap into fragments and assigns each fragment to a single node, increasing locality and reducing memory access imbalance. Finally, the **segregated space** policy sets node-local collection preventing work-stealing from other GC threads, improving locality and access, but confined to never move objects to rebalance the nodes.

2.2.4.4 NUMA-Aware Dominant-thread-based copying

The copying garbage collector using dominant-thread in a NUMA heap (NUMA-Aware DTB) is a cache-coherent NUMA (cc-NUMA) aware that executes a copying algorithm based on an exclusive thread. It was proposed in (Ogasawara 2009), given the desperate need for NUMA-aware collectors in servers that execute multi-threaded automatic memory management programs, written in object-oriented languages. The exclusive thread is the **dominant thread** (DoT), the thread that most often accesses an object. The approach is to use the DoT to heuristically decide what is the object's preferred location for copying. The location is the destination node that contains the DoT thread, and the surviving object is allocated (by the NUMA-aware allocator) on the **survivor buffer**.

In (Ogasawara 2009) it is explained that on NUMA-unaware JVMs, the tendency is having threads allocating objects on remote locations, due to blocks being obtained by the same CPU and the high page size. Therefore, the heap layout must be aware of the NUMA architecture, managing the objects in a distributed form, and the allocator must know that, although the object's block was obtained from another CPU, the thread that initializes the object allocates in the CPU where it is running. Also, the GC thread, whenever garbage collection is triggered, must find the preferred CPU, the one that contains the DoT thread, to set the destination of the copying procedure. Ogasawara's algorithm does so by using scanning the thread stack, the owned objects and the parent objects.

2.3 Object Layout and Locality

The Java virtual machine has several abstractions to deal with the complex mechanisms of managing heap. Those include implicit pointers (everything is a reference), garbage collection algorithms, wrappers of objects in collections and simple object creation. However, these abstractions may not be ideal on a memory hungry systems, where memory footprint must not spiral out of control. This section shows some "behind-the-scenes" structures that the JVM keeps in order to fulfill its manager responsibility, and work in the literature that studied various levels of optimization.

2.3.1 Java Memory Layout.

The JVM runs an user process with its own heap and a native heap, where part of the heap is a subset of the native heap (Bailey 2012). The native heap allocates GC specific memory for keeping objects information updated.

Java Big Data applications scale the heap to enormous sizes, i. e., easily greater than 16GB. This results in high memory footprint and high overhead ratio of object to type. To quantify,

take an IBM 64-bit JVM where a `java.lang.Integer` object that encapsulates an `int` primitive type is 28 bytes. This produces an overhead ratio of 7:1 (Bailey 2012). The IBM JVM uses the **flags** and **locks** words in separate — these words save object's information to help the VM manage the object — while on the other hand, the HotSpot JVM packs those in one heap word. This allows the HotSpot JVM to save a `java.lang.Integer` object with 20 bytes. However, object sizes must be aligned to the machine word size, turning 20 bytes into 24 bytes. This produces an overhead ratio of 6:1.

Compressed OOPs (ordinary object pointers) and compressed class pointers can reduce the overhead ratio, by storing the object's metadata in 32 bit words. However, such an approach is limited to 32GB of heap [9], which can be impractical in some Big Data systems.

In big data applications large collections are their favorite method for keeping in-memory data. For example, in Hadoop source code [10] it can be seen that the several nodes (in HDFS), or the several jobs (in its MapReduce framework), depend heavily on `HashMap` collections. Collections, however, may incur additional overhead over fragmentation when they become too large, and optimal distribution is not achieved when there are, approximately, 10K entries. That demands for higher pauses for the GC, in a Stop-The-Word scenario, or uneven "allocation race" for a concurrent GC.

2.3.2 Locality Optimization Schemes

The ordering schemes of Java objects were studied in (Ilham & Murakami 2011), to evaluate if careful optimization can achieve less cache misses or DTLB misses. Their first step was to experiment using an extended copying garbage collector, that orders the parent and child objects in the to-space, in a *Depth-First* (DF) or in a *Breath-First* (BF) scheme. The results showed that the DF scheme compared to BF scheme reduces L1 cache misses by 1.5%-17%, L2 cache misses by 6%-21% and the number of DTLB misses by 9%-21%. Their second experiment was to modify the DF scheme such as that hot-fields (child objects), fields frequently accessed within the object during runtime, are placed close to the parent after copying. This enables the parent and the hot child object to reside at the same cache line, thus improving spatial locality. The final results showed that the Hot-DF (HDF) scheme reduces L1 and L2 cache misses by 4%-11% over the DF scheme, and DTLB misses are also reduced by 5%-12%. They conclude that the reduction of cache and TLB misses is a major influence for the total performance, generally achieving considerable speedup. Although the study is 5 years old, the reality is still the same for high-level languages like Java.

While the study in (Ilham & Murakami 2011) showed optimization at the cache level, it does not include locality optimizations for potentially large objects found on large scale platforms. In (Moon 1984; Wilson et al. 1991) optimization was achieved by separating ephemeral

[9] https://wikis.oracle.com/display/HotSpotInternals/CompressedOops
[10] http://svn.apache.org/viewvc/hadoop/common/trunk/

objects of those that are for general-purpose and can be collected in an early stage. Their optimization leverages special objects to separate rooms in memory to reduce the bloat that they cause when mixed with other objects, and to reduce the traversing of the object graph during GC. On the other hand, in (Chen et al. 2006) they instrument object code in Microsoft's Common Language Runtime[11] (CLR) system to create profiles of object usage in order to achieve better cache locality using optimal object placement in memory.

Summary

This chapter presented the available mechanisms and platforms, available in the literature, that inspired the design of the architecture now presented in Chapter 3. It covered the Big Data platforms that can be targeted for improvement, the available garbage collection algorithms and techniques that can be extended to fulfill this work's solution, and a summary of how the object memory in the Java heap is designed and what optimizations are available. A summary of the algorithm categorization is illustrated in Table 2.2.

[11] https://msdn.microsoft.com/en-us/library/8bs2ecf4(v=vs.110).aspx

Table 2.2: Summary of GC Algorithm capabilities

GC	Non-Parallel	Concurrent	Parallel	Is Generational?
Mark&Sweep	X			No
Copying GC	X			No
Incremental Copying	X			No
Generational GC	X			Yes
Boehm MP		X		No
Appel RT		X		No
Pauseless		X		Yes
C4		X		Yes
Immix			X	No
PS			X	Yes
NAPS			X	Yes
NUMA-Aware DTB			X	No

3 Architecture

> Put your heart, mind, and soul into even your smallest acts. This is the secret of success. — *Sivananda Saraswati (1887–1963)*

In Chapter 1 it was said how the JVM is unaware of the memory bloat that a Big Data application can bring onto the Java heap. Splitting the spaces to make room for certain types of objects that have tendency to outlive most objects, and whose references require a great deal of loads is a possible solution. This chapter aims to describe the main architecture for the proposed solution, taking into account the design goals of the HotSpot virtual machine. The HotSpot virtual machine is a complex project and highly optimized for each platform it supports. Therefore, a "birds eye view" description of some of the modules is given, but the focus goes to the GC module, in addition to some other components that had to be extended.

There are several modules in the HotSpot virtual machine ranging from CPU support, OS support, compiler code, reference code, garbage collector code, etc. Also, there are three throughput oriented collectors currently implemented: the Parallel Scavenge collector, the Garbage First collector and the Concurrent Mark&Sweep collector. The efforts in this thesis work are applied to the Parallel Scavenge collector, which is the default for server-class machines. This does not mean that the solution is restrictive since, in fact, can be easily adapted to other collector implementations.

The following sections are organized following an approach of increasing detail. First, in Section 3.1 an overview of the HotSpot virtual machine is presented, with the key relevant modules highlighted. In Section 3.2 the target GC module is described, at high-level. The GC module is the parallel scavenge collector, as referred above. The last Section 3.3 will describe, this time with more detail, the decisions and the approach taken.

3.1 HotSpot Architecture Overview

The HotSpot virtual machine is composed of several components, such as platform dependent code (OS and CPU support), the JIT compilers, the runtime bytecode interpreter, the garbage collectors and the memory management performance engine; the latter closely tied to the garbage collector in use. Due to its cross-platform capabilities, the HotSpot VM has several components connected by interfaces, such as heap managing classes, bytecode interpreters (such as the template assembly interpreter and the C++ interpreter) and two of the

current bytecode compilers: the server-class compiler and the client-class compiler. This work is solely based on the heap managing classes.

An overview of the HotSpot is now described raising the importance of some modules. The modules, in particular the Parallel Scavenge GC module is the are the direct intervenient on this architecture's solution. The Parallel Scavenge code includes classes for GC tasks, collection managers, allocation buffers and the heap itself. Therefore, many of its components required extensive modification or had their current implementation extended given their impact on the outcome of this solution. As for the rest of the classes, like `Klass` and `oopDesc`, who are determinant for the correct functioning of the VM, they also had their implementation extended.

Before continuing, some terms need to be defined. A `Klass` is the internal representation of a Java class object. It contains pointers to its super classes, subclasses and information about its size. An `oopDesc` describes an **oop** (from "ordinary object pointer") which is the reference type used throughout the VM, implemented as a native machine address.

The HotSpot virtual machine commences its execution by starting the OS module and parsing the arguments passed to the launcher. It then initializes the thread local storage module (TLS) and the virtual machine global structures, such as basic types and monitors. In the next step it is ready to boot all the virtual machine globals, namely, the bytecode definitions, the class loader using the command line arguments, the string table, the symbol table, the known type array class objects and the heap. Also, it is ready for the creation of the main Java thread and the VM thread and to set their runtime stack. With the threads in a ready state and all the basic types set, the known object klass types can be initialized. In the next lines, a brief description of the initialization of some modules is given.

The OS module initialization is done in two separate phases, interleaved with other modules initialization. The separation is needed because of the dependency of some initialization code like argument parsing. The first invocation of the OS module initialization sets the virtual memory page size, the number of stack yellow and red pages and the clock. The second one sets the stack size for the Java threads and, using the parsed arguments, it sets the thread priority policy.

The monitors play a vital role during the VM execution. They protect critical areas of code and are indispensable for multi-threading GC. On the other hand, mutexes help serialize the code in critical sections of VM. To enforce consistency between monitors and mutexes, the HotSpot defines a set of monitor and mutex pointers specific for each purpose, initialized at the VM startup and pushed into an array. The current implementation, as of JDK8, defines mutexes as a monitor with restrictions on the `notify` and `wait` methods.

All other modules would be useless if there was no thread code to execute. Thus, the thread classes define an interface for native thread support and related operations. In fact, each thread in the virtual machine must have OS specific information for frame and stack tracking

and suspend/resume support. The thread objects also contain handler lists, i. e., wrappers for references, object monitors and thread local allocation buffer support (TLAB). Then, several types of threads implement specific behavior such as, Java thread, compiler thread, VM thread and GC thread.

Left for last is the heap initialization, which is one of the main subjects of this work. With it, the TLAB has its maximum size set, since it depends on the overall size of the heap. The heap is abstracted as a "collected heap", defined on the top-level of the GC-related interface modules. Therefore, the specific implementation of the heap, which is dependent of the GC algorithm in use, executes any virtual calls directed to the heap. The heap manages all virtual space, committed and reserved, and maintains statistical information of the heap's usage, in the case of adaptive resizing.

At runtime, the HotSpot virtual machine executes a never ending loop for the VM thread. This loop is responsible for executing, within safepoints, VM operations pushed onto the operation queue. VM operations can be pushed onto the queue by the other threads, e. g. Java threads or Worker threads, when these need the give control to the VM thread.

3.2 Garbage collection — Parallel Scavenge

The Java virtual machine specification states that any implementation requires a garbage collector. In the HotSpot virtual machine there are three main options for a collector: garbage first GC, concurrent mark-and-sweep and Parallel Scavenge; respectively referred to as G1, CMS and PS. Among the three, the Parallel Scavenge garbage collector is the default server-mode collector used by the HotSpot virtual machine, as of OpenJDK 8. It is the only currently implemented collector that has no official information available. In fact, both the papers for the G1 and the CMS collectors were written by David Detlefs and Tony Printezis (Detlefs & Printezis 2000; Detlefs et al. 2004) at Sun Microsystems. However, Lokesh Gidra *et. al.* studied the scalability of the Parallel Scavenge collector and described its internals (Gidra et al. 2013), thus serving as a the solid base for this thesis work.

As referred in the previous section, the collector in use is what defines the heap layout. This is due to the collector's implementation definition of how the heap should be structured in order to maximize its throughput when collecting. In fact, the current implementation includes two heaps and one of those is split into two more subtypes. Those two heaps are the `SharedHeap` and the `ParallelScavengeHeap`. Both must follow the `CollectedHeap`'s directives, since it is an abstract class for a Java heap. The efforts in this work are set over the `ParallelScavengeHeap`, illustrated in Figure 3.1, and it is the starting point for the devised solution.

Parallel Scavenge is a throughput-oriented garbage collector and is the default collector of the HotSpot Java Virtual Machine for server-class machines. It is a Stop-the-World, gener-

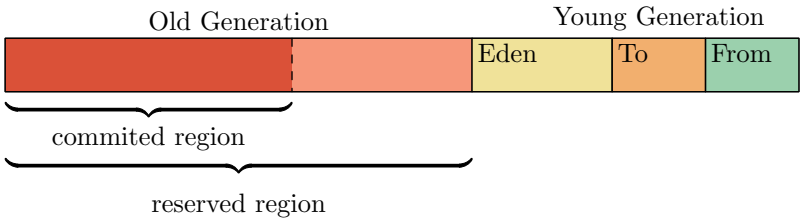


Figure 3.1: Parallel Scavenge heap layout

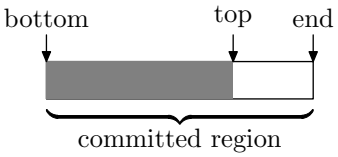


Figure 3.2: A space's layout

ational parallel collector. The Parallel Scavenge heap is separated into two generations, the **young generation** and the **old generation** (the **permanent generation** no longer exists as of OpenJDK 8 following JSR122 [1]). The young generation contains the **eden-space** for recently-allocated objects and two **survivor spaces**, the **from-space** and the **to-space**. On the other hand, the **old generation** contains mature objects, i. e., those that survived a certain number of collections. Globals, such as class definitions, are contained in the native heap (a subset of the heap allocated in the C-heap space of the JVM) and the class static variables and strings are in the Java heap. Each generation operates in its committed memory region, within their reserved space.

Figure 3.1 illustrates how the Parallel Scavenge heap is organized. The allocation scheme uses a **bump-pointer** technique keeping track of a pointer to the start of the next allocated address. This pointer is called *top-pointer*. Allocation spaces, such as those belonging to the young and old generations, base their allocation on a `MutableSpace` object that handles the atomic update of the top-pointer and sets the allocation boundaries. A `MutableSpace` layout is illustrated in Figure 3.2, with its limits represented — the *bottom* and *end* pointers.

The allocation of objects takes place on the eden-space, directly or, preferably, through TLABs. It allocates directly only when the TLAB has no space left for that object and it is not worth discarding the TLAB to allocate a new one. This happens when there is too much free space on the TLAB according to the waste limit threshold. If there's no space left at all then a **young collection** must take place. A **young collection** only collects the young generation whereas a **full collection** takes place in both the young generation and the old generation.

[1] http://openjdk.java.net/jeps/122

3.2.1 Young collection

A young collection operates only on the young generation. It uses a parallel copying algorithm to promote surviving objects from both the eden-space and the from-space to the to-space (the to-space is initially empty). The copying algorithm is, hereafter, referred to as a *scavenge*. A scavenge of the young generation promotes live objects allocating from Promotion-Local Allocation Buffers (PLAB), the same way that recently-allocated objects are allocated from the TLAB. There are two kinds of PLAB, the young PLAB allocated in the to-space fragment of the young generation and the old generation PLAB. Both PLABs have a fixed size, set at the VM startup.

The VM thread first prepares the collection. It enables the reference processor, saves the top-pointers for the to-space and old-space, creates the GC task queue, enqueues the required tasks, sets the active workers and then releases the lock on the monitor where the workers wait. When released, the workers consumed fetched tasks from the GC task queue. The tasks enqueued are to steal tasks and to scavenge the root-set from three sources: the VM global variables, the old objects that reference young objects (*Old-To-Young roots*), and references at the mutator threads stacks (including registers).

To find the Old-To-Young root-set, the old generation contains a card table write-barrier set, which divides the space into fixed-sized *cards* of 2^9 words. It then dirties cards similar to Urs Hölzle's method (Hölzle 1993). Figure 3.3 illustrates the following of dirty cards to mark objects in the young spaces in order to keep them alive.

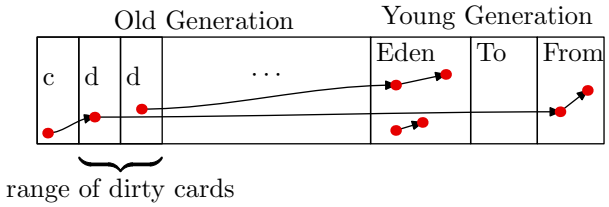


Figure 3.3: Young collection marking

The steal tasks are to prevent idle GC threads. These make the consuming thread to search for, and dequeue, tasks in overloaded queues, belonging to other GC threads, using a random value. If it is unsuccessful in stealing a task then it enters a **termination protocol**. The termination protocol works by atomically increasing a global counter to indicate its termination. If all threads are terminating, i. e., the counter is equal to the number of threads, then the parallel scavenging phase is finished. However, if the thread, after peeking the task queues of the other threads finds that there is still work to do, then it atomically decrements the global counter (indicating it is no longer terminating) and tries to steal a random task once again.

All GC threads have a promotion manager object assigned, which they use to drain the queue of claimed tasks and to maintain the current allocated PLABs. During scavenging on a

young collection, the live objects are, preferably, allocated in the young PLAB. If the PLAB has no space left for that object then it is flushed and a new one is allocated. During flushing, the space left empty is filled with an unreachable filler-object. Generally, it leaves the space slightly fragmented, but allows equal division of work between scavenging threads. If an object is old enough, according to the tenuring threshold, i. e., it survived a certain number of collections, then it is promoted to the old generation. However, some objects may be promoted immediately when two situations arise; the wrapper that iterates over the references is set to do so, such as those that iterate over the code cache of native methods; or the to-space has no space left for the allocation of more PLABs. Figure 3.4, illustrates the allocation of four PLAB$_n$, with $n = 1, 2, 3, 4$, where each belongs to the GC Thread$_n$. In the figure, the live objects in the from-space are old enough or are being processed by the GC Thread$_4$, which allocated its PLAB in the old space. The rest of the objects are promoted onto the to-space, where the threads that are processing them allocated their young PLAB. This distinction is needed since each thread can keep two PLAB, an old generation PLAB and an young generation PLAB.

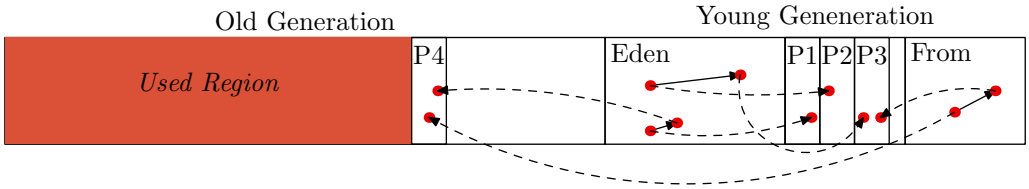


Figure 3.4: Young collection promotion

At the end of a young collection, both the eden-space and the from-space are empty. The to-space and the from-space are then swapped, so that from-space now contains the surviving objects and the to-space is empty for the next collection. The GC threads also flush their current PLAB and terminate.

3.2.2 Full collection

If a young collection was unsuccessful in promoting all its objects then a full collection must take place. There are two available methods for collection, the *Parallel Mark-Sweep-Compact* collector (defined as `ParallelCompact`) and the *Serial Mark-Sweep-Compact* collector (defined as `MarkSweep`). The latter has no parallel capabilities, therefore the *Parallel Mark-Sweep-Compact*, hereafter simply referred to as *Parallel Compact*, is the default for any multicore machine.

Parallel Compact is a throughput-oriented Stop-The-World two-phase *mark-sweep-compact* collector. Although it is a two-phase (mark and compact), in practice its work is divided in four phases: marking phase (Section 3.2.2.1), summary phase (Section 3.2.2.2), compacting phase (Section 3.2.2.3) and cleanup phase (Section 3.2.2.4). The marking phase and the compacting phase are executed in parallel, whereas the summary phase and clean up phase are executed by the VM thread.

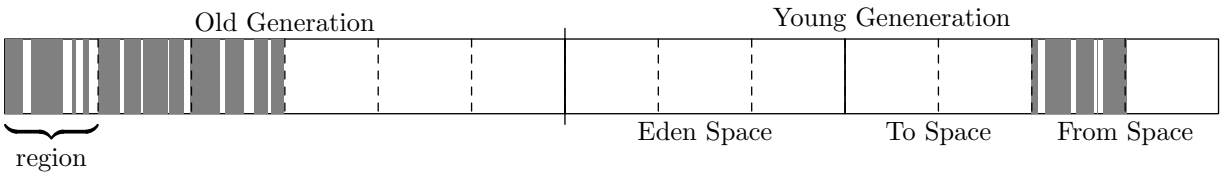


Figure 3.5: Full collection — overview of regions and live objects

A collection using the Parallel Compact collector goes through dividing the heap in fixed-sized regions of 64K words (a total of 512kB in the 64-bit VM and 256kB in the 32-bit VM). This window is its base unit of work. A region contains the amount of live data it contains, in words, and supports holding data for a partial object. A partial object is one that starts in a previous region and crosses into a next region. This is illustrated in Figure 3.5, where the old-space and the from-space contain objects, and the eden-space and the to-space are empty given that they suffered a previous *scavenge* (Section 3.2.1). Some objects cross the region boundaries, these are defined, for the collector, as *partial objects*. Each region contains smaller fixed-sized chunks called blocks, of 128 words, to help the search throughout the marking bitmap. The bitmap is an array of bit offsets, implemented with unsigned integer words. There are two mark bitmaps, one for the objects' begin word and another for their end word. A bit set in both bitmaps marks that object as live. If both bits do not correspond then there is a concurrency problem, i. e., another thread marked invalid bits.

3.2.2.1 Marking phase

The first phase, as the name suggests, comprehends marking the live objects in parallel. Just like the young collection, the marking is executed using tasks pushed onto a task queue. The kind of tasks are also similar; marking tasks and steal tasks. Contrary to the young collection, marking tasks only include the root-set. The tasks are popped and consumed from the task queue using the same mechanism of the young collection. The termination protocol is also the same. Nevertheless, the tasks are consumed differently. Whereas in the young collection the scavenging tasks promote objects, in the full collection the marking tasks mark the root objects' start and ending in the corresponding bitmaps, add its live data amount to the regions containing the start and end addresses — and those in between in the case of a very large object — and push the reference onto the marking stack, a queue backed by an overflow stack. After scanning the root set, the GC threads pop references from the marking stack and begin to follow their contents (fields and the class holder). Each reference found is subjected to the same operation of the root references, creating a breadth-first search.

3.2.2.2 Summary phase

When marking is finished and all GC threads are stopped, the VM thread can regain control and start the summary phase. The summary phase concept is to calculate the destination of the compacting regions. A region is the base unit of work for a compacting task. It is a window that can hold 64K words total. In practice, it is just a wrapper that manages a set of words. Every region contains a destination address and a source region. The destination address defines the exact place in the heap to where the live data will be copied to. On the contrary, the source region value indicates which region will copy data to its location. This creates a kind of chained list between regions.

Compacting is done on a region basis, so all generations and spaces must be aligned to the region size to avoid overflows. Also, after marking, every region should have the live data amount set for objects that end in the region and, if it is the case, the amount of data that crosses into the region boundary — the case for partial objects. At the end of the summary phase, every region with live data should have its destination field set. For each region, the destination field should correspond to the bottom address of the target space plus the sum of the live object size of all previous regions targeted for the same space.

The summary phase is divided into three parts: quick compacting summary of each space into itself, summary of the old generation and summary of the young generation. The first part is to calculate the amount of live data by calculating the new top pointer for each space. This gives an idea of how much space there is live to check if all fits in the old generation. If it does not then the collector must try to compact the maximum possible. Maximum compaction has implications in the computation of the *dense-prefix*, an interval of regions whose objects do not move. There is the *dense-prefix-end* which marks the start of the first compacted region, i. e., all space to the left is full. The *dense-prefix-end* address is always on a region boundary, such as illustrated in Figure 3.6. The second part is to summarize the old space, that for the Parallel Scavenge's heap is the whole old generation. The summary of the old space goes by calculating the dense-prefix and, if not using maximum compaction, summarize the rest of the regions starting from the dense-prefix — the regions to the left are useless to process again. The last summary step is to summarize all young generation spaces. It tries to move the young spaces regions to the old space, while there is enough free space. If there is none then the remaining regions are compacted into themselves, as shown in Figure 3.7 where the from-space is compacted into itself since there's no more space left in the old space.

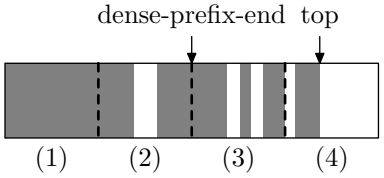


Figure 3.6: Summary phase — Dense-prefix-end and current top pointer

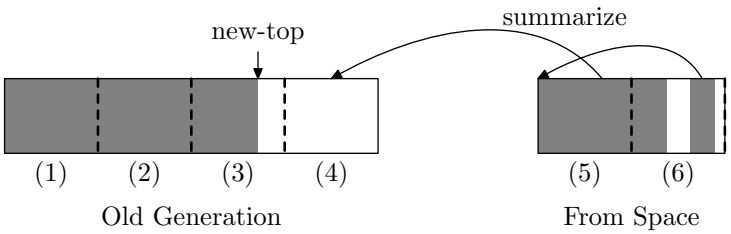


Figure 3.7: Summary phase — Summarizing the From-space

A summary of spaces consists on iterating over all the source regions to set their destination address and the respective *destination-count*. The destination-count is a value that indicates the number of regions that the region under processing will span when copied. In Figure 3.7, region(5) will have its destination-count set to two (2), because when slid after the *new-top* pointer of the old space it will span both regions (3) and (4). It is important to note the two cases for a *new-top* pointer: a new top-pointer after the summary of a compaction into itself, and a new top-pointer after the summary of the other spaces into the old-space. A single iteration starts by setting the destination address which is equal to the target begin address and the sum of the live data for all previous processed regions — the first region's destination is always the target begin address, which in the figure is the *new-top* pointer after the summary of compaction into itself. Since all regions contain only live data, the compaction is guaranteed. After setting the destination address, the destination count value is computed. Its computation is based on two region indexes, one that contains the destination's address and the other that will contain the address of the last live word within the region. If the two indexes are different, then the region's live words will span two regions. The destination-count value also reflects when a region is copied elsewhere, e. g., regions from young to old spaces. Therefore, a destination count cannot be larger than two (2). The last step is to set the source field for the destination regions — the indexes computed in the previous step — equal to the current region under processing. When the iteration is over the target's next top pointer must be updated to reflect the data that will be copied, Figure 3.9.

3.2.2.3 Compact phase

Compaction is done in parallel, such as in the marking phase. The VM thread first enqueues tasks for the GC threads namely drain tasks, dense-prefix tasks and region stealing tasks. Drain tasks pop regions assigned to the local manager's stack, first trying to pop the overflow stack, and only after depletion, a try of the local stack is attempted. Since there was no pushing of the region pointers onto the managers stacks, then the VM thread takes care of it by distributing, in a round-robin fashion, all regions with work to do from the dense-prefix-end until the new top pointer. A region that contains work is one that can be *claimed*, i. e., it has no destination (destination-count is zero). For the regions below the dense-prefix-end, the VM thread also distributes, evenly in region amount, the dense-prefix tasks. Region stealing tasks behave

similarly to the previous stealing tasks and initiate the same termination protocol.

Dense-prefix tasks iterate throughout their assigned region range, find dead blocks (only happens if maximum compaction was not triggered), fill the dead blocks with filler objects and, for live objects, assign their reference to the old generation's object start array. There are no dense-prefix tasks for young generation regions, because the young spaces are only subject for compaction onto themselves if there is no space left on the old generation.

On the other hand, draining region tasks fill summarized regions those of which can be targeted to other spaces. The filling of a region consists on series of steps in order to completely fill regions, or the remaining until the new top pointer. The algorithm goes by getting the source region first word targeted for the region; iterating from the source word to the end of the region, copying objects while they fit; and if the source region was fully processed (it contains no more words to copy), and there's still room in the target region, then the next source region can be fetched, switching source spaces if necessary. When iterating, the source words for an object may not fit entirely. If such happens, the remaining words are copied and the region sets its *deferred-object-address* field indicating an object that starts in the region but does not end in the region. The rest will be processed in the end of the compact phase, when deferred objects are updated. Every time a source region is fully processed its destination count is decremented and, if it can be claimed, it is added to the queue to be filled. This procedure is illustrated in Figure 3.8. The *FillTask 1*, to fill region (3), gets the source from region (5) (previously assigned during summary) and copies until it is full. Then it decrements the *destination-count* for the region (5) and terminates. The *destination-count* of region (5) was initially two (2) because in the end it will be copied over two regions, (3) and (4). *FillTask 2* fills region (4) and gets the source from the remaining words in region (5). In the end, since there was no space left in the old space for more regions, *FillTask 3* fills region (5) getting the source from region (6). Regions (1) and (2) were already filled.

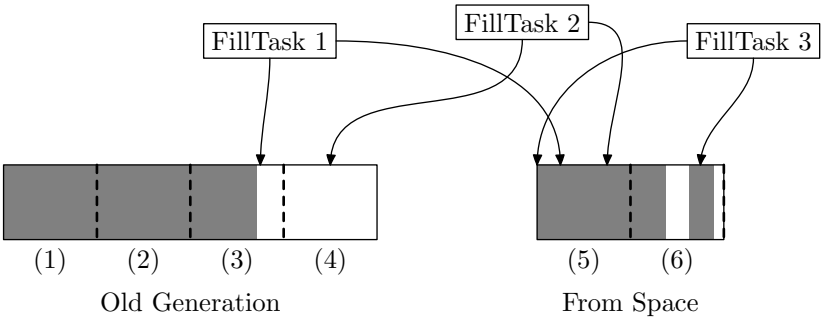


Figure 3.8: Compact phase — Filling regions on the old space

The last step for the compact phase is to update the deferred objects for all spaces. The reason for not doing it earlier was that there may exist object references inside the moved object that could not be updated, due to crossing the region boundary. After compaction, the effective result looks like as shown in Figure 3.9.

3.2.2.4 Cleanup phase

The last phase for the Parallel Compact collector is to set the new top pointers, delegating the call to the spaces themselves, clean the summary and marking bitmap data, and clean or invalidate (set to dirty) the card table barrier set depending whether the young generation is empty or not, respectively.

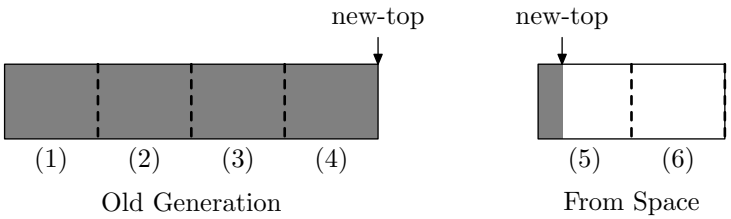


Figure 3.9: Final view of the summary phase and final effective result after compaction

3.3 From Parallel Scavenge to Bloat-Aware PS

Big Data applications have tendency to cause bloat on the heap, since they are data-intensive with huge memory footprint. Given its community support and ease of the development cycle, Java has been the programming language choice to implement such applications. But Java, just as any other managed language, has large memory overhead for each object it creates, which may lead to bloat in the heap. Bloat is the result of several allocations of large objects, such as arrays, and the spreading of the objects it references throughout the heap, after successive garbage collections.

Bloat-aware garbage collection to improve object locality is not a completely new idea. In fact, it has been subject to several research papers (Moon 1984; Wilson et al. 1991; Chen et al. 2006). Although a bloat-aware design for Big Data Applications (Bu et al. 2013) has been proposed, the Java Virtual Machine has no knowledge of the severe load and lack of object locality that Big Data applications bring. This can lead to performance degradation and, in some cases, `OutOfMemoryException` errors.

In Java, much of the use cases for arrays are wrapped within collections, such as `HashMap` or `Hashtable`. This section introduces a new design for the Java heap, aware of the bloat that collections cause, in particular with Big Data applications that run on top of a Java Virtual Machine. Since this new design is targeted for the `ParallelScavengeHeap`, it is hereafter termed as *BDA-heap* (from "Big Data Aware heap"). Also, since a heap must have knowledge of its internals to assert correct functioning, this section also proposes a *bloat-aware Parallel Scavenge* garbage collection algorithm. While developed to extend Parallel Scavenge, the extension is meta-algorithmic, as it can also be used to extend other GC algorithms. The collections chosen to be handled differently were the `java.util.HashMap` and `java.util.Hashtable`

packages, given that they are usually present in Java applications that keep large amounts of in-memory data. However, the following procedures can be applied to all kinds of object classes.

Making the Parallel Scavenge's heap bloat-aware requires knowing which space do *target-objects* live most of their lifetime and then making room for them. This work defines *target-objects* to be those that have tendency to cause bloat on the heap on the long term and should receive special treatment. Target-objects must include two characteristics: must be *long-lived* and large enough. A *long-lived* object is categorized as one that survives long enough to be kept alive in the old generation for a number of full collections. As seen in Section 3.2, old enough objects end up being promoted to the old generation. Hence, those contained in the old spaces are possible candidates to be included in the long-lived set and, if large, also in the target-object set.

3.3.1 Partitioning the old space

A strategy on how to organize the heap space is required to give the old generation a bloat-aware distribution. Parallel Scavenge's old generation is a contiguous committed virtual memory region, backed by a contiguous reserved address range. For the HotSpot VM, the reserved address range is managed by a C++ object that keeps track of two adjoining generations, the old generation and the young generation. It allows the generations to expand and shrink freely under their reserved space, while keeping the separating boundary. The boundary can, however, be moved if necessary. Due to the restriction of only two adjoining generations on this C++ object, it is not trivial to create more generations to hold the target-objects. Therefore, the separation of the spaces had to be done on the old generation itself, such as illustrated in Figure 3.10.

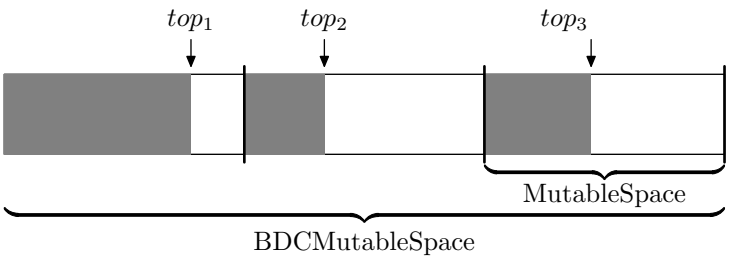


Figure 3.10: The partitioned bloat-aware old generation

To maintain correct VM execution, the old space's division was implemented by subclassing the `MutableSpace` class into a `BDCMutableSpace`, as illustrated in Figure 3.10 (named from "Big Data Collections Mutable Space"). The new space contains an inner class to represent each *BDA-region*, which contains an ordinary `MutableSpace`. A BDA-region is defined as a fragment of the old space that only manages objects of an exact, or derived, type. The `BDCMutableSpace` contains an array of BDA-regions and manages the division of ob-

jects during allocation by implementing the `MutableSpace`'s virtual calls. This design allows transparency to the VM and avoids patching all the extent of the existing code.

Object initialization in the HotSpot VM is executed in two steps: retrieval of the reference pointer through allocation and Klass field initialization. During allocation the only important value is the object size. Therefore, during allocation in the `MutableSpaces` there is no information, whatsoever, about the object's Klass field. Hence, to allocate in a specific `MutableSpace`, the `BDCMutableSpace` uses the calling thread as a proxy to be informed about the object's type. This is so that the new old space only needs to delegate the allocation to the correct BDA-region and update its top-pointer, if needed. This maintains transparency for the `MutableSpace` virtual calls.

3.3.2 Fast Klass information

It is inefficient to frequently check the object size to check if it can be considered a large object. It would require, for each field, the calculation of its size, which would turn into a tree-based calculation for just one object. Inferring through the object type is faster and allows grouping of objects of the same type. Array objects are generally large and can be included as fields of other objects. In fact, most Java collection objects wrap an array of elements and, unless it is a primitive type, each entry is a reference to another object; linked lists are a particular exception. As such, a long-lived array of objects can create bloat on the occupying heap since it accounts its size — header, including array length, plus the entries references —, the size of each entry's object reference and the lifetime of the object. The existence of long-lived large objects of a certain type indicates the tendency for an application to allocate those frequently, and also to keep them alive.

Each application has a number of favorite Java collections it uses for processing their main jobs. At some point in time, these collections have their elements dispersed throughout the heap. Therefore, in order to group them together, objects of those types should be identified using a fast mechanism. As stated in the beginning of this section, the collections that this work handles differently are the `java.util.HashMap` and `java.util.Hashtable` packages. As such, in Algorithm 1 it is shown the retrieval of Klass information for these type of objects. It must be run only once for each object and the value returned kept in a data structure depending on the query-mechanism used. Objects identified as non-critical should be identified as "other", i. e., an ordinary object, while objects of the relevant types should be identified using the package that implements them. For saving the type information, a naïve approach would be to use the unused bits of the object's header mark word. However, the restrictions are high, such as very limited choice, added difficulty and, in some cases, it would not work, e. g., if it was implemented over CMS[2]. This work implements two methods of type addressing: a new header word and hashing of the klass pointer.

[2]http://hg.openjdk.java.net/jdk8u/jdk8u60/hotspot/file/621a3638fd8c/src/share/vm/oops/markOop.hpp

Algorithm 1 Klass information retrieval

```
1: for k = 0 to super_klass_depth do
2:   if k == super − limit then
3:     return region_other
4:   else if substring (super_depth (k) → name (), "java/util/HashMap") then
5:     return region_hashmap
6:   else if substring (super_depth (k) → name (), "java/util/Hashtable") then
7:     return region_hashtable
8:   end if
9: end for
10: return region_other {Default to the other region}
```

Header region mark. An added word gives greater flexibility for object type addressing. On a 64-bit VM (the standard to run Big-Data applications), it allows up to 2^{64} different types, which is more than sufficient. The lookup can be made by masking the bits or direct comparison of the BDA-region identifiers. It is faster when masking the bits because it reduces the overhead of numerous comparisons, which is something to avoid in a Stop-the-World collector. Although this reduces the number of addressable types to the architecture's address width, the amount of managed types is still considerably large. Figure 3.11 shows that this methodology has a larger footprint on the memory and during promotion or compaction (more words to process). However, the information about the object is always precise.

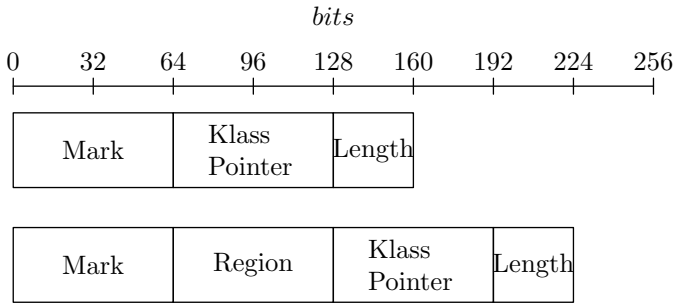


Figure 3.11: Layout of an array object header with an added word for BDA-region indexing

Hashing Klass pointer. Hashing the klass pointer allows indexing type information over an array of entries. Contrary to the additional header word, hashing the klass pointer does not incur additional memory overhead but gives less flexibility and requires more computational effort. Depending on the entry array size, the accuracy can be reduced due to hash collisions. This would be solved with a good hash function but such complexity leads to more overhead, so the hashing function must be kept simple. Also, collisions should not be handled to avoid latency during promotion (in the young collection) or marking (in the full collection), since these are the phases when the object reference is available to have its target BDA-region determined. Figure 3.12 shows three objects with their type in plain text, accessing the array of entries to get their target BDA-regions. Object a sees that there is no region assigned on the entry and fills in

its type ("region hashmap") using Algorithm 1. Object b, on the other hand, accesses an entry with a region set, thus a collision occurs. When it occurs, the object returns the BDA-region identifier it read in the entry and assumes to be of that type. Object c does not need to compute anything and returns the correct region.

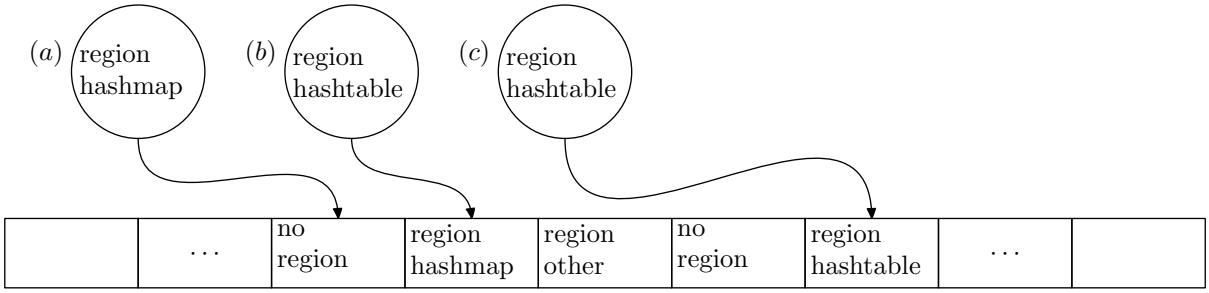


Figure 3.12: An array of entries with BDA-region identifiers

Algorithm 2 demonstrates a simple hash function using a modified Cormen multiplication method — it uses a seed of 101, computes the hash using the lower 24 bits of the pointer and retrieves the most significant 16 bits of the result.

Algorithm 2 : Hash function for Klass pointer — Modified Cormen multiplication method

1: $seed \leftarrow 101$
2: $n\_low\_bits \leftarrow 24$
3: $low\_bits \leftarrow k$ **and** $(1 << n\_low\_bits) - 1$
4: $mix \leftarrow floor(seed \times 1 << n\_low\_bits)$
5: $mult \leftarrow low\_bits \times mix$
6: **return** $hash \leftarrow mult >> (n\_low\_bits - 16)$

3.3.3 Adjusting the space

When the old space is initialized there is no information on how to divide the BDA-regions, and thus are divided evenly. However, allocation on specific BDA-regions can be uneven, unbalancing the space quite significantly and creating large blocks of dead space. Hence, the segmented space must adjust its BDA-regions according to their occupation and free ratios, such as shown in Algorithm 3. The values for *multiplier* and *difference* correspond to predefined values. As for the *MinRegionSize* it is the size of a compacting region and, since all BDA-regions must be region-aligned, it is also the minimum size for a BDA-region, i. e., 64K words (512kB). The algorithm executes the following sequence of steps:

1. Find the BDA-region that needs expansion based on the occupation ratio of all spaces (the value of 80% is predefined)

2. Compute the expected expand size, also based on the occupation ratio
3. Find a forwarding neighbor (the spaces can never expand below the bottom pointer) based on the following criteria:
    - If the free space of the neighbor is much larger than the free space of the expanding space, and it fits more than the double of the expected expand size (line: 14–17), then give up half of the space and break
    - If it is only larger than expand size, then only let the expansion go forward if there is enough free space to give to the other BDA-regions in between (if it is the next neighbor, there are no BDA-regions in the middle)
4. If a neighbor has been found then expand onto it and re-set the limiting pointers (bottom, end and top) accordingly.

Adjusting the old space must be subtle on the BDA-regions, because when expanding a BDA-region the neighbors will have their currently allocated space overrun by the expanding space, and then distributed evenly. Therefore, adjusting the space should never be done during the young collection, because it moves the boundary pointers and that has implications on the *Old-To-Young-Root* tasks (more on Section 3.4.1). It is possible to also use allocation rates in the adjustment strategy but these are more important for the young spaces. In fact, allocation rates are not significative in the old space because objects in the old space are promoted to stay until the end of their lifetime.

3.4 Collecting on bloat-aware Parallel Scavenge

Most allocation in the old space happens with promotion during young collection, or with copying during a full collection; an exception to this rule is direct allocation for objects that still do not fit in the eden space after young collection. So far, a design for a split heap by object type has been given (Section 3.3), but the allocation of objects is still missing. This section fills in that hole and explains how to extend Parallel Scavenge's collections to use the *BDA-heap* old-space. It covers both the young (Section 3.4.1) and full (Section 3.4.2) collections in separate, presents the problems and difficulties that may arise, and a solution for those.

3.4.1 Bloat-aware young collection

As explained in Section 3.2.1, the young collection operates on both the eden and from spaces and, if possible, it tries to promote objects into the old-space. For the *BDA-heap*, as described in the previous section (Section 3.3), this can raise some problems in the decision mechanism and in the scanning phases. The next paragraphs explain how to cope with this difference.

Algorithm 3 Adjusting of BDA-regions

```
1: expanding_region ← max_occupy_ratio (BDA_regions)
2: if expanding_region → occupy_ratio < 80% then
3:    return true {Nothing to do}
4: end if
5: occ_ratio0 ← expanding_region→used / →capacity
6: free_ratio0 = 1− occ_ratio0
7: expand_size ← occ_ratio0 ×multiplier × MinRegionSize
8: k = −1
9: i ← expanding_region-id
10: for j = i + 1 do
11:    neighbor ← region_spaces (j)
12:    free_size1 ← neighbor →free
13:    free_ratio1 ← free_size1 / capacity
14:    if free_ratio1 − free_ratio0 > difference then
15:       if (free_size1 / 2 > expand_size) && (free_size1 / 2 > MinRegionSize) then
16:          expand_size = free_size1 / 2
17:          k = j
18:       end if
19:    else
20:       if free_size1 > expand_size && (pointer_delta (neighbor→end, neighbor→ top + ex-
          pand_size)) ≥ (j − i) × MinRegionSize then
21:          k = j
22:       end if
23:    end if
24: end for
25: if k = −1 then
26:    return false {No one has enough space}
27: end if
28: resize_space (expanding_region, expand_size)
29: return true
```

Old-To-Young Root tasks

The first step for any of the Parallel Scavenge collectors, consists on scanning the references throughout the object graph starting from the root-set. The root-set varies according to the tasks in progress. For scavenging roots in general the root-set is composed of the VM known objects, for thread roots it is composed of the objects in the threads stack frame and registers, and for old roots are the objects with dirtied bits in the card table write-barrier set.

Tasks that scavenge the Old-To-Young roots have a stripe number assigned, according to the ID of the GC thread, i. e., from 0 to the number of active workers minus 1. Each stripe is part of a slice of the whole occupied space in the old generation; therefore, each GC thread processes the same stripe number for all slices until the saved top-pointer. The top-pointer must be saved to avoid transverse the slices onto the PLAB, which are being allocated to promote new objects onto the space. The stripe size is fixed and, consequently, the slice size depends on the number of workers.

The process of finding the Old-To-Young root-set consists on iterating the assigned stripe for all slices. Each iteration consists on the following sequence of steps: get the start and last card for a given stripe; adjust those according to the first object that starts in the range and the last word for the last object that starts in the range; find a range of dirty cards; clean those cards; and push those references into the promotion manager queue. Adjusting the start and last cards is needed because an object may be crossing the address that the card refers to. This requires finding object boundaries close to the cards and adjust accordingly. Finding objects that start after a card index consists on traversing backwards the blocks of an *object-start-array* and, when an object is found on a given block, advance using its size, to the next objects until the one above the card address is found. An *object-start-array* is a byte array divided by 2^9 byte blocks which contain an offset for the last object in that block. Every time an object is allocated in the old generation it must have its offset, from the start of the block, set in the object-start-array.

Figure 3.13 illustrates the finding of an object start address using the *object-start-array* when on a segmented heap. The dashed lines represent cards and not byte blocks. Two cards are illustrated, the first contains object a and part of object b and the second part of object b and object c. If a stripe starts in the second card, it adjusts its start address to the start of the object c, to be able to scan its contents. It does so by traversing backwards the *object-start-array* blocks, finding the start of the object b and jump over its contents using its size. However, if the same is done for object a (such as when a stripe starts in the first card) the *object-start-array* will find object d and jump over its contents. Since the address is not above object a, it will try to jump once again reading that address contents. This will raise a SIGSEGV signal and the VM process will terminate (it does not know how to handle this fault).

With the *BDA-heap*, the old-space is now segmented into BDA-regions each with its own top-pointer and bottom-pointer. Thus, the space below a region's bottom-pointer contains the

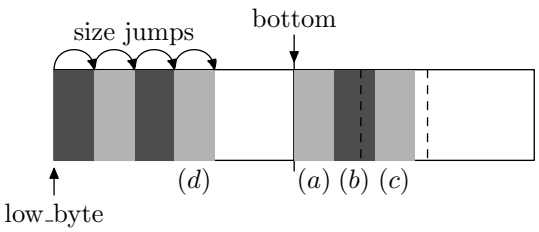


Figure 3.13: Object-start-array fetch of invalid addresses

unallocated space that belongs to the previous space. The *object-start-array* is only aware of the bottom of the old generation (the low_byte in Figure 3.13) to limit its searches. Therefore, when doing a backwards search it may cross the space boundary and start fetching invalid addresses. To solve this, each BDA-region's bottom-pointer must be passed to the methods that find the first object start address. Also, all the top-pointers need to be saved, and not just one. This requirement implies that the old-space can never adjusted while scanning Old-To-Young Root references, such as previously referred in Section 3.3.3, specially because it moves bottom-pointers.

Promoting by type

The *BDA-heap* heap expects that objects belonging to a certain type are allocated in the correct BDA-regions. Promoted objects are allocated from the PLAB (Section 3.2.1) and each GC thread allocates its own, as many times it needs. A PLAB cannot be constantly moving to other BDA-regions on-demand, thus each GC thread must be able to have a number of old generation PLAB equal to the number of BDA-regions. This design is illustrated in Figure 3.14, where $GC\ Thread\ 2$ is promoting through three BDA-regions at the same time using that same number of PLABs. A dashed line corresponds to a flushed PLAB when a new one had to be allocated, e. g., it was full. After fast scanning of the type for the promoted reference (Section 3.3.2), the objects only need to be targeted to the their respective PLAB. In the case that they are directly allocated (a case for objects bigger than half of the PLAB size), or a new PLAB must be allocated, then the bloat-aware collector must set type information on the proxying thread and delegate the allocation call to the old-space. The old-space will, in turn, bump-allocate in the correct BDA-region using the thread's information.

3.4.2 Bloat-aware full collection

A full collection (3.2.2) is subtle on the *BDA-heap*. It requires many changes since half of its work occurs in the old generation. Also, it no longer relies on the old-space virtual calls (thread type proxying) and copying is done on a compacting region basis. The following paragraphs give an explanation on how to extend the existing Parallel Scavenge collector to cope with type-specific spaces. The order of the paragraphs will follow the same order that the collection takes.

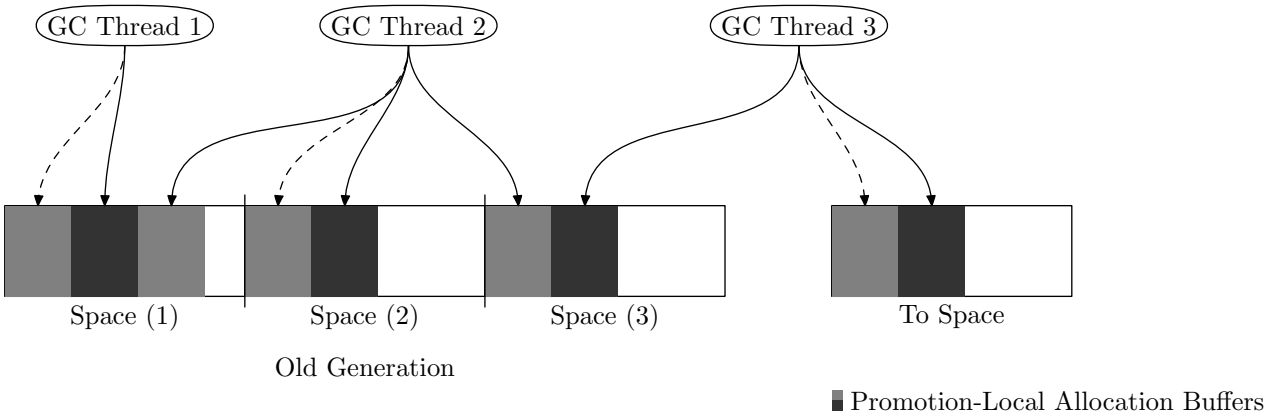


Figure 3.14: Allocation of a variable number of PLAB over BDA-regions

Marking — decision information

Compaction is done on a region basis. After marking (when all live size is computed), there would be no efficient way to scan the region, object by object, and get each object's type with the fast-klass mechanism (Section 3.3.2). Such an approach would cause severe overhead during the summary phase and it could not be done in parallel (the summary is computed by the sole VM thread). However, during marking, each scanned live object, after having its begin and end bits set, has its size amount added to the region they span. At this point, the object's begin address is available for access. Then, it is possible to execute a fast klass-query and decide based on the result. Since the whole region is moved, and not each individual object, the region must have additional fields to help compute the destination region. Thus, each compact region has two additional fields for each special BDA-region, where one indicates the number of objects of type X in the region and another the sum of all type X sizes. This information is then used at the summary phase. As said before, this work implements two special BDA-regions (the BDA-region for general objects must always exist), one for `HashMap` based objects and another for `Hashtable` based objects. Therefore, a compacting region should contain the following fields:

- Count of `HashMap` based objects
- Total size of `HashMap` based objects
- Count of `Hashtable` based objects
- Total size of `Hashtable` based objects

Summary — targeting regions

At the summary phase, the old-spaces first compute their new top-pointer values by compacting into themselves. This leaves room for the promotion of all the young spaces, if they fit. In

the unmodified Parallel Scavenge, the summary of young spaces only targets one old-space. However, in bloat-aware compaction the summary of young spaces must target all the BDA-regions. Since on the marking phase the information about the special objects was already computed, each iteration of the young space's regions (Section 3.2.2.2) will use Algorithm 4 to decide which BDA-region will set the region's destination field. This is needed because some regions may have an approximate number of special object elements, thus the collector needs to know which BDA-region will target. The algorithm decides based on a given *threshold* (set at the VM startup) which weights the ratio of the number of objects in order to choose a decision metric. The decision metric is one of two: the count of objects for each space, or the average size for each special object. This allows choosing between heavy objects or many small objects, depending on the application characteristics. However, the decision algorithm cannot happen if the region under processing contains a partial object, i. e., an object from a previous region extends onto the region. In this case, the region under processing must be targeted to the same BDA-region as the previous to avoid split the object's words.

Algorithm 4 Decision algorithm for target BDA-region

```
1:  avg_hashmap_element ← hashmap_totalsize/hashmap_count
2:  avg_hashtable_element ← hashtable_totalsize/hashtable_count
3:  if hashmap_count > hashtable_count then
4:     ratio ← hashtable_count/hashmap_count
5:  else
6:     ratio ← hashmap_count/hashtable_count
7:  end if
8:  if ratio ≤ threshold then
9:     if hashmap_count > hashtable_count then
10:       target ← region_hashmap
11:    else
12:       target ← region_hashtable
13:    end if
14: else
15:    if avg_hashmap_element > avg_hashtable_element then
16:       target ← region_hashmap
17:    else
18:       target ← region_hashtable
19:    end if
20: end if
```

Compacting — dealing with region stealing

During compaction, Parallel Scavenge compaction tasks use summary data to fill regions. They get the source field and iterate through all marked objects in the source region, copying words to the target destination. Regions are filled until full or, for the last region in a space, until the new pre-computed top-pointer. If the source is fully claimed, i. e., no more words to copy, and

the region is still not full, it gets a new, non-empty, source region.

The next source region is fetched by iterating through the next adjacent source regions to find the first that is not empty. Parallel Scavenge, in its native form, imposes no restriction on region iteration and fetching. The reason is that, for self-compacting spaces the next source region always targets that same space, and for promotion to the old-space the next source regions must target the only old-space in existence. But, now there are several old-space partitions and, as illustrated in Figure 3.15, regions are targeted to other spaces. In this figure, there are two GC threads consuming a filling task. $GC\ Thread\ 1$ will fetch region A and copy to the beginning of the $Space\ 1$, and $GC\ Thread\ 2$ will do the same with region B. Since region A will not fill a complete region in $Space\ 1$, then the $GC\ Thread\ 1$ will get the next source region to fill the remaining space, and that will be region B. This "stealing" causes wrong word placement and a negative *destination-count* — both threads will decrement the *destination-count* and region B was only targeted for one destination. To avoid the fetching of regions in an intrusive way the algorithm must check if a destination of a region corresponds to the expected destination space.

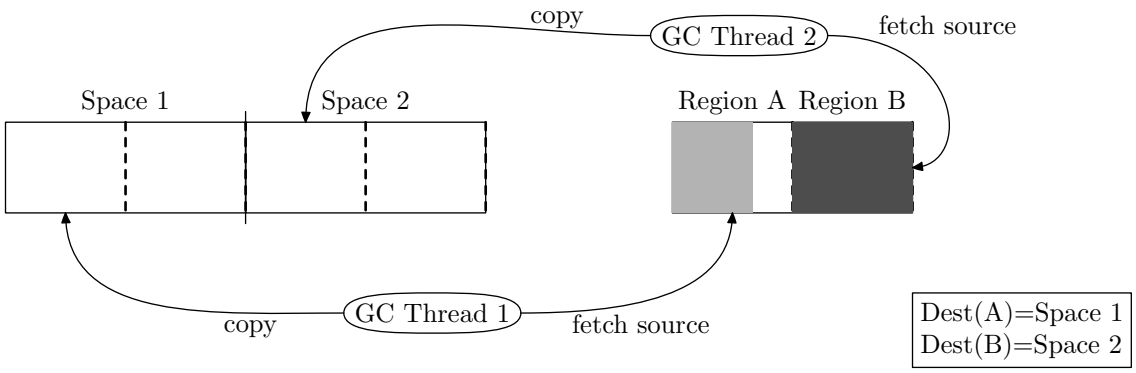


Figure 3.15: Compaction — fetching and filling of regions

Cleanup — the top-pointer

On the cleanup phase, the new top-pointers are finally committed to the spaces. The top-pointer reflects the amount of data for each space and anything above is unused-space. As referred in Section 3.3, the bloat-aware old-space tries to be as transparent as possible to the rest of the VM. Therefore, the top-pointer of the old-space must reflect the whole occupied space below. As a consequence, it must be set equal to the top-pointer of the last BDA-region, i. e., the higher address of them all.

Summary

This chapter described the whole design for a bloat-aware adaptation for Parallel Scavenge and an extended algorithm to be able to promote to objects based on their type. This design

provides a way of contention for objects that have tendency to cause bloat, i. e., collections. The contention is advantageous to provide a way for dependent objects (elements of the array) to be closer to the wrapper object and to prevent them of mixing with the other objects in the heap. Following the architecture's directives, such as how to split spaces (3.3) and how to collect on those (3.4), it is possible to port this design to other collectors.

4 Implementation

> Always bear in mind that your own resolution to succeed is more important than any other. — *Abraham Lincoln (1809–1865)*

So far, only high-level mechanisms have been shown. With the architectural design in place, presented in Chapter 3, the low-level details of this dissertation's work are now ready to be described. The architecture was presented in a template view of the modules that required modification, divided in two sections: Section 3.3 for the bloat-aware heap and Section 3.4 for collecting in such heap. This chapter fills in that template with code snippets.

One important aspect when implementing was transparency to the VM and try to modify critical sections, such as fast-paths and hot-methods, as little as possible. In such methods, the code that had to be injected in order to implement different strategies, such as fast Klass-queries with the two devised mechanisms, was guarded by C pre-processor directives. This significantly reduces the amount of if-like conditions. Transparency was achieved implementing everything separate but adhering the VM norms on class hierarchy.

Another important aspect of the HotSpot VM is how it handles allocation of its data structures. The HotSpot VM states that every class must be a subclass of one of the following classes: `StackObj`, `ValueObj`, `ResourceObj`, `AllStatic`, `MetaspaceObj`, and `CHeapObj`. `StackObj` represent values allocated in the stack, `ValueObj` represent embedded values, `ResourceObj` for values allocated in the thread's resource area, `AllStatic` for classes used as namespaces and `MetaspaceObj` for classes stored in the Metaspace, i. e., loaded Java class data. On the other hand, a `CHeapObj` represents values allocated in the C-heap and it overrides all `new` operator calls. Therefore, the HotSpot VM has its own structure for raw memory representation, and thus provides macro calls as interfaces for allocation. This implementation attempt uses the HotSpot's data structures to better integrate with the existent code.

4.1 Partitioning the old-space

As described in Section 3.3.1, and illustrated in Figure 3.10, the partition of the old-space went by subclassing the `MutableSpace` into a `BDCMutableSpace`. A `BDCMutableSpace` acts as a container for the `CGRPSpace` class, which represents a BDA-region. Each `CGRPSpace` class (named from "Collection Group Space") contains information about the region it manages and a pointer to a `MutableSpace` object. Listing 4.1 shows how a `CGRPSpace` object is structured.

```
class CGRPSpace : public CHeapObj<mtGC> {
   BDARegion _coll_type;
   MutableSpace* _space;
  public:
   CGRPSpace(size_t alignment, BDARegion coll_type) : _coll_type(coll_type) {
     _space = new MutableSpace(alignment);
   }
   ~CGRPSpace() {
    delete _space;
   }
   static bool equals(void* group_type, CGRPSpace* s) {
    return *(BDARegion*)group_type == s->coll_type();
   }
   BDARegion coll_type() const { return _coll_type; }
   MutableSpace* space() const { return _space; }
 };
```

Listing 4.1: CGRPSpace class

```
BDCMutableSpace::BDCMutableSpace(size_t alignment) : MutableSpace(alignment) {
  _collections = new (ResourceObj::C_HEAP, mtGC) GrowableArray<CGRPSpace*>(0, true);
  _page_size = os::vm_page_size();
  // initializing the array of collection types
  // It must be done in the constructor due to the resize calls
  collections()->append(new CGRPSpace(alignment, region_other));
  collections()->append(new CGRPSpace(alignment, region_hashmap));
  collections()->append(new CGRPSpace(alignment, region_hashtable));
}
```

Listing 4.3: BDCMutableSpace constructor

The `BDARegion` (named from "Big-Data Aware Region") type deserves an explanation. It is the basic identifier for all BDA-regions, as shown in Listing 4.2. Each region has a number assigned such that masking all bits from all regions should return the representation for NULL (0×0). This allows the fast klass-queries to execute quick if-conditions.

```
enum BDARegion {
  no_region = 0x0,
  region_other = 0x1,
  region_hashmap = 0x2,
  region_hashtable = 0x4
};
```

Listing 4.2: BDA-region identifiers

Therefore, the constructor for the `BDCMutableSpace` must push the needed amount of BDA-regions onto its array of managed BDA-regions, such as illustrated in Listing 4.3.

Allocation in the segmented old-space requires, by the space's command, a target BDA-

```
HeapWord* BDCMutableSpace::cas_allocate(size_t size) {
  Thread* thr = Thread::current();
  BDARegion type2aloc = thr->alloc_region();

  int i = collections()->find(&type2aloc, CGRPSpace::equals);

  CGRPSpace* cs = collections()->at(i);
  MutableSpace* ms = cs->space();
  HeapWord* obj = ms->cas_allocate(size);
  if(obj != NULL) {
    HeapWord* region_top, *cur_top;
    while((cur_top = top()) < (region_top = ms->top())) {
      if( Atomic::cmpxchg_ptr(region_top, top_addr(), cur_top) == cur_top ) {
        break;
      }
    }
  } else {
    return NULL;
  }
  return obj;
}
```

Listing 4.4: Allocation in BDCMutableSpace

region and the updating of the whole space top-pointer (to reflect that all space below contains live data). The allocation calls are generally multi-threaded, as such, it needs an atomic mechanism in order to update the BDA-region's top-pointer. It is also important to assure that if a chunk is successful in allocating within a BDA-region, then the whole space must conform to that change, i. e., it must increase its top-pointer if need be. The allocation mechanism is shown in Listing 4.4. It can be seen that the allocation is done over the BDA-region first. That is to assure that a successful allocation in the BDA-region must be successful for the whole space as well. Also, the current thread is that which contains the decisive value for the target BDA-region. This value must be set previously by the garbage collection code and is described in more detail in Section 4.3.

4.2 Fast Klass information

Section 3.3.2 introduced two mechanisms to retrieve Klass information in a fast-path. However, the Klass information must first be retrieved from the object before it is saved on the chosen mechanism data structures. Both of these mechanisms use the same procedure for retrieval: parsing the *name* field of the Klass object. The *name* field is of type `Symbol`, a canonicalized string which for instanced Java objects represents its Klass name. Therefore, for a `HashMap` Java object its `Symbol` representation would return a *java/util/HashMap* string. Shown in Algorithm 1 are the steps taken when iterating through all the super classes of the object. This is needed since the Klass under inspection could be a sub-class of the type in search (e. g.,

```
class regionMarkDesc : public oopDesc {
private:
  // Conversion
  uintptr_t value() const { return (uintptr_t)this; }
public:
  // Constants used for shifting the word
  enum { region_bits = 4 };
  enum { region_shift = 0 };
  enum {
    region_mask     = right_n_bits(region_bits),
    region_mask_in_place = region_mask << region_shift
  };
  bool is_hashmap_oop() const {
    return mask_bits(value(), right_n_bits(region_bits)) == region_hashmap;
  }
  regionMark set_hashmap() {
    return regionMark((value() & ~region_mask_in_place) | region_hashmap);
  }
  // inline statics in order to construct this
  inline static regionMark encode_pointer_as_hashmap(void* p) { return
      regionMark(p)->set_hashmap(); }
  inline static regionMark encode_pointer_as_region(BDARegion r, void* p) {
    if(r == region_hashmap)
      return encode_pointer_as_hashmap(p);
    else
      return encode_pointer_as_noregion(p);
  }
  inline BDARegion decode_pointer_as_region() { return (BDARegion)value(); }
};
```

Listing 4.5: Header region mark — regionMark.hpp

an extended HashMap). The values returned must be always compliant with the `BDARegion` enumeration definition.

The following sections explain how each of the mechanisms is implemented. Both mechanisms must run Algorithm 1 before storing the information.

4.2.1 Header region mark

One way for retrieving previously stored Klass information is adding a new header word. This approach gives each object reference the possibility of carrying, for all its lifetime, the needed information for storage in the *BDA-heap*. Listing 4.5 shows how a header word can be described (the Listing shows less BDA-regions than currently implemented to reduce space). It uses the bits of its value to represent the BDA-regions, previously defined in the `BDARegion` enumeration. Hence, whenever it needs to set the region word it masks its bits to 0 and adds the new value.

The `regionMarkDesc` describes a value and it needs to be added to the header of every object reference. Before doing so, the header region mark should become a pointer to the above

```
typedef class markOopDesc*       markOop;
typedef class regionMarkDesc*    regionMark;
```

Listing 4.6: Oop Hierarchy — oopHierarchy.hpp

```
class oopDesc {
  friend class VMStructs;
 private:
  volatile markOop _mark;
#ifdef HEADER_MARK
  volatile regionMark _region;
#endif
  union _metadata {
    Klass*   _klass;
    narrowKlass _compressed_klass;
  } _metadata;
  //...
};
```

Listing 4.7: RegionMark installed in oop — oop.hpp

class description to assert that the word only occupies the width for a pointer in the current machine (8 bytes for 64-bit VM) and to simplify future accesses to the type. The HotSpot VM already defines an *oop* hierarchy, thus the added word must be added to this hierarchy. An excerpt is shown in Listing 4.6.

The `markOop` is the header mark word, used by GC, thread biasing and hash value of the object pointer. To maintain correct runtime state of the VM, the `regionMark` word must be implemented after the `markOop` word — there are calls that depend on the `markOop` and `Klass` address offsets within the oop, therefore some care must be taken. This is illustrated in Listing 4.7.

The whole heap is filled with invalid words at the VM's startup, a term defined as "*zapped*". It is fitting that the `regionMark` value must be set at some point. The ideal place to set its value is when the `Klass` field is set for the object, as illustrated in Listing 4.8.

```
inline void oopDesc::set_klass(Klass* k) {
  // ...
#ifdef HEADER_MARK
  BDARegion r = k->is_subtype_for_bda();
  set_region(regionMarkDesc::encode_pointer_as_region(r, 0x0));
#endif
  // ...
}
```

Listing 4.8: Set of regionMark word — oop.inline.hpp

```
BDARegion
KlassRegionMap::region_for_klass(Klass* k) {
  uint hash = compute_hash((intptr_t)k);
  int index = indexof(hash);
  if(table()->at(index) == (char)no_region) {
    BDARegion region = k->is_subtype_for_bda();
    (table()->at(index)) = (char)region;
    return region;
  } else {
    return (BDARegion)(table()->at(index));
  }
}
```

Listing 4.9: Hashing the Klass pointer

4.2.2 Hashing Klass pointer

Contrary to the Header region mark, the hashing of the Klass pointer does not save the returned value in any field. Therefore, the lookup in the hash array has to be done every time a query is made. The lookup includes computing the hashed value of the Klass pointer in every instance, as shown in Listing 4.9. The `compute_hash` method is implemented using the Algorithm 2. If when indexing, the value kept is still a `no_region` (the array is initialized with `no_region` on all its entries) then the Klass information must be fetched, using Algorithm 1, and set in the array. There is no collision resolution to reduce the computation to a minimum, therefore a correct result depends on the number of array slots. The region identifiers are encoded as `char` to occupy only one byte.

4.3 Bloat-aware young collection

With segmented spaces for keeping target-objects separated by type, the object promotion code requires some changes to cope with this demand. Section 3.4.1 showed the problems that can arise when the old-space becomes split into BDA-regions, with unallocated space in between. That same Section also presented a way of informing the delegated space about the type of the object that is under promotion, which is using a thread to proxy this information. This Section illustrates how the young generation collector — the *scavenger* — was extended.

4.3.1 Old-To-Young Root tasks

The Old-To-Young root tasks, in their native form, are only aware of a contiguous, bump-allocated, space. They respect no boundaries while scanning old objects — they scan until the bottom byte, which represents the lower raw address of the heap. These tasks, when created, get the current old-space top-pointer so that they do not transverse their scanning onto the

```
class BDACardTableHelper : public CHeapObj<mtGC> { private: int
  _length; HeapWord** _tops; MutableSpace** _spaces; public:
  BDACardTableHelper(BDCMutableSpace* sp); ~BDACardTableHelper();
  int length() const { return _length; } HeapWord** tops() const {
    return _tops; } MutableSpace** spaces() const { return _spaces;
  } };
```

Listing 4.10: BDACardTableHelper

```
#if defined(HASH_MARK) || defined(HEADER_MARK)
   for(int i = 0; i < _helper->length(); i++) {
     card_table->scavenge_contents_parallel(_gen->start_array(),
                                    _helper->spaces()[i],
                                    _helper->tops()[i],
                                    pm,
                                    _stripe_number,
                                    _stripe_total);
   }
```

Listing 4.11: Scavenging the contents of the old generation separately

PLAB. With split spaces there are more top-pointers to keep. To achieve this, a new class is implemented for saving all the top-pointers. This class, `BDACardTableHelper` ("Big-Data Aware Card Table Helper") as shown in Listing 4.10, not only saves the top pointers but also the pointer to each space to access more information.

When scavenging the contents of the old-space, the card table divides the old generation in slices and gets its assigned stripe to scan for dirty objects. This is done in a contiguous form, from the bottom of the space until the saved top. In the *BDA-heap*, the old generation is split. So, in order to prevent the jumping of the gaps between each space, each split space is scavenged separately; as shown in Listing 4.11 (the `_helper` field is a pointer to a `BDACardTableHelper` object).

Scavenging of the Old-To-Young objects in parallel stripes require traversing the *object-start-array*. This happens when the start and end boundaries of a stripe need to adjusted and to find the start and end of a dirty range of objects. To find the beginning of an object, a backwards search is required. And, as mentioned at the begin of this Subsection 4.3.1, the *object-start-array* is only aware of the bottom raw byte address of the heap. This can cause accessing invalid memory addresses due to the scanning for objects next to the last, i. e., objects past the top-pointer of a BDA-region which do not exist. Before scavenging the contents, it is asserted that if there are no objects starting within the stripe then it skips to the next slice. Therefore, if the *object-start-array* is called then an object must start in the stripe's range. Since the first object in a BDA-region must start at the bottom address of its BDA-region, then it is possible to limit the search backwards with the bottom pointer of that BDA-region, as shown in the excerpt in Listing 4.12. The usage of the `_spaces` array within the `BDACardTableHelper` is thus

```
HeapWord* object_start(HeapWord* addr, HeapWord* low_bound) const {
   jbyte* block = block_for_addr(addr);
   HeapWord* scroll_forward;
   // This code prevents the scan of objects below a specified region
   if(addr <= low_bound)
     scroll_forward = addr;
   else {
     scroll_forward = offset_addr_for_block(block--);
   }
   while (scroll_forward > addr) {
     scroll_forward = offset_addr_for_block(block--);
   }
   // ...
}
```

Listing 4.12: Extended version of object_start method of Object Start Array

justified. The requirement that the old-space cannot be adjusted while scanning Old-To-Young Root references is kept.

4.3.2 Promoting by type

The main goal of this thesis's work is to be able to promote objects to an old generation split into different BDA-regions. It was shown how to acquire fast Klass type information in order to reduce the overhead of this search. It was also shown in Section 3.4.1 that a GC thread must contain the same number of PLAB as the number of BDA-regions, and that it will act as proxy for the promotion code to allocate the next PLAB. When the promotion manager resets its state it always initializes the PLAB to a zero-size memory region. When promoting, the promotion manager must allocate the first PLAB. It also flushes the current PLAB in the case it is not the first. In the *BDA-heap*, this code is shared among fast klass mechanisms given that both set the right value to the proxying thread, as illustrated in Listing 4.13 for objects of type `HashMap`.

The next paragraphs show how the promotion can be achieved using the two presented fast klass-query mechanisms, and from where the above-mentioned `target` value comes. The following snippets come before Listing **??**, i. e., allocation is first tried on the PLAB and only, if unsuccessful, another is allocated.

Promoting with header region mark. It was shown in Listing 4.8 how a header region mark is set for an object reference. Listing 4.5 also shows a number of methods for querying and to decode the value of the word. Before trying to allocate a new PLAB, the promotion mechanism tries to allocate in the existing PLAB. The header region mark way of doing so is shown in Listing 4.14.

Promoting with hash of klass pointer. On the other hand, when hashing the Klass pointer the target region must be fetched from the hash array, as represented by the Listing 4.15. Then,

```
  Thread *thr = Thread::current();
  //...
if (new_obj == NULL) {
#if defined(HASH_MARK)
  thr->set_alloc_region(target);
#elif defined(HEADER_MARK)
  thr->set_alloc_region(o->region()->decode_pointer_as_region());
#endif
  //...
  HeapWord* lab_base = old_gen()->cas_allocate(OldPLABSize);
  // ...
#if defined(HASH_MARK) || defined(HEADER_MARK)
  if(thr->alloc_region() == region_hashmap) {
    _hashmap_old_lab.flush();
    _hashmap_old_lab.initialize(MemRegion(lab_base, OldPLABSize));
    // Now try the collections old plab again
    new_obj = (oop) _hashmap_old_lab.allocate(new_obj_size);
  } else
#endif
  {
    _old_lab.flush();
    _old_lab.initialize(MemRegion(lab_base, OldPLABSize));
    // Or try the old lab allocation again.
    new_obj = (oop) _old_lab.allocate(new_obj_size);
  }
```

Listing 4.13: Allocation of a PLAB with thread proxying

```
if (o->region()->is_hashmap_oop())
  new_obj = (oop) _hashmap_old_lab.allocate(new_obj_size);
else if (o->region()->is_other_oop())
  new_obj = (oop) _old_lab.allocate(new_obj_size);
else {
  // The header is none of the above and it is being scavenged...
  // weird kind of oop (klass, etc)
  o->set_region(regionMarkDesc::encode_pointer_as_other(0x0));
}
```

Listing 4.14: Header region mark allocation on a PLAB

```
BDARegion target = old_gen()->region_map()->region_for_klass(o->klass());
// If plab is already allocated, go through it directly
if (mask_bits(target, region_hashmap)) {
  new_obj = (oop) _hashmap_old_lab.allocate(new_obj_size);
} else {
  new_obj = (oop) _old_lab.allocate(new_obj_size);
}
```

Listing 4.15: Hash of Klass pointer allocation on a PLAB

for faster computation, the bits for the returned BDA-region are masked with the bits corresponding to each BDA-region identifier. The `region_map()` method returns a pointer to the above-mentioned `KlassRegionMap`.

4.4 Bloat-aware full collection

Just as the young collection, the full collection algorithms need their implementation extended for awareness in the *BDA-heap*. However, the Parallel Compact implementation is quite extensive and is divided in four phases. Each phase requires a decisive action, since this collector operates on the old-space which now became split. The following subsections follow the same pattern presented in the Architecture Section 3.4.2 and show, for each phase, a way of implementing the issues raised in that section. Given that compaction is done on a region basis, it is important to keep the distinction of a **BDA-region** — a space resultant of the splitting in the old generation — and a compacting **region** — a memory range of fixed-size implemented as `RegionData`.

4.4.1 Decision information

While marking, it was stated that each live object had some additional information added to the region. This information constitutes two new fields for each special BDA-region: number of objects found of a certain type and total size of objects of that type. As illustrated in Listing 4.16, this work implements, for its three BDA-regions, two new pair of fields; one for `HashMap` and another for `Hashtable`. To avoid adding one more method per BDA-region, each increment for a special object does two operations: increments the counter and adds the size to the total.

What is left is calling these methods. This is done in the `add_obj()` method for the summary data, called by each GC thread after setting the bits in the parallel bitmaps. Therefore, as shown in a short snippet of the `add_obj()` method in Listing 4.17, the values for the increment of counters depend on the object size or, if the object spans different regions, the size it occupies within those. In the Listing, the value of `r` has assigned a BDA-region identifier retrieved using one of the methods shown in Section 4.2.

```
int             _hashmap_ctr;
int             _hashtable_ctr;
size_t          _hashmap_total_size;
size_t          _hashtable_total_size;
// ...
inline jint
ParallelCompactData::RegionData::incr_hashmap_counter(size_t sz) {
  Atomic::inc((volatile int*)&_hashmap_ctr);
  return Atomic::add((jint)sz, (volatile int*)&_hashmap_total_size);
}
inline
jint ParallelCompactData::RegionData::incr_hashtable_counter(size_t sz) {
  Atomic::inc((volatile int*)&_hashtable_ctr);
  return Atomic::add((jint)sz, (volatile int*)&_hashtable_total_size);
}
```

Listing 4.16: Added fields to RegionData

```
if(r == region_hashmap)
  _region_data[beg_region].incr_hashmap_counter(len);
else if(r == region_hashtable)
  _region_data[beg_region].incr_hashtable_counter(len);
```

Listing 4.17: Object data added to region

4.4.2 Targeting regions

After the marking phase, the region data has enough information to decide the target empty region to set as destination during the summary phase. The Algorithm 4 already demonstrated the decision method for this computation. The method that implements this behavior is shown in Listing 4.18. It is implemented in a similar way to the existing `summarize()` method in `ParallelCompactData` (a class that contains a set of `RegionData` objects) but aware of three group of pointers. If dynamic BDA-regions are required, this method can make use of the `<stdarg.h>` library for variable arguments.

```
bool summarize_parse_region(SplitInfo& split_info,
  HeapWord* source_beg, HeapWord* source_end,
  HeapWord** source_next,
  HeapWord* target0_beg, HeapWord* target0_end,
  HeapWord** target0_next,
  HeapWord* target1_beg, HeapWord* target1_end,
  HeapWord** target1_next,
  HeapWord* target2_beg, HeapWord* target2_end,
  HeapWord** target2_next);
```

Listing 4.18: Summarize an young space to the old generation

```
while (src_region_ptr < top_region_ptr) {
  if(space_id(src_region_ptr->destination()) != space_id(closure.destination()) ||
     src_region_ptr->data_size() == 0) {
       ++src_region_ptr;
       continue;
     }
     break;
   }
```

Listing 4.19: Dealing with "region stealing"

```
HeapWord* max = MAX3(_space_info[old_space_other_id].new_top(),
                     _space_info[old_space_hashmap_id].new_top(),
                     _space_info[old_space_hashtable_id].new_top());
_space_info[old_space_id].set_new_top(max);
_space_info[old_space_id].publish_new_top();
```

Listing 4.20: Publish of the old generation top-pointer

4.4.3 Dealing with region stealing

It was previously raised (in Section 3.4.2) a problem for "region stealing" by the other GC threads. This issue happens during the fetching of new source regions, when the thread is iterating through the adjacent regions. If a thread does not check if the destination of that next region is for the space it is filling, then a "region steal" occurs. If such happens, the future of the compact algorithm is compromised because data becomes corrupted and the destination-count gets wrong values. In order to fix this "stealing", the `next_src_region` method contains an added condition while iterating supposed empty regions; Listing 4.19. In the Listing the `top_region_ptr` is a pointer to a `RegionData`

4.4.4 Broadcast top-pointers

The HotSpot VM relies on the top-pointers for knowledge about the used space in any allocated space. Although the top-pointer for the whole old generation, i. e., the top-pointer for the `BDCMutableSpace`, is rarely used within the VM, it is important to make sure that the VM knows that below there is live data. At allocation time on the old generation — Listing 4.4 — the top-pointer for the `BDCMutableSpace` is updated only if the last updated object had an address bigger than his. Thus, it is common to have the old generation top-pointer contained in the last BDA-region. Listing 4.20 shows how the old generation top can be published so that it always stays above, or equal, to the other top-pointers.

Summary

This chapter highlighted the most important implementation details regarding the bloat-aware heap extension on the Parallel Scavenge garbage collector of the HotSpot VM. It also gave a picture on how the implementation of the extensive rest of the source code was written.

5 Evaluation

Success is not the key to happiness. Happiness is the key to success. If you love what you are doing, you will be successful. — *Albert Schweitzer (1875–1957)*

5.1 Overview

This chapter presents the results obtained after putting to the test the implementation of the architecture presented in Chapter 3 and the implementation in Chapter 4. Given that there are numerous tuning metrics, such as the *threshold* for summarizing compaction (Section 3.4.2) and the size of the hash array for the decision mechanism by hashing the Klass pointer (Section 3.3.2), a transverse study would lead to countless graphs. Hence, the next sections start by narrowing the case study and only then a comparison between this implementation and the unmodified HotSpot virtual machine is made, using the same configuration.

The rest of the chapter is organized as follows: a narrowed approach to obtain an approximation of the optimal size for the hasharray in Section 5.2; a study of the *threshold* value in Section 5.3; and performance and locality benchmarking using the best results of the two previous sections in Section 5.4.

The following results are obtained using the DaCapo benchmark suite (Blackburn et al. 2006), using three specific benchmarks, considered to be intensive on data processing:

- H2, which executes a number of transactions, using multiple threads, against the H2 Java SQL database;
- Tradebeans, which executes a daytrader benchmark via JavaBeans on a GERONIMO [1] server backed by an in-memory H2 database;
- Tradesoap, which is the same as Tradebeans but via SOAP.

The reason behind choosing such benchmarks is that, of all the DaCapo suite, these are the ones stressing the old generation the most. It is also important to note that the object locality results naturally depend, in part, on the iteration it ran on. However the tendency can be clearly traced and analyzed using the specified metrics (*threshold* and size of hash array).

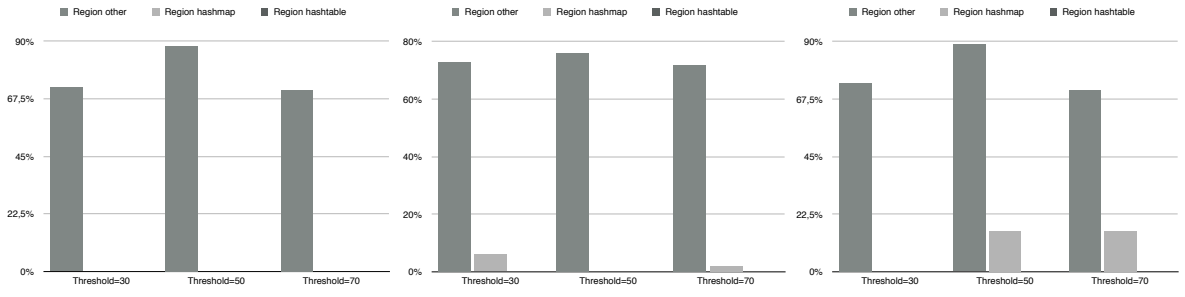
[1] http://geronimo.apache.org/

The benchmarks are run in a 8-core machine with 12GB of memory. All of the runs executed on 12 iterations with a minimum heap size of 1.5GB and a maximum heap size of 8.5GB.

5.2 Choosing Better Locality

Although hashing the Klass pointer as a decision mechanism for allocation in the BDA-regions is a low memory overhead solution, it lacks the precision that the header region word has. While one mechanism makes that every object should carry information about its Klass field, the other depends on an external table. Hashing of the Klass pointer requires computing an hash value for its Klass pointer to index an array of elements. Since this is a hot-path during promotion, the hash function needs to be kept simple. This causes collisions when indexing. The method for reducing such collisions is to increase the array size. However, it should not be too large because it would trigger, too many times, a run for Algorithm 1 which would reduce the performance of the garbage collector.

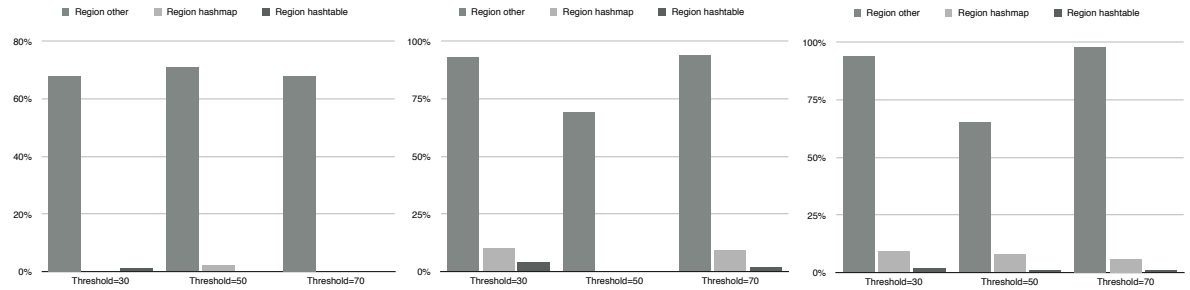
Figures 5.1, 5.2 and 5.3 show the object distribution, by occupancy ratio. It can be observed that for array sizes of 800 entries, the occupancy on the *Region hashmap* and *Region hashtable* is very low. This is a problem because this solution leverages the object placement and co-locality (dependent objects are close in terms of memory offsets). The reason for the result is due to many hash collisions when an object tries to find its BDA-region information. Since most of the objects used in the VM would be categorized as "other", they quickly set to *Region other* most of the array entries, leaving no room for the special objects to set their information in the array, i. e., `HashMap` and `Hashtable` variants. However, on array sizes of 1200 entries the promotion of objects of special types yields correct results improving co-locality and reducing the bloat in the *Region other* space, as can be seen in the graphs with the values of 16% in H2, 6%-9% in Tradebeans and 1% to 6% in Tradesoap for the *Region hashmap*.



(a) Array size 800 (b) Array size 1000 (c) Array size 1200

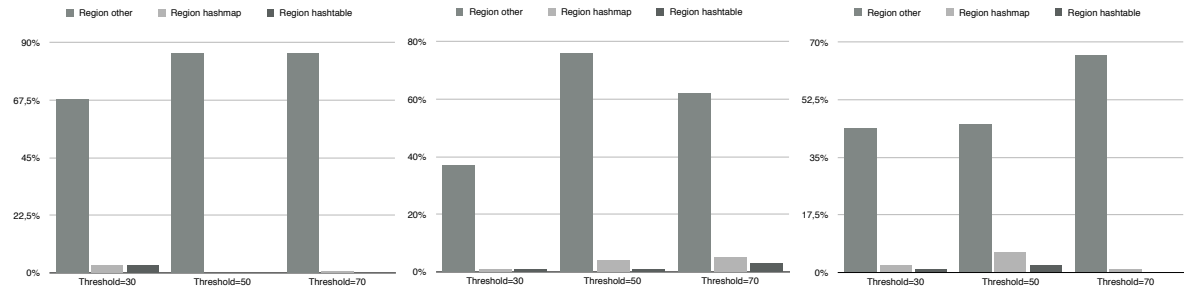
Figure 5.1: H2 — Used space using the hash of the Klass pointer on an array of 800 (a), 1000 (b) and 1200 (c) entries, using the given decision thresholds

It is natural to think that making the array larger may result in better locality (close to the header region mark which yields exact results). However, excessive calls for the Algorithm 1



(a) Array size 800 (b) Array size 1000 (c) Array size 1200

Figure 5.2: Tradebeans — Used space using the hash of the Klass pointer on an array of 800 (a), 1000 (b) and 1200 (c) entries, using the give decision thresholds



(a) Array size 800 (b) Array size 1000 (c) Array size 1200

Figure 5.3: Tradesoap — Used space using the hash of the Klass pointer on an array of 800 (a), 1000 (b) and 1200 (c) entries, using the give decision thresholds

may result in latency and that must be avoided in hot-paths such as this. As shown in Figures 5.1, 5.2 and 5.3, it can be deduced that for an array size of 1200 entries the object placement tends to satisfy this work's goals, as shown, for example, in values of 16% for the H2 runs.

5.3 Evaluating Object Locality

The previous section demonstrated that a good value for the hash array size is 1200 entries, because it reduces collisions and does not incur too much overhead with triggering an object search. During full collection objects are not promoted individually. A full collection is region-based, compacting in windows of 64K words. As seen in Section 3.4.2, a decision mechanism is needed in order to target the most compelling of the target BDA-regions. The decision mechanism leverages the *threshold* value to help decide if it should give priority to the number of elements or to the average size of each element. Here, in this section it is shown how the *threshold* value can help deciding the optimal object placement during a full collection, for each klass-query mechanism.

It is important to note that the adjusting of spaces can reduce the amount of words currently allocated in that space, because the neighbor of the expanding space can be overrun.

This is illustrated in the graphs when the number of allocated words suddenly drops. They regain their allocated space in the next garbage collector runs. It is also important to assess the ordering of the spaces in order to deduce if a drop of the number of words was caused by an adjustment of the space or by a collection. The spaces are ordered as follows:

1. *Region other* which contains general-purpose objects;
2. *Region hashmap* which contains `HashMap`-based objects;
3. *Region hashtable* which contains `Hashtable`-based objects.

Therefore, the *Region other* is at the bottom of the old generation and the *Region hashtable* has its end-pointer at the currently committed virtual space.

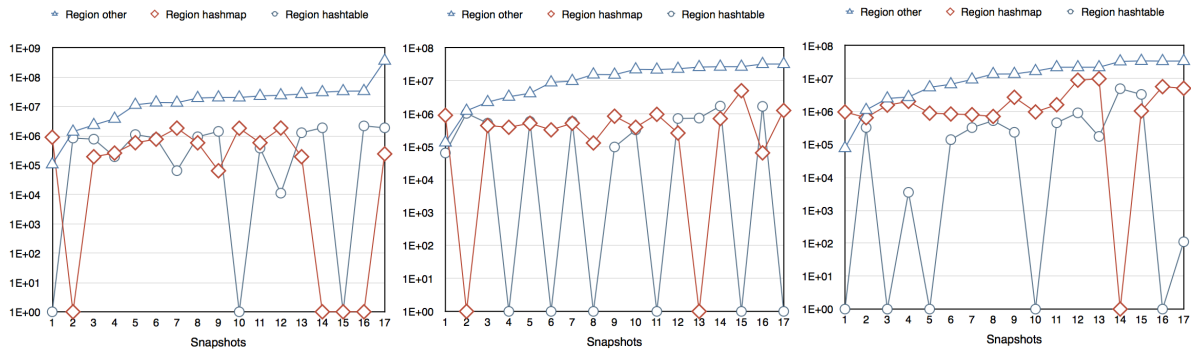
5.3.1 Header word locality

The header region word specifies BDA-region targets for each object, in a precise manner, by saving the bits that correspond to the BDA-region identifier. Therefore, any special object that is marked as live, in the marking phase, is added to the compaction region statistics for special objects. This makes the compacting region a candidate to be promoted onto a special object BDA-region (*Region hashmap* or *Region hashtable*).

The *threshold* value decides the target region, in accordance to the number of special elements it contains, and their average size. Figures 5.4, 5.5 and 5.6 show the tendency for each application to allocate the two special objects and derived types, and how they are distributed according to the *threshold* value set at the VM startup. The y-axis shows the amount of words in the heap that each BDA-region is occupying at the time of the snapshot (x-axis). The reasoning for this is that showing the percentage of occupation for each space can result in low values, which would be difficult to analyze and compare. This method also has the advantage of providing direct comparison with the *Region other* space which is where most of the objects end up.

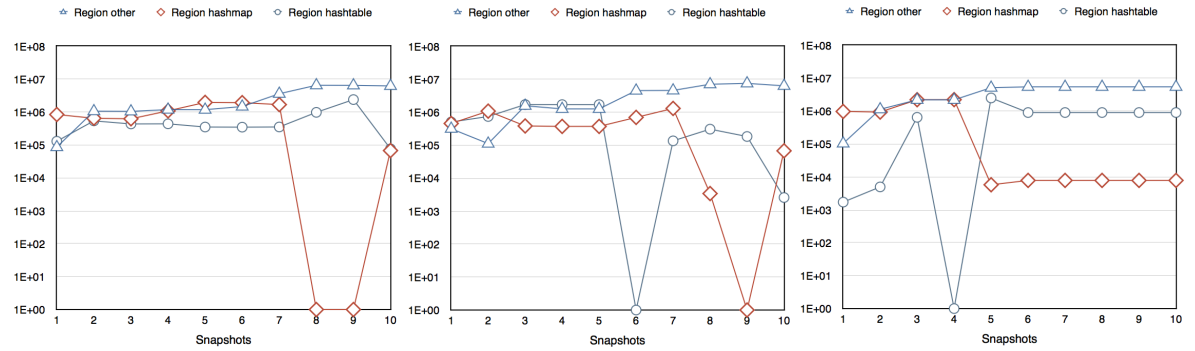
Figure 5.4 shows how the header region word behaves in H2. It can be observed that H2 allocates numerous `HashMap` and `Hashtable` objects and uses them for a short time. The sudden drops for the *Region hashtable* are not caused by the adjusting of spaces, because it can be seen in Snapshot 8 of the *threshold* 50% graph (b) that both *Region hashmap* and *Region hashtable* have their used space reduced after that collection. Analyzing the graph, it is seen that a *threshold* value of 70% increases the number of `HashMap` objects compared to the other runs, which indicates that these are objects with smaller size but in greater number.

On the other hand, Figure 5.5 shows that for Tradebeans there is greater stability when precedence to the larger objects is given, i. e., with a *threshold* value of 30%. It reduces the number of garbage collections and the reduction in used space only occurs in Snapshots 8 and



(a) Threshold 30% (b) Threshold 50% (c) Threshold 70%

Figure 5.4: Header Region word — H2 — Object locality (in heap words) in each BDA-region



(a) Threshold 30% (b) Threshold 50% (c) Threshold 70%

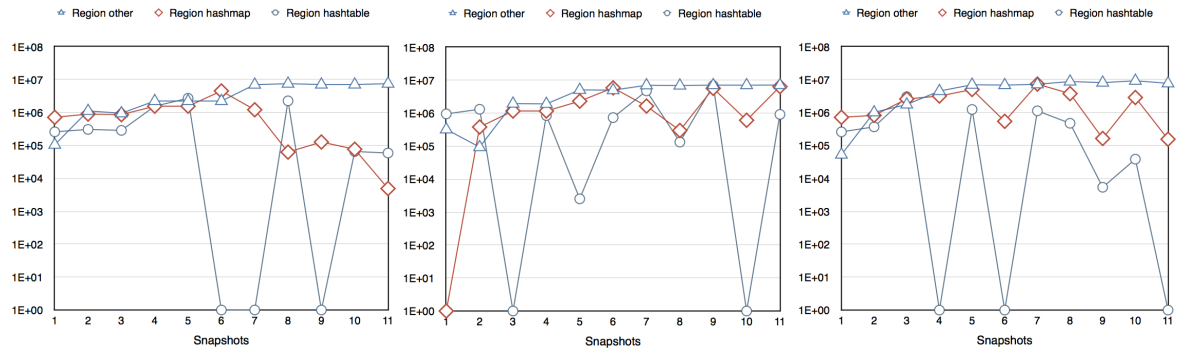
Figure 5.5: Header Region word — Tradebeans — Object locality (in heap words) in each BDA-region

9, in the *Region hashmap* space, because the *Region other* expanded onto it. On the other hand, with a *threshold* of 70% the Tradebeans benchmark demonstrates some stability in the end. Since precedence was given to smaller objects but in larger numbers, the *Region hashtable* gets most of the allocated special object words. This, in turn, is contrary to this work's goals which is to pack larger objects together in their appropriate memory regions.

The Tradesoap results in Figure 5.6 are very similar to the results of Tradebeans in Figure 5.5 for general-purpose objects and `HashMap`-based objects — a characteristic of the GERONIMO backend. The observed drops in the *Region hashtable* used space is due to the SOAP protocol use of `Hashtable`-based objects to execute the transactions.

5.3.2 Hash Klass pointer locality

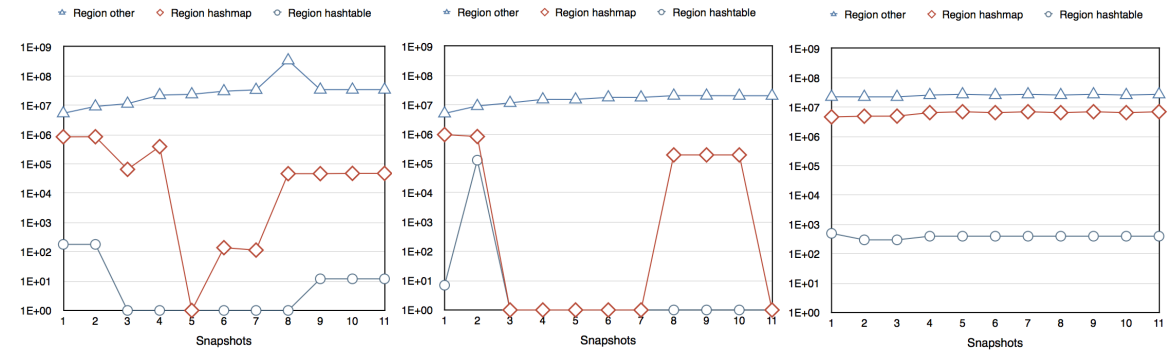
Contrary to the header region word, the hashing of the Klass pointer as a fast-type query mechanism is not always precise. However, it greatly reduces the memory footprint which could possibly reduce the number of triggered full collections.



(a) Threshold 30% (b) Threshold 50% (c) Threshold 70%

Figure 5.6: Header Region word — Tradesoap — Object locality (in heap words) in each BDA-region

In Section 5.2 it was shown that an array of 1200 entries yields sufficient correct results for fast-type queries. The results in Figures 5.7, 5.8 and 5.9 trace the behavior of the benchmarks using the same three *threshold* values used in the previous Subsection 5.3.1. Just as the previous graphs, the y-axis shows the used space, in words, for each BDA-region and the x-axis the snapshots of when the information was taken.

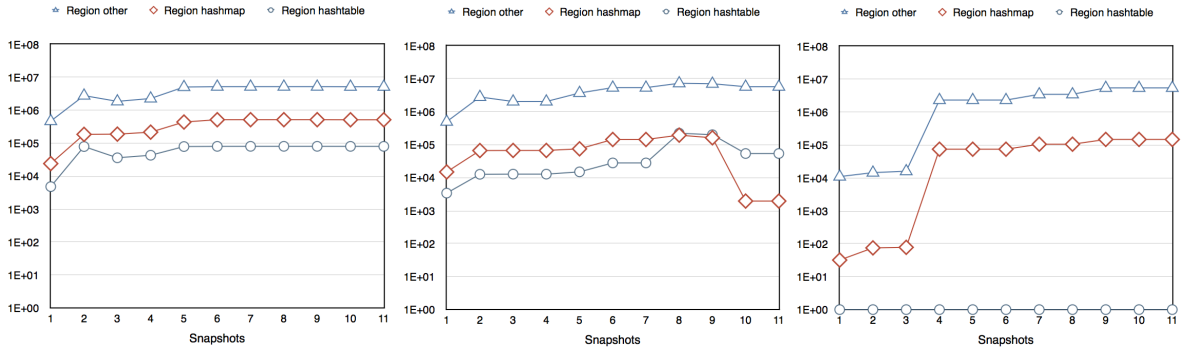


(a) Threshold 30% (b) Threshold 50% (c) Threshold 70%

Figure 5.7: Hash of the Klass pointer — H2 — Object locality (in heap words) in each BDA-region

It was previously seen that H2 has tendency to allocate many small `HashMap` and `Hashtable` objects. In Figure 5.7, due to collisions and the fact that a *threshold* of 30% is giving precedence to the larger objects, the *Region hashmap* gets most of the objects and *Region hashtable* fails to allocate until the last Snapshots. H2, as a database, has tendency to allocate many objects, unrelated to `Hashtable` and `HashMap`, part of its data structures. These general-purpose objects quickly fill the "other" BDA-region triggering collisions. On the other hand, if small `Hashtable` objects are placed on their correct BDA-region, they will no longer cause bloat on the *Region other* space, triggering less full collections and incurring less overhead and pause times for the application, fostering better performance. This allows the old generation to

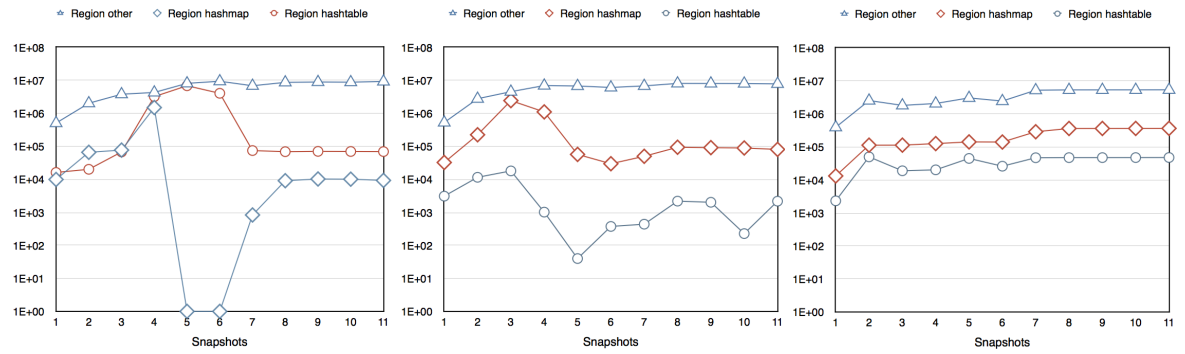
stabilize its spaces, maintaining a well adjusted capacity.



(a) Threshold 30% (b) Threshold 50% (c) Threshold 70%

Figure 5.8: Hash of the Klass pointer — Tradebeans — Object locality (in heap words) in each BDA-region

Figure 5.8 continues to demonstrate the tendency for the benchmark to allocate several large `HashMap`-based objects due to the GERONIMO backend. This is shown in the figures for the *threshold* of 30% and 50%. In fact, it triggers a very small number of full collections, with the given heap size. As for the figure that reports the *threshold* value of 70% experiment, `HashMap` and general-purpose objects got hold of most of the hash array entries, not giving any room `Hashtable`-based object. This would be solved with a slightly larger array size.



(a) Threshold 30% (b) Threshold 50% (c) Threshold 70%

Figure 5.9: Hash of the Klass pointer — Tradesoap — Object locality (in heap words) in each BDA-region

The Tradesoap benchmark allocates several `Hashtable`-based objects while executing its requests over SOAP. These `Hashtable` objects are small in size and short lived. Since they are high in number, they quickly set their corresponding slots in the hash array, allowing every other instance to be promoted to the correct BDA-region. This is demonstrated in all graphs of Figure 5.9. Given that `Hashtable`-based objects in Tradesoap are small but high in number, the figure for the *threshold* 70% yields much better results. The only collection triggered in this series of tests was when *Region hashmap* required more space and ended up expanding over the

Region hashtable.

5.4 Performance benchmark suite

The goal of this thesis is to prove that certain objects, such as collections, that cause memory bloat on the Java heap if separated into other spaces could improve performance in the long term from increased locality. To improve performance of the garbage collector, certain modifications in the access of fields would have to be made, such as on the tracing of all special object references. This is a matter not included in this work, however it is important to assess that the creation of segmented spaces does not increase the overhead of the garbage collector, due to the conditionals needed to promote objects accordingly. It would be noticeable for short-run applications such as the H2, Tradebeans and Tradesoap benchmarks. But, as Figure 5.10 illustrates, the execution times are very similar when compared with an unmodified HotSpot VM.

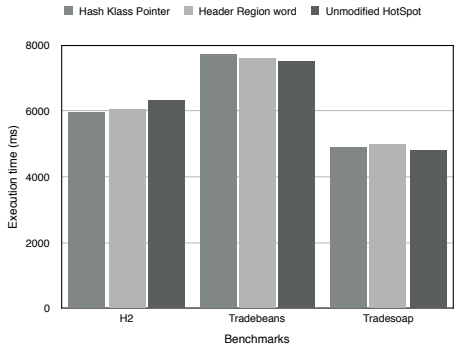
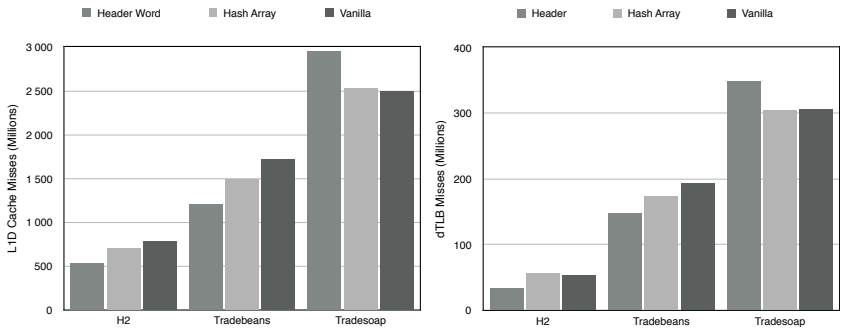


Figure 5.10: Execution times over the previous benchmarks in respect to the bloat-aware PS

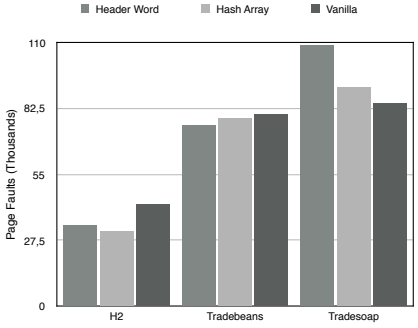
From Figure 5.10, it can be observed that this solution conditionals and query mechanisms do not incur latency. In fact, over the three versions of the HotSpot VM — (1) *BDA-heap* using the hashing of the Klass pointer, (2) *BDA-heap* using the header BDA-region word and (3) unmodified HotSpot — the execution times are equivalent. Given the high number of collections that each run triggers, it can be observed that the GC pauses (overhead) for a *BDA-heap* are not considerably larger, or else an increase over the execution times would happen. This balance (equivalency of the execution times) is the result of the increased locality, which reduces misses, at the L1-data cache and DTLB level, and page faults, such as shown in Graphs 5.11.

From analysis of the graphs, it can be observed the tendency for increased locality in respect to the type of benchmark in which it ran. H2 is a database benchmark, therefore it allocates more objects part of its data structures. When these dependent objects are kept together, locality over the same cache line can increase by 40% for precise object information (header word method) and by 10% on less precise object information (hash array method), as illustrated in Figure **??**. This in turn reduces accesses to the DTLB and to the Page Table as shown in Figures **??** and **??**. The spike in misses for the header word for BDA-region indexing on the



(a) L1D Cache Misses

(b) DTLB Misses



(c) Page Table Faults

Figure 5.11: Locality improvement over the L1 Data Cache, the DTLB and the Page Table

Tradesoap benchmark is a result of the usage of SOAP envelopes and an additional 8 bytes per object. Since objects such as SOAP envelope wrappers are generally large in size, with additional bytes per object it stretches the range of addresses they span, occupying large amounts of cache lines and virtual pages.

Summary

This chapter showed how this thesis's work implementation compares with an unmodified HotSpot running benchmarks used throughout the literature. It was demonstrated that the approach of segmenting the spaces, and making the default garbage collector aware of a new segmented heap, did not cause latency despite having conditionals and mechanisms in order to decide the target BDA-regions.

6 Conclusion

> Change is the law of life. And those who look only to the past or present are certain to miss the future. – *John F. Kennedy (1917–1963)*

6.1 Concluding remarks

This thesis related the design of an architecture for the Java heap to deal with the overhead caused by the memory bloat that Big Data applications cause, and its implementation. To sum up, on Chapter 1 the problem was introduced, showing what motivated this work and what it tries to address. On Chapter 2, it was presented the existing work in the literature that helped narrow down the solution. After, on Chapter 3, the specific details of this solution, the challenges and the difficulties that can arise when implementing were offered. Chapter 4 gave the exact details of the implementation of the solution and showed how to extend the Parallel Scavenge garbage collector. To confirm the predictions set for this work in the introduction, Chapter 5 demonstrated the result of running a modified version of the HotSpot JVM using decision mechanisms in order to assess the distribution of objects.

One of the most important factors was to not increase the application pause times, and consequently the application execution times, under the Parallel Scavenge STW collector. The presented architecture can, in fact, reduce the number of triggered GC by saving, in a safe heap region, objects that do not need to be collected as often. This reduces bloat in the old generation, because objects that do not need to be collected as often are not mixed with objects that after promotion, only survive for a small number of full collections.

It was believed that although some instrumentation code had to be added in order to promote objects accordingly, some performance would come out, such as balancing the execution times over short-run benchmarks due to the increased locality, which in turn reduces the load times for object references. That prediction was confirmed. The gains of locality can be more visible on applications that run for a long period of time and that execute many accesses to long-lived objects. Those gains can be even larger in NUMA architectures, due to the dominant nodes for a type of object. Such architectures would promote locality by keeping in their cache line local and remote references dependent of each other.

Thus, this work contributes with an architectural design for starting to develop bloat-aware heaps. It also contributes with techniques on how to promote on such heaps, decision mecha-

nisms and algorithms for adjustment of the heap, resultant of uneven allocation.

6.2 Future Work

The work just described is still not enough to provide performance for Big Data applications that run under a managed runtime. This is due to the garbage collection algorithm still not being completely aware of such segmented spaces.

Parallel Scavenge is a high-throughput parallel STW garbage collector. The parallelism it has can only increase a collection performance, which reduces latency. However, no matter how segmented is the space, the level of parallelism is always the same. Thus, this solution demands a specific garbage collector in order to efficiently collect on the BDA-regions it created. Also, the JVM could have profiling information, whether dynamic or static in order to know for which object types does a segmented space heap must create room for.

Here it is proposed that a hybrid GC, with parallel phases and concurrent phases, can greatly increase the overall GC performance if using this work's bloat-aware heap. A concurrent GC could be watching each BDA-region, profile about its use and collect each BDA-region separately. This would lead to high GC throughput in a subsequent collection.

On the other hand, it is important for the JVM to know which types it must handle differently. A profile-based dynamic approach would check for large object use and create heap spaces specific to those. Another way to deal with this is an API which would expose wrappers of types considered to cause bloat on the heap, e. g., hashmaps, hashtables, hashsets and linked lists. An easier approach would be to include global variables within the VM to define the klass type that is to be handled in the BDA-heap, during the launching the application.

Bibliography

Appel, A. W. (1989). Simple generational garbage collection and fast allocation. *Software: Practice and Experience 19*(2), 171–183.

Appel, A. W., J. R. Ellis, & K. Li (1988). Real-time concurrent collection on stock multiprocessors. In *ACM SIGPLAN Notices*, Volume 23, pp. 11–20. ACM.

Bacon, D. F., P. Cheng, & V. Rajan (2003). A real-time garbage collector with low overhead and consistent utilization. *ACM SIGPLAN Notices 38*(1), 285–298.

Bailey, C. (2012, February). From java code to java heap. http://www.ibm.com/developerworks/library/j-codetoheap/j-codetoheap-pdf.pdf.

Baker Jr, H. C. & C. Hewitt (1977). The incremental garbage collection of processes. *ACM SIGART Bulletin 12*(64), 55–59.

Baker Jr, H. G. (1978). List processing in real time on a serial computer. *Communications of the ACM 21*(4), 280–294.

Blackburn, S. M., R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, et al. (2006). The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, Volume 41, pp. 169–190. ACM.

Blackburn, S. M. & A. L. Hosking (2004). Barriers: friend or foe? In *Proceedings of the 4th international symposium on Memory management*, pp. 143–151. ACM.

Blackburn, S. M. & K. S. McKinley (2008, June). Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. *SIGPLAN Not. 43*(6), 22–32.

Boehm, H.-J., A. J. Demers, & S. Shenker (1991). Mostly parallel garbage collection. In *ACM SIGPLAN Notices*, Volume 26, pp. 157–164. ACM.

Bu, Y., V. Borkar, G. Xu, & M. J. Carey (2013). A bloat-aware design for big data applications. In *ACM SIGPLAN Notices*, Volume 48, pp. 119–130. ACM.

Chen, W.-k., S. Bhansali, T. Chilimbi, X. Gao, & W. Chuang (2006, June). Profile-guided proactive garbage collection for locality optimization. *SIGPLAN Not. 41*(6), 332–340.

Cheng, P. & G. E. Blelloch (2001, May). A parallel, real-time garbage collector. *SIGPLAN Not. 36*(5), 125–136.

Click, C., G. Tene, & M. Wolf (2005). The pauseless gc algorithm. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pp. 46–56. ACM.

Dean, J. & S. Ghemawat (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*(1), 107–113.

Detlefs, D., C. Flood, S. Heller, & T. Printezis (2004). Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, New York, NY, USA, pp. 37–48. ACM.

Detlefs, D. & T. Printezis (2000). A generational mostly-concurrent garbage collector.

Dijkstra, E. W., L. Lamport, A. J. Martin, C. S. Scholten, & E. F. Steffens (1978). On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM 21*(11), 966–975.

Gidra, L., G. Thomas, J. Sopena, & M. Shapiro (2013, March). A study of the scalability of stop-the-world garbage collectors on multicores. *SIGARCH Comput. Archit. News 41*(1), 229–240.

Hertz, M. & E. D. Berger (2005, October). Quantifying the performance of garbage collection vs. explicit memory management. *SIGPLAN Not. 40*(10), 313–326.

Hölzle, U. (1993). A fast write barrier for generational garbage collectors. In *OOPSLA/ECOOP*, Volume 93.

Hunt, P., M. Konar, F. P. Junqueira, & B. Reed (2010). Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, Volume 8, pp. 9.

Ilham, A. A. & K. Murakami (2011). Evaluation and optimization of java object ordering schemes. In *Electrical Engineering and Informatics (ICEEI), 2011 International Conference on*, pp. 1–6. IEEE.

Lakshman, A. & P. Malik (2010). Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review 44*(2), 35–40.

Laney, D. (2001). 3d data management: Controling data volume, velocity and variety. http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM 3*(4), 184–195.

Moon, D. A. (1984). Garbage collection in a large lisp system. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, New York, NY, USA, pp. 235–246. ACM.

Neumeyer, L., B. Robbins, A. Nair, & A. Kesari (2010). S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pp. 170–177. IEEE.

Ogasawara, T. (2009). Numa-aware memory manager with dominant-thread-based copying gc. In *ACM SIGPLAN Notices*, Volume 44, pp. 377–390. ACM.

Sakr, S., A. Liu, D. M. Batista, & M. Alomari (2011). A survey of large scale data management approaches in cloud environments. *Communications Surveys & Tutorials, IEEE 13*(3), 311–336.

Singer, J., G. Brown, I. Watson, & J. Cavazos (2007). Intelligent selection of application-specific garbage collectors. In *Proceedings of the 6th international symposium on Memory management*, pp. 91–102. ACM.

Sumbaly, R., J. Kreps, L. Gao, A. Feinberg, C. Soman, & S. Shah (2012). Serving large-scale batch computed data with project voldemort. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pp. 18–18. USENIX Association.

Tene, G., B. Iyengar, & M. Wolf (2011). C4: the continuously concurrent compacting collector. *ACM SIGPLAN Notices 46*(11), 79–88.

Toshniwal, A., S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. (2014). Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 147–156. ACM.

Ungar, D. (1984). Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM Sigplan Notices 19*(5), 157–167.

White, T. (2009). *Hadoop: the definitive guide: the definitive guide*. " O'Reilly Media, Inc.".

Wilson, P. R., M. S. Lam, & T. G. Moher (1991, May). Effective “static-graph” reorganization to improve locality in garbage-collected systems. *SIGPLAN Not. 26*(6), 177–191.

Yang, F., E. Tschetter, X. Léauté, N. Ray, G. Merlino, & D. Ganguli (2014). Druid: a real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 157–168. ACM.

Yang, X., S. M. Blackburn, D. Frampton, & A. L. Hosking (2013). Barriers reconsidered, friendlier still! *ACM SIGPLAN Notices 47*(11), 37–48.

Zhao, Y., J. Shi, K. Zheng, H. Wang, H. Lin, & L. Shao (2009). Allocation wall: a limiting factor of java applications on emerging multi-core platforms. *ACM SIGPLAN Notices 44*(10), 361–376.