# Garbage Collection Optimization for JVM running Big Data Workloads

Duarte Patrício
Computer Science and Engineering Department
Instituto Superior Técnico
Avenida Rovisco Pais, 1
Email: duarte.carvalho@tecnico.ulisboa.pt

***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 required by a managed runtime. This overhead is negligible for small server applications but not for large scale data processing. The JVM is the main bottleneck for all Java programs. If a program misbehaves, the JVM should take appropriate action. Large scale data analysis programs misbehave in terms of the memory bloat they cause. The problem is in the huge amount of objects and collections of objects that these programs allocate.**

**This thesis's work presents a solution to deal with the memory bloat that large objects cause on the Java heap, for the HotSpot Java Virtual Machine. The solution consists on seperating 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 virtual machine, the Parallel Scavenge, to collect on such heap and promote old objects accordingly.**

**It is shown that this solution can provide better application throughput when the spaces for the large objects contain long lived references, that managed to maintain dependent object references close together.**

## I. Introduction

Many of the Big Data applications in use today are written in the Java language. The Java language facilitates application development because it relies on the Java Virtual Machine (JVM) for its memory-management, it has a flexible object-oriented design, quick release cycle, portability and worldwide support. However, the JVM can be a bottleneck for programs that rely on huge memory usage, such as Big Data Applications. This is due to the allocation and collection mechanisms, which are still not fully aware of the large memory footprint under Big Data.

The two culprits for the limited performance in some Big Data installations for memory restrained systems is the application itself and the JVM. Tuning the application code is impractical given the increasing number of applications. This leaves the JVM as the principal driver.

On the JVM, all objects go into the Java heap, a runtime structure closely managed by the GC module. Management of the heap consists on interpreting allocation calls and triggering a garbage collection when the spaces are exhausted. Therefore, the GC module has great impact on the throughput of the application requiring adaptation to become *bloat-aware*.

From the works of [1], [2] it was shown that *bloat-awareness* can be achieved optimizing locality for the objects in the heap. In fact, Moon's work [1] 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, Wilson *et. al.* demonstrated reorganization techniques to improve locality showing that there is improvement on the traversal algorithm when objects of a certain type are given special treatment.

This work consists on optimizing the Parallel Scavenge garbage collector of the HotSpot VM by adapting the Java heap to give special treatment to certain objects, considered to be possible candidates for causing memory bloat, and to garbage collect such heap.

Duarte Patrício
October 16, 2015

### A. Contributions

The main contributions of this paper are:

- An extension for the Java heap which handles potential 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.

## II. Related Work

This section presents the most relevant work regarding this work'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. Section II-A describes the relevant Big Data platforms that could improve under this work's solution, Section II-B describes relevant GC algorithms available in the literature, and in Section II-C some lower level details for various optimization levels are shown.

### A. 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**, did not quite fit for the storage of data, due to their lack of scalability, throughput and dynamism. These kind of databases are commonly referred to as *NoSQL* (from "Not Only SQL").

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, thus trusting in the distributed computing design of the applications for performance.

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. Voldemort [3] is a Key-Value storage in use by LinkedIn, based on Amazon's DynamoDB, which offers high availability and quick lookups, since it is, essentially, a huge distributed hash table (DHT).

On the other hand, column-oriented databases are contrary to the conventional **RDBMS**, and store their data in columns rather than in rows. This provides faster lookups in disk when reading an large set of blocks. It is a popular data orientation in NoSQL data-stores. Examples of implementations include HBase [4], Cassandra [5] and Druid [6].

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 of data. Batch processing in Big Data involves the scheduling a "job" as its workload, such as the Hadoop Map-Reduce framework [7]. However, some applications require real-time guarantees for service. These are stream-processing applications, such as the S4 [8] and Storm [9].

### B. Garbage collection

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 [10].

Classic algorithms, such as the Mark&Sweep [11], the Copying GC [12], the Incremental GC [13] and the Generational GC [14] set the ground for the more modern algorithms such as the *parallel* (several GC threads executing at the same time) and the *concurrent* (GC threads execute at the same time of the *mutator* — the application thread). Examples of relevant parallel collectors include the Immix [15], a mark-region GC with contiguous allocation and opportunistic evacuation; the Parallel Scavenge and NAPS (NUMA-Aware Parallel Scavenge) [16], a generational mark-sweep-compact GC implemented in the HotSpot JVM where NAPS is an adaptation in order to make it NUMA-Aware; and NUMA-Aware Dominant-thread-based copying [17], a copying GC that uses the thread that most accesses the object on a NUMA architecture. Concurrent collectors try to mitigate the Stop-The-World (STW) pauses caused by the parallel collectors. In this category are included the Real-Time Concurrent GC [18], which uses page virtual protection to raise traps and keep up with the mutator; the Mostly-Parallel GC [19], which is essentially a concurrent Mark&Sweep using virtual dirty bits to track the mutator's touched pages; the Pauseless GC [20], also a concurrent Mark&Sweep but with the mutator doing GC work; and the C4 [21], a generational extension of the Pauseless GC.

### C. Object Layout and Locality

The JVM keeps some "behind-the-scenes" structures in order to fulfill its manager responsibility. These include object metadata, indispensable for the garbage collector, whose size depends on the VM implementation [22].

Object ordering schemes can provide performance for the application, by leveraging their placement in order to increase locality in system-level memory structures. Ilham *et. al.* studied and evaluated *Depth-First*, *Breath-First* and *Hot-Depth First* ordering schemes, while prior works, such as those by Moon [1] and Wilson [2], showed that optimization can be achieved by separating *long-lived* objects of ephemeral ones or by categorizing objects depending on their type. On the other hand, Chen *et. al.* [23] instrumented object code in Microsoft's CLR [1] to create profiles of object usage.

## III. Architecture

This section describes the main architecture for the proposed solution, taking into account the design goals of the HotSpot virtual machine.

The following sections are organized following an approach of increasing detail. First, in Section III-A an overview of the HotSpot virtual machine runtime is presented, and then in Section III-B the target GC module is described, at high-level. The last Section III-C will describe, this time with more detail, the decisions and the approach taken.

### A. HotSpot Architecture Overview

The HotSpot virtual machine is composed of several components. For this work, the Parallel Scavenge GC module (one of the three currently implemented garbage collectors) is the most direct intervenient. The Parallel Scavenge code includes classes for GC tasks, garbage collection managers, allocation buffers and the heap itself. Since each component depends on the other, many of those require extensive modification or need their current implementation extended. As for the rest of the classes, like `Klass` and `oopDesc`, who are determinant for the correct functioning of the VM, also need their implementation extended.

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

In the HotSpot VM 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.

### *B. Garbage collection — Parallel Scavenge*

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. However, Lokesh Gidra *et. al.* studied the scalability of the Parallel Scavenge collector and described its internals [16].

The GC in use is what defines the heap layout. This is due to the GC's implementation definition of how the heap should be structured in order to maximize its throughput when collecting. The efforts in this work are set to the `ParallelScavengeHeap`. Parallel Scavenge is a throughput-oriented garbage collector, more specifically it is a Stop-The-World, generational parallel collector. The Parallel Scavenge heap is separated into two generations, the *young generation* and the *old generation*. The *young generation* contains the *eden-space* 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. This layout is illustrated in Figure 1. 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. Its allocation scheme uses a *bump-pointer* technique, a pointer called the *top-pointer*. Allocation spaces, such as those belonging to the young and old generations, base their allocation on a `MutableSpace` object, which handles the atomic update of the top-pointer and sets the allocation boundaries. A `MutableSpace` layout is illustrated in Figure 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. 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) 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).

To mark the objects, the VM thread sets steal tasks and marking tasks. Marking tasks follow the root-set, composed of the threads' stack frames, registers, global and static variables, and the Old-To-Young root objects, i. e., objects in the old generation that reference young generation objects. The Old-To-Young root-set is composed by objects contained in dirty cards, in the write-barrier card table. The write-barrier set is a division of the space into fixed-sized *cards* of $2^9$ words. The cards are dirtied in a similar fashion as Urs Hölzle's method [24]. On the contrary, the steal tasks are to prevent idle GC threads. These make the idle thread to search for, and dequeue, tasks in overloaded queues belonging to other GC threads. If it is unsuccessful in stealing then it enters a *termination protocol*.

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. 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.

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.

*2) Full collection:* If a young collection was unsuccessful in promoting all its objects, or a *System.gc()* is issued, then a full collection takes place. The algorithm that collects all of the generations is a *Parallel Mark-Sweep-Compact*, hereafter simply referred to as Parallel Compact, and is part of the Parallel Scavenge and the default for any multi-core machine. Its work is divided in four phases: marking phase (Section III-B2a), summary phase (Section III-B2b), compacting phase (Section III-B2c) and clean up phase (Section III-B2d). 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.

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), hereafter referred to as *region*. A region is essentially a wrapper around an interval of addresses. This window is its base unit of work.

*a) Marking phase:* The first phase, as the name suggests, comprehends marking the live objects in parallel. Whereas in the young collection the scavenging tasks promote objects, in the full collection the marking tasks mark the root objects' start and ending, add its live data amount to the compacting regions and push the reference onto the marking 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.

*b) Summary phase:* When marking is finished, the VM thread can regain control and start the summary phase. The summary phase concept is to calculate the destination of the compacting regions. 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

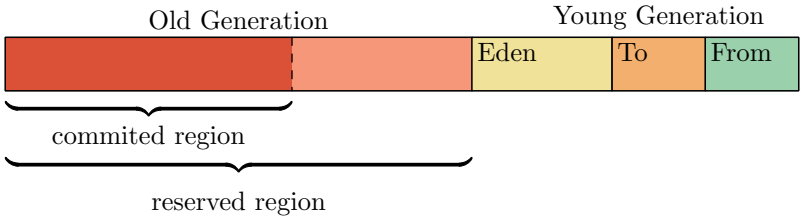


Fig. 1. Parallel Scavenge heap layout

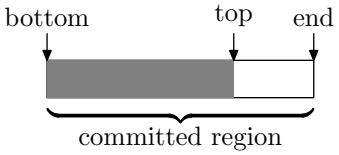


Fig. 2. A space's layout

will copy data to its location. 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. Quick compacting is to calculate the amount of live data by calculating the new top-pointer for each space. Here, the *dense-prefix-end*, which marks the start of the first compacted region, i. e., all space to the left is full, is calculated. The *dense-prefix-end* address is always on a region boundary, such as illustrated in Figure 3. The second part is to summarize all the regions in the old space past the dense-prefix-end. The last summary step is to summarize all young generation spaces. It tries to move the young spaces regions to the old space. Summarizing a space consists on iterating over all its regions. Each iteration checks if there is live data, sets the destination address if it contains data, by sliding into the next top-pointer of the target-space, and increment the *destination-count*. The destination-count is a value that indicates the number of regions that the region under processing will span when copied (a maximum of 2).

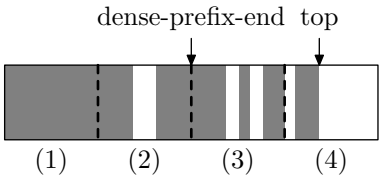


Fig. 3. Summary phase — Dense-prefix-end and current top pointer

*c) Compact phase:* Compaction is done in parallel such as marking. It consists on the filling of regions with source words targeted for them. A region that contains work to do is one that can be *claimed*, i. e., it has no destination (destination-count is zero). The filling of a region consists on a series of steps: get the source's first word targeted for the region; iterate from first word to the end of the source region, copying objects while they fit in the target; if the source region was fully processed (it contains no more words to copy), and there's still room in the target region, then fetch a new source region (adjacent to the former), switching source spaces if necessary. Every time a source region is fully processed, its destination-count is decremented and, if it can be claimed by a GC thread (the destination-count is zero), it is added to the queue to be filled. After compaction, the effective result looks like as shown in Figure 4.

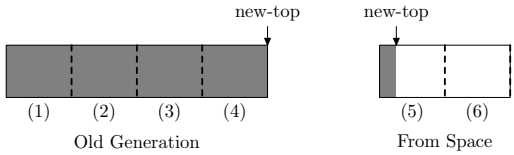


Fig. 4. Final view of the summary phase and final effective result after compaction

*d) 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.

### C. From Parallel Scavenge to Bloat-Aware PS

This Section introduces a new design for the Java heap, aware of the bloat that collections cause. This new design is targeted for the `ParallelScavengeHeap`, thus it is hereafter termed as *bloat-aware PS*. 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. The collections chosen to be handled differently were the `java.util.HashMap` and `java.util.Hashtable` packages, 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. *Target-objects* are defined to be those that have tendency to cause memory bloat or disperse throughout the heap, on the long term. Target-objects include two characteristics: are *long-lived* and large enough. The old generation spaces are the ones that contain long-lived objects, thus making the young generation irrelevant for target-objects. Therefore, the old generation is the most relevant space

to track target-objects, and can be split such as illustrated in Figure 5.

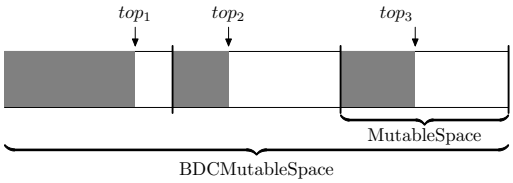


Fig. 5. The partitioned bloat-aware old generation

The *BDCMutableSpace* (named from "Big Data Collections Mutable Space") is a subclass of the `MutableSpace` class in order to maintain transparency to the VM. This new space contains an inner class that represents each *BDA-region*, which is a fragment of the old-space.

*1) 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 comparison with other objects, computation of the values to consider and the computation of the sizes for each object and its references. Inferring through the object type is faster and allows grouping objects of the same type.

Each application has a number of favorite Java collections it uses for processing their main jobs. These collections tend to disperse their references throughout the heap at some point in time and, for the cases when they must increase their capacity, it can generate memory bloat. Objects of those types should be identified, preferably using a fast mechanism to reduce latency during promotion. Objects identified as non-critical should be identified as "other", i. e., an ordinary object, while objects of the some relevant types should be identified using the package that implements them. Since this work's special-objects are `HashMap`-based and `Hashtable`-based, they are identified as "hashmap" and "hashtable", respectively. For saving this information, this work implements two methods of type-addressing: a new header word and hashing of the klass pointer.

***Header region mark:*** An added word gives great 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 region-space identifiers. However, Figure 6 shows that this methodology has a larger footprint on the memory, which can be significant for applications that run for a long time. As an advantage, the information about the object is always precise.

***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 add 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. Since latency must be reduced to a minimum, there can be no collision mechanism. Figure 7 shows three objects with their type in plain text, accessing the array of entries to get their target BDA-regions.

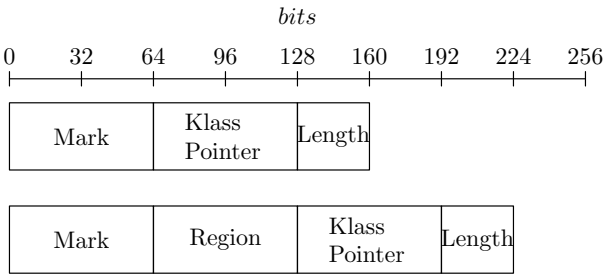


Fig. 6. Layout of an array object header with an added word for region-space indexing

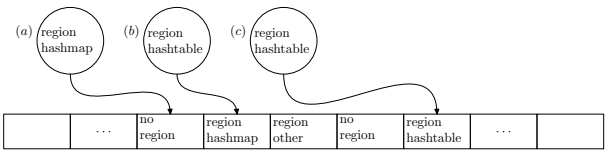


Fig. 7. An array of entries with region-space identifiers

*2) Adjusting the space:* When the old space is initialized there is no information on how to divide the region-spaces, and thus are divided evenly. However, allocation on specific BDA-regions can be uneven, unbalancing the space quite significantly. Hence, the segmented space must adjust its BDA-regions according to their occupation and free ratios. Algorithm 1 shows how the adjustment is computed. 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 (the value of 80% is predefined);
2) Compute the expected expand size;
3) Find a forwarding neighbor (the spaces never expand below the bottom pointer) based on:
   - 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: 9–11), 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 (line: 14–15) — 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.

This algorithm can not run during Old-To-Young root tasks because it moves the boundary pointers (*bottom* and *end*).

### D. Bloat-aware young collection

Most allocation in the old space happens with promotion during a collection; an exception to this rule is direct allocation for objects that still do not fit in the eden space after a young collection. So far, a design for a split heap by object type has been given (Section III-C), but the allocation of objects

**Algorithm 1** Adjusting of BDA-regions

Find $max$ occupying BDA-region such that $occupy\_ratio > 80\%$
$o_0 \leftarrow$ occupy ratio of $max$
$f_0 \leftarrow$ free ratio of $max$
$e \leftarrow o_0 \times multiplier \times MinRegionSize$ {Amount to expand}
**for** $n \leftarrow max + 1$ **do**
  $f_1 \leftarrow$ free ratio of $n$
  $s_1 \leftarrow$ free size of $n$
  **if** $f_1 - f_0 > difference$ **then**
    **if** $s_1/2 > e \geq MinRegionSize$ **then**
      $e \leftarrow s_1/2$ {Give-up half the space}
      **break**
    **end if**
  **else**
    **if** $s_1 > e$ & $s_1 - e > (j-i) \times MinRegionSize$ **then**
      **break**
    **end if**
  **end if**
**end for**
**if** no neighbor was found **then**
  **return** $false$ {No one has enough space}
**end if**
resize($max$, $e$)
**return** $true$

is still missing. This section fills in that hole. As explained in Section III-B1, the young collection operates on the eden and from spaces and, if possible, it tries to promote objects into the old-space. For the BDA-heap, this can raise some problems in the decision mechanisms and in the scanning of the Old-To-Young roots. The next paragraphs explain how to cope with this difference.

***Old to Young Root tasks:*** The Old-To-Young root tasks have a stripe number assigned, 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 and its size is fixed. The process of finding the Old-To-Young root-set consists on iterating the assigned stripe for all slices. Each iteration consists on: adjusting the first and the last card for a given stripe according to the first object that starts in the range and the last word for the last object in the range (objects cannot cross the start nor the end boundary points); find a range of dirty cards; clean those cards; and push those references into the promotion manager queue. To adjust the card range, an *object-start-array* is used. It is a byte array divided by $2^9$ byte blocks, were each byte is an offset for the last object in that block. Every object allocated in the old generation must have its offset, from the start of the block, set in the object-start-array. Figure 8 illustrates the finding of an object start using the *object-start-array* when on a split heap. The dashed lines represent cards. It does so by traversing backwards the *object-start-array* blocks.

With the BDA-heap, the old space is now split into BDA-regions, each with its own top-pointer and bottom-pointer.

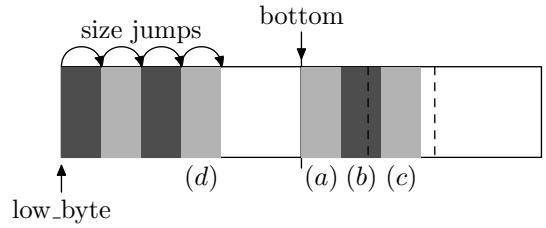


Fig. 8. Object-start-array fetch of invalid addresses

Thus, the space below a region's bottom-pointer contains the unallocated space that belongs to the previous BDA-region. The *object-start-array* is only aware of the bottom of the old generation (the $low\_byte$ in Figure 8) to limit its searches. Therefore, when doing a backwards search it may cross the BDA-region boundary and start fetching invalid addresses. To solve this, each space must be scavenged separately and each BDA-region's bottom-pointer must be passed to the methods that find the objects start address. It is also important to save the top-pointers of all the BDA-regions, in order to prevent the scavenging threads to transverse onto the currently allocated PLAB.

***Promoting by type:*** The bloat-aware heap expects that objects belonging to a certain type are allocated in the correct BDA-regions. Promoted objects are allocated from the PLAB (Section III-B1) and each GC thread allocates its own, as many times it needs. Therefore, each GC thread must have a number of old generation PLAB equal to the number of BDA-regions. On the other hand, each PLAB is allocated through a virtual call shared by all the threads in the VM. This gives no flexibility when allocating the PLAB, because the old-space does not know in which BDA-region is to allocate the requested PLAB. The approach taken was to use the thread to proxy the BDA-region information retrieved in an earlier step using one of the devised mechanisms presented in III-C1.

### *E. Bloat-aware full collection*

A full collection no longer relies on the old-space virtual calls (thread type proxying) and promotion 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-aware BDA-heap. The order of the paragraphs follow the same order that the collection steps take.

***Marking — decision information:*** As said earlier, 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 one of the fast-klass mechanism (Section III-C1). It is possible to execute a fast klass-query when objects are marked and decide based on the result. Therefore, each region has two additional fields for each 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 will then be used at the summary phase, when deciding the destination of the regions. This work gave focus to the `HashMap` and `Hashtable` collections,

thus following this example a compacting region contains 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, each BDA-region first compute their new top-pointer values (the quick compaction step). Then, each BDA-region computes its *dense-prefix-end* and summarizes its regions for compaction onto themselves. Then, on the third step, the summary of young spaces must target all the BDA-regions. Objects are not promoted individually, being instead bundled on a region. Some decision mechanism is needed in order to compute the target BDA-region for each region in the young space. On the marking phase, the information about the special objects was already computed. Hence, in each iteration of the young space's regions (Section III-B2b), Algorithm 2 decides which BDA-region will set the region's destination field. This algorithm decides based on a given *threshold* (set at the VM startup) which weights the ratio of the number of objects in each BDA-region in order to choose a decision metric. The decision metric is one of two: the count of objects for each space (line: 9–13), or the average size for each special object (line: 15–19).

**Algorithm 2** Decision algorithm for target BDA-region

```
1:  avg_hashmap ← hashmap_totalsize/hashmap_count
    {Average size of hashmap elements}
2:  avg_hashtable ← hashtable_totalsize/hashtable_count
    {Average size of hashtable elements}
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. If a new source region needs to be fetched it is done by iterating through the next adjacent source regions and find the first that is not empty. However, now there are several old-space partitions and, as illustrated in Figure 9, regions are targeted to other spaces. A "stealing" of a region can occur if a thread that did not fill a region completely fetches an adjacent source region that was targeted for another BDA-region. This can cause wrong word placement and a negative *destination-count*. To avoid the iteration of source regions in an intrusive way, the algorithm must check if a destination of a region corresponds to the expected destination space.

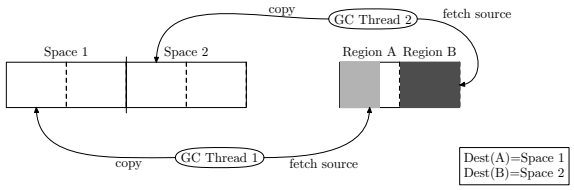


Fig. 9. Compaction — fetching and filling of regions

***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 dead-space. As referred in Section III-C, the BDA-heap 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 region-space, i. e., the higher address of them all.

## IV. Implementation

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 modified in order to implement different strategies, such as fast fast-klass queries with the two devised mechanism, was guarded by C pre-processor directives. This significantly reduced the amount of if-like conditions.

### A. Partitioning the old-space

The partition of the old-space went by sub-classing the `MutableSpace` into a `BDCMutableSpace`. A `BDCMutableSpace` acts as a container for the `CGRPSpace` class, which represents a BDA-region.

Allocation in the segmented old-space requires, by the space's command, a target BDA-region and the updating of the whole space top-pointer (to reflect that all space below contains live data).

### B. Fast Klass information

Section III-C1 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 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 it represents its Klass name. Therefore, for a `HashMap` Java object its `Symbol` representation would return a *java/util/HashMap* string.

*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 bloat-aware heap. It was implemented using the `regionMarkDesc` class, which describes a word that is added to the header of every object reference.

*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. 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.

### C. Bloat-aware young collection

With split spaces for keeping target-objects separated by type, the object promotion code requires some changes to cope with this demand. This is specially true for methods that allocate on the PLAB, which now they must first read the object's information (whether on the header or by consulting the hash array) and then allocate on the appropriate PLAB for that object. On the other hand, Old-To-Young root tasks must keep more top-pointers to avoid scanning the PLAB area. This was achieved by implementing a new class, the `BDACardTableHelper` class ("Big-Data Aware Card Table Helper"). This class not only saves the top-pointers but also the bottom-pointers, to prevent the traversing of the *object-start-array* to invalid addresses.

### D. Bloat-aware full collection

Just as the young collection, the full collection algorithms needed their implementation extended, specially because they deal with the whole old-space which is now split. For example, for the marking stage, it was stated that each region needed certain statistical information about the target-objects it wraps. This included adding methods and fields to the `RegionData`, a class that implements each region. This statistical information is then important for the summary stage, which in its summarize methods — a new method was added, called `summarize_parse_regions` — it decided the target space for each region it iterated on. After the summary phase, it was found that the fetching of adjacent source regions was unaware that some regions were targeted to other old-space partitions. This required the addition of one more condition during the iteration of new source regions, which checks if the destination space is the one it is expecting to be. And, in the last stage, the delegation of new top-pointers to the respective BDA-regions was simply implemented by packing every space identifier in a cycle.

## V. Evaluation

This section shows the results obtained in achieving better locality using the split spaces approach. The results were obtained using the DaCapo benchmark suite [25] on a 8-core machine with 12GB of memory. All runs were executed on 12 iterations with a minimum heap size of 1.5GB and a maximum of 8.5GB. The hash array size used was of 1200 entries, which is enough to cover all classes created in each run, and the *threshold* value was of 30%, i. e., precedence is given to larger objects. This section is organized as follows: In V-A a study of how the H2 benchmark behaves when facing a split space using the two devised mechanisms (Header word and Hash array), and in V-B how much locality was achieved and how this is affecting overall performance.

### A. Evaluating Object 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. Figure 10 shows the tendency for the H2 benchmark to allocate the two special objects and derived types. 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.

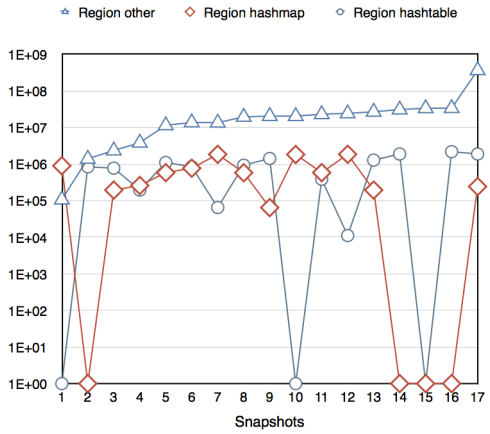


Fig. 10. Header Region word with threshold of 30% — H2 — Object locality (in heap words) in each region-space

In Figure 10 it can be observed that H2 allocates several `HashMap` and `Hashtable` objects and uses them for a short time. This is due to the DaCapo's benchmarks short runs, which give more impact on the garbage collection times and less on the execution itself. 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% (b) that both *Region hashmap* and *Region hashtable* have their used space reduced after that collection.

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 reduces the number of triggered full collections, as seen in Figure 11.

In Figure 11, 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

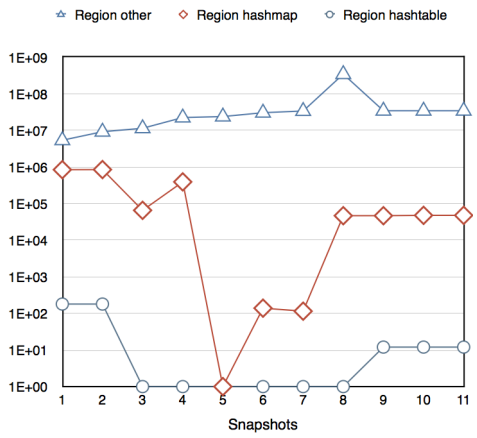


Fig. 11. Hash of the Klass pointer with threshold of 30% — H2 — Object locality (in heap words) in each region-space

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, making the occupation rates of the BDA-regions more stable.

### B. Performance benchmark suite

Although execution performance may not return direct results by using short-run applications, it is important to assert that the BDA-heap's instrumentation does not increase the execution time of the applications. Figure 12 shows the execution times, after 12 iterations, for each of the indicated benchmarks. It can be seen that this solution's conditionals and query mechanisms do not incur latency, when compared with the unmodified HotSpot VM. In fact, the execution times are equivalent, i. e., within $-2.30\%$ and $+3.20\%$.

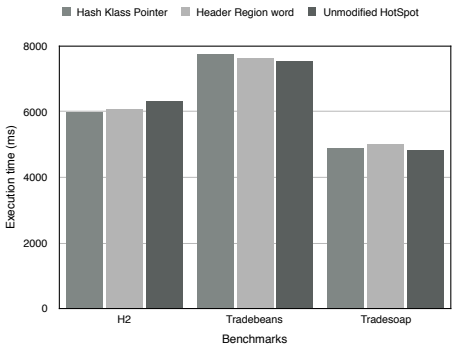


Fig. 12. Execution times over three benchmarks — H2, Tradebeans and Tradesoap — in respect to the BDA-heap

However, the execution times are still not low enough to consider the BDA-heap as a possible solution for the problem introduced. As stated earlier, this is because of the DaCapo benchmarks way of allocating the objects and accessing them at most one time. If they allocated long-lived objects more often, performance would be visible due to increased locality, as observed in Figures 13, 14 and 15.

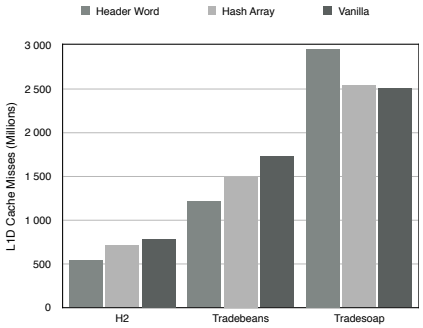


Fig. 13. L1 Data (L1D) Cache Misses

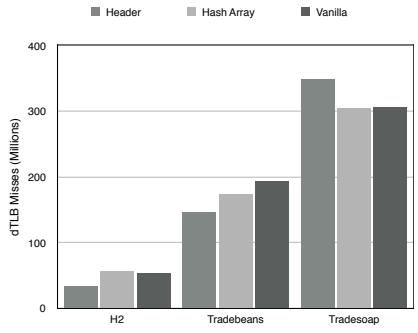


Fig. 14. Data TLB (DTLB) Misses

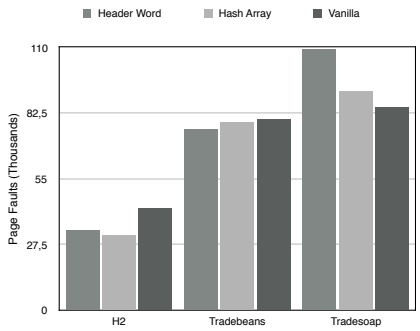


Fig. 15. Page Table faults

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 13. This in turn reduces accesses to the DTLB and to the Page Table as shown in Figures 14 and 15. The spike in misses for the header word for BDA-region indexing on the 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.

## VI. Conclusion

One of the most important factors in this work was to not increase the application pause times, and consequently the application execution times, under the Parallel Scavenge garbage collector, and increase the locality of objects for faster accessing by the CPU. This, in turn, 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 promote objects accordingly, some performance can be visible. This is seen on the balance between the increased locality effects and the instrumentation code, which slightly adds more CPU instructions during collection. It is believed that for an application that runs for a long period of time and that accesses a great amount of collection objects that previously allocated, can execute its workloads faster due to the increased load times, result of increased locality on system-level memory structures.

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

## VII. Future Work

There are two downsides with this approach. The garbage collection (Parallel Scavenge in this case) collects all the BDA-regions at one time. There are times when the GC is not needed for a specific BDA-region (for example, one BDA-region contains a huge number of collection objects that are to stay alive and do not need scanning). It would be a good idea to adapt the Parallel Compact, i. e., the old generation collector to collect BDA-regions separately depending on the need of each. This would greatly reduce the number of triggered GC. On the other hand, the object types that are available are still statically defined. A good approach would be to profile the application during run-time, or prior to the effective execution, to find which group of objects have the most tendency to stay in the heap and that increasing the locality between their dependencies would return improvements. Also, an easier approach would be to create VM global variables for the klass types to treat differently, that could be set at launch.

## References


[1] D. A. Moon, "Garbage collection in a large lisp system," in *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, ser. LFP '84. New York, NY, USA: ACM, 1984, pp. 235–246. [Online]. Available: http://doi.acm.org/10.1145/800055.802040

[2] P. R. Wilson, M. S. Lam, and T. G. Moher, "Effective “static-graph” reorganization to improve locality in garbage-collected systems," *SIGPLAN Not.*, vol. 26, no. 6, pp. 177–191, May 1991. [Online]. Available: http://doi.acm.org/10.1145/113446.113461

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

[4] T. White, *Hadoop: the definitive guide: the definitive guide*. " O'Reilly Media, Inc.", 2009.

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

[6] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli, "Druid: a real-time analytical data store," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 157–168.

[7] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[8] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*. IEEE, 2010, pp. 170–177.

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

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

[11] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part i," *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, 1960.

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

[13] H. C. Baker Jr and C. Hewitt, "The incremental garbage collection of processes," *ACM SIGART Bulletin*, vol. 12, no. 64, pp. 55–59, 1977.

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

[15] S. M. Blackburn and K. S. McKinley, "Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance," *SIGPLAN Not.*, vol. 43, no. 6, pp. 22–32, Jun. 2008. [Online]. Available: http://doi.acm.org/10.1145/1379022.1375586

[16] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro, "A study of the scalability of stop-the-world garbage collectors on multicores," *SIGARCH Comput. Archit. News*, vol. 41, no. 1, pp. 229–240, Mar. 2013. [Online]. Available: http://doi.acm.org/10.1145/2490301.2451142

[17] T. Ogasawara, "Numa-aware memory manager with dominant-thread-based copying gc," in *ACM SIGPLAN Notices*, vol. 44, no. 10. ACM, 2009, pp. 377–390.

[18] A. W. Appel, J. R. Ellis, and K. Li, "Real-time concurrent collection on stock multiprocessors," in *ACM SIGPLAN Notices*, vol. 23, no. 7. ACM, 1988, pp. 11–20.

[19] H.-J. Boehm, A. J. Demers, and S. Shenker, "Mostly parallel garbage collection," in *ACM SIGPLAN Notices*, vol. 26, no. 6. ACM, 1991, pp. 157–164.

[20] C. Click, G. Tene, and M. Wolf, "The pauseless gc algorithm," in *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*. ACM, 2005, pp. 46–56.

[21] G. Tene, B. Iyengar, and M. Wolf, "C4: the continuously concurrent compacting collector," *ACM SIGPLAN Notices*, vol. 46, no. 11, pp. 79–88, 2011.

[22] C. Bailey, "From java code to java heap," http://www.ibm.com/developerworks/library/j-codetoheap/j-codetoheap-pdf.pdf, Feb. 2012.

[23] W.-k. Chen, S. Bhansali, T. Chilimbi, X. Gao, and W. Chuang, "Profile-guided proactive garbage collection for locality optimization," *SIGPLAN Not.*, vol. 41, no. 6, pp. 332–340, Jun. 2006. [Online]. Available: http://doi.acm.org/10.1145/1133255.1134021

[24] U. Hölzle, "A fast write barrier for generational garbage collectors," in *OOPSLA/ECOOP*, vol. 93, 1993.

[25] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer *et al.*, "The dacapo benchmarks: Java benchmarking development and analysis," in *ACM Sigplan Notices*, vol. 41, no. 10. ACM, 2006, pp. 169–190.