Locality-Aware GC Optimisations for Big Data Workloads

Duarte Patrício[1,2], Rodrigo Bruno[1,2] José Simão[1,3], Paulo Ferreira[1,2], and Luís Veiga[1,2]

[1] INESC-ID Lisboa
[2] Instituto Superior Técnico, Universidade de Lisboa
[3] Instituto Superior de Engenharia de Lisboa, Instituto Politécnico de Lisboa
{dpatricio,rbruno}@gsd.inesc-id.pt
jsimao@cc.isel.ipl.pt {paulo.ferreira,luis.veiga}@inesc-id.pt

Abstract. Many Big Data analytics and IoT scenarios rely on fast and non-relational storage (NoSQL) to help processing massive amounts of data. In addition, managed runtimes (e.g. JVM) are now widely used to support the execution of these NoSQL storage solutions, particularly when dealing with Big Data key-value store-driven applications. The benefits of such runtimes can however be limited by automatic memory management, i.e., Garbage Collection (GC), which does not consider object locality, resulting in objects that point to each other being dispersed in memory. In the long run this may break the service-level of applications due to extra page faults and degradation of locality on system-level memory caches. We propose, LAG1 (short for Locality-Aware G1), an extension of modern heap layouts to promote locality between groups of related objects. This is done with no previous application profiling and in a way that is transparent to the programmer, without requiring changes to existing code. The heap layout and algorithmic extensions are implemented on top of the Garbage First (G1) garbage collector (the new by-default collector) of the HotSpot JVM. Using the YCSB benchmarking tool to benchmark HBase, a well-known and widely used Big Data application, we show negligible overhead in frequent operations such as the allocation of new objects, and significant improvements when accessing data, supported by higher hits in system-level memory structures.

Keywords: Cloud Infrastructure, Java Virtual Machine, Garbage Collection, Locality-Aware, Big data

1 Introduction

Managed languages (such as Java) are gaining space as the choice to implement Big Data processing and storage frameworks [19, 14, 10, 15], as they facilitate application development, This is mostly due to its automated memory management capabilities, flexible object-oriented design and quick development cycle. These languages, and Java in particular, run on top of a runtime system (the Java Virtual Machine, JVM, is one such case) that manages code execution and memory

management. Memory management is governed by the Garbage Collector (GC), a component that controls how objects are allocated and collected. Despite the considerable development benefits of automatic memory management, the GC can lead to serious performance problems in Big Data applications.

In particular, some of these performance problems are caused by the fact that the GC does not respect application's working set locality. In fact, while the application is running, the GC will move application objects throughout memory, possibly separating objects that belong to the same dataset and that, therefore, should be close to each other. This is a consequence of throughput oriented management mechanisms implemented by the GC that, however, hinder co-locality of related objects and the way objects are represented and placed in memory [6, 5, 9].

Space locality is known to have a relevant impact in performance [16, 29, 10]. For example, Wilson *et. al.* [29] exploited the hierarchical decomposition of data structure trees to reorganize the tracing algorithm, instead of strict depth-first or breadth-first tracing. Dynamic profiling was also studied by Chen [6], so that information on frequency of access is gathered and used in the placement of those objects. Huang [11], on the other hand, showed different strategies for online object reordering during GC, in order to improve program locality. Also, Ilham's work [12] shows increased locality in system-level memory structures, such as the L1D[4] cache and the dTLB[5], when ordering schemes for children object placement are accounted for, i. e., Depth-First (DF), Breath-First (BF) and Hot Depth-First (HDF). However, these works either apply a similar approach to all objects, which makes it difficult to tailor for storage-specify data-structures, or are hard to scale to very large heaps given the impact of per-object profiling in execution time. Furthermore they were not evaluated with modern parallel GC algorithms.

To reduce the impact of GC in the context of Big Data applications, others have made extensive modifications to the way certain objects are created and managed in special propose memory spaces, either requiring compiler and GC modifications, or application-specific data structures [5, 17].

In this work, we are focused on large-scale key-value databases such as HBase [1], Cassandra [14], and Oracle KVS [3]. These databases tend to hold massive amounts of objects (key-value pairs) in memory, which end up being scattered in memory due to poor GC techniques, that completely disregard object locality. As the application graph grows, the number of misses and faults in memory increase with clear negative impact on applications performance.

We propose a novel approach by enhancing GC with locality awareness. In other words, we propose *LAG1*, a modified GC which goal is to keep highly related groups of objects (i.e., objects that have many references between each other, for example, a data structure) close to each other in memory, leading to improved locality. Thus, *LAG1* takes takes into account object references when moving objects in memory. *LAG1* is implemented by modifying a state-of-art GC algorithm, the Garbage First (G1), the new by-default GC for the OpenJDK

[4] The first level of the CPU data cache
[5] The data Translation-Lookaside-Buffer

JVM. *LAG1* does not require the use of new data structures, or even changes to existing code, thus making the solution easier to adopt in current and new systems.

In sum, the main contributions of this paper are:

i. A tracing algorithm, designed for automatically managed heaps, to identify highly related groups of objects.
ii. A garbage collector extension that copies highly related groups of objects to specific memory segments.

The rest of the paper is organized as follows. Section 2 presents the main building blocks of modern garbage collectors, and the factors that hinder locality in NoSQL Big Data storages. Section 3 presents the architecture of *LAG1*, a novel extension, for an existing GC, to co-locate object graphs in memory. Section 4 shows the modifications made to G1, a modern parallel GC, on top of which we built *LAG1*, and heap organisation to avoid the previous problems without modifications to the application. Section 5 presents the evaluation of *LAG1* showing its benefits, the small overheads of this solution and the benefits at application level. Section 7 draws final conclusions.

2 Background

Many of today's most used NoSQL databases are written in high-level languages [19], such as Java and C#. By doing so, developers rely on services supported by these managed runtimes, in particular, automatic memory management, GC. However, GC introduces several performance issues. As already mentioned, the lack of object locality compromises application performance; in this paper, this is exposed in the context of NoSQL databases, and a solution (*LAG1*) is proposed.

This section is used to further motivate for the problem and to provide sufficient background for the next sections, which describe the proposed solution.

2.1 NoSQL Databases and Object Locality

Currently, NoSQL databases are a popular tool to store massive amounts of data. Examples of these storage systems include HBase [1], Cassandra [14] and Oracle KVS [3]. These are distributed, column-oriented NoSQL databases, whose 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.

NoSQL databases use large caches to hold hot accessed data. However, it is a challenge for the GC to efficiently manage such large in-memory data structures. In fact, when running the YCSB benchmark framework [7] with a dataset of 12 GB, we noticed an excessive use of *page swapping* resulting from poor GC decisions that cause long application pauses (the result of this is shown in Section 5). Previous works, such as Bu *et al.* with their *bloat-aware design* [5] have also alerted to this problem; we believe that going a step further to co-locate

related objects achieves even better locality on system-level memory structures to benefit the overall execution time.

As time goes by and operations are performed on top of a NoSQL database, the GC needs to reclaim unreachable objects in order to free space for new application objects. By doing so, the GC copies objects in memory in order to free segments of memory. Since the GC does not account for object locality, it copies groups of highly connected objects (i.e., objects with many references between each other, for example, a data structure) into distant memory locations. This degrades application performance as cache locality does not hold in these scenarios. As datasets become larger, the amount of memory consumed by a NoSQL database grows, leading to an increased distance between highly connected objects.

2.2 Garbage Collection Algorithms and Heap Layouts

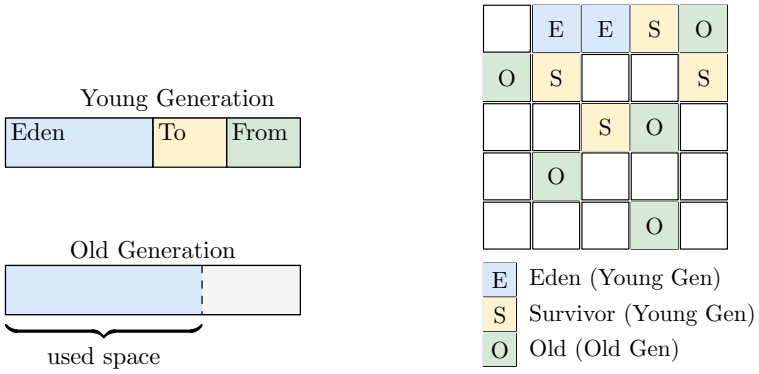
Garbage Collection (GC) is a well-known and widely used technique to automatically manage memory, i.e., programmers do not need to free objects after using them [13]. The GC operates over a large memory space called heap. All application objects reside in the heap, and is the job of the GC to provide memory for new application objects and to collect memory used by unreachable application objects.

Modern garbage collectors are generational, meaning that they follow the assumption that *most objects die young* [26]. Thus, most popular GC implementations divide the heap into two generations, one that holds newly allocated objects (the *young* generation) and one to hold objects that survived for at least a number of GC cycles (the *old* generation).

The young generation is further divided into three spaces: *eden*, *to* and *from*. The *eden* is used to fulfill object allocation requests while the *to* and *from* are used to hold objects that survived at least one GC cycle and that will be eventually copied to the *old* generation.

G1 [8], the baseline of our work, is one such generational GC with a region-based heap. A heap of this kind is split in small fixed-sized regions, instead of strictly dividing the heap as in Figure 1 (a). This is illustrated in Figure 1 (b), where several regions can be seen, each with its purpose. Thus, regions with an O are old regions (abstractly they belong to the *old generation*); regions with an S are survivor regions; and regions with a Y are young regions, both abstractly belonging to the *young generation*. G1 also keeps a per-region remembered-set. For each region, this set describes inter-region references between objects. Its use is important during garbage collection to know if there are objects from survivor or old regions that reference objects in young regions (thus allowing to collect only young regions).

Recent GC implementations, including G1, provide two main types of GC cycles: minor and full. A minor collection is designed to collect the *young* generation and to copy/promote survivor objects into the *old* generation. Since objects that survive only one minor collection might not be automatically promoted into the *old* generation (this is a GC configurable option), survivor spaces are used



(a) Parallel Scanvenge Heap Layout

(b) G1 GC Heap Layout

Fig. 1: GC Heap Layouts

to hold these objects until they are old enough to be promoted. A full collection, as the name suggests, collects the whole heap (both *young* and *old* generations).

3 Gang Promotion in Heap Management

This section presents *LAG1*, a GC extension to co-locate dependent objects in order to improve the locality in system-level memory structures. With *LAG1* we introduce the concept of gang promotion to achieve this result. Also, we introduce a new memory region definition, which manages an unique segment of memory, called *container-region*.

The concept of gang promotion consists in copying live objects (in contrast with dead objects, i.e., unused objects in the heap) in a manner where the sub-graph of a family of objects does not intersect another sub-graph of a family of objects. We define a family as a group of objects belonging to the same sub-graph (for example, a data structure such as a linked list). The result is a single segment of memory for each family of objects. These segments of memory are *container-regions*.

For gang promotion to work, several challenges need to be addressed. These challenges are the following: i) how to identify sub-graphs of highly-related objects, and ii) how to efficiently promote sub-graphs of highly related objects, without causing overhead on the existing promotion mechanism, to a specific memory region. The following sections explain how we tackled these challenges in the two participating sub-systems for object management: the runtime (Section 3.1), the system that executes the application code and allocates new objects; and the garbage collector (Section 3.2), the system that collects objects no longer in use by the application.

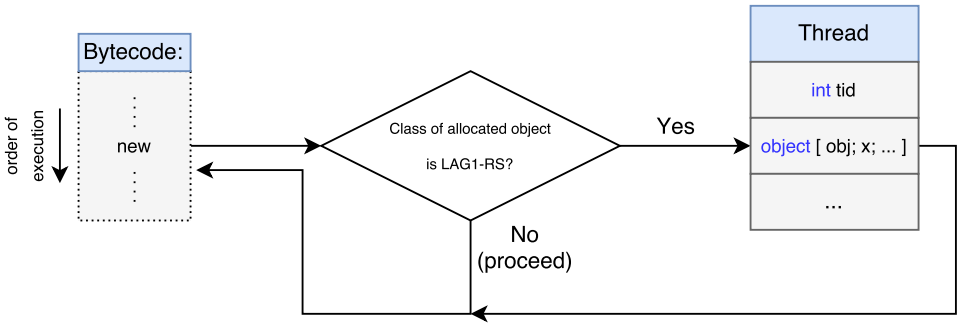


Fig. 2: Bookkeeping head of an object sub-graph at instrumented allocation site

3.1 Identifying Relevant Object-Graphs

One of the challenges in gang promotion is to identify sub-graphs of highly-related objects to co-locate, non-intrusively to the runtime sub-system. Naively, bookkeeping every allocation call and, at a *safepoint*[6], filter the sub-graph of allocated objects could suffice. But for a very large number of object sub-graphs, such as in a Big Data environment, that would largely consume system resources.

We have taken the approach of instrumenting only relevant allocation sites. Relevant allocation sites are those that allocate objects of a type deemed to be the head of a highly-related object sub-graph. This goes along with the fact that the head of an object sub-graph is usually the first object to be allocated in the system (another terminology is to refer the head of the sub-graph as the *root* of a structured tree). Therefore, identifying the root is enough for *LAG1* to move the whole tree to a special memory region in a later stage. In order for *LAG1* to identify the root, the user of the program must specify what is the type of the object that is the root. The set of types for the root objects is called *LAG1-RS*. The heuristics behind the identification of the root's type, and thus what can be included in the *LAG1-RS*, is left for the user to decide.

To avoid conflicts among mutator threads, *LAG1* saves identified roots on a thread-local array shown in Figure 2. This figure shows what the instrumented allocation site does; it queries if the class of the object being instantiated is a *LAG1-RS* class and, if so, it inserts the object in the thread-local array. The term *LAG1-RS* (abbreviated form of *LAG1* Root Set) is the set of sub-graph roots across all mutator threads. This step is important for the GC stage, i.e., when the live object-graph is moved to a survivor space.

3.2 Gang GC on a Large Heap

To correctly promote object sub-graphs identified through the mechanism presented in Section 3.1, two more challenges need to be addressed: i) avoid unreach-

[6] The mechanism used in HotSpot to create Stop-the-World pauses. Garbage collection cycles run inside a safepoint, during which all application threads are stopped.

able sub-graphs referenced by *LAG1-RS*; and ii) integrate sub-graph promotion in *LAG1* with the existing promotion mechanism in G1. We address these two challenges in the paragraphs below.

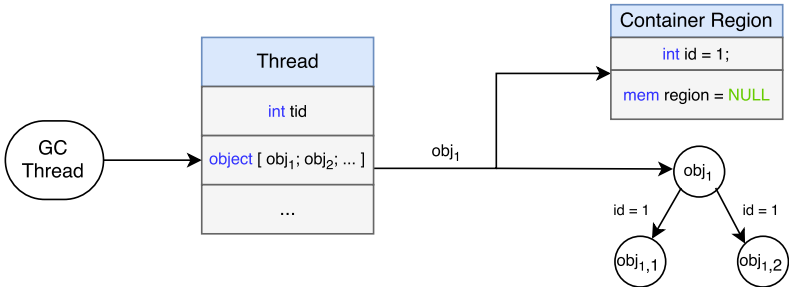
Avoid Unreachable Sub-Graphs Saving allocated heads of sub-graphs in thread-local arrays can have adverse effects, if these references are used as roots for tracing the remaining sub-graph at GC time. This is so because the references may belong to objects used as temporary storage (use cases for this are VM warm-ups, cloning data, etc.) and can become unreachable by the tracing algorithm. *LAG1* uses the *LAG1-RS* to create *container-regions*, instead of using it for tracing live objects. There is no memory region associated with newly created *container-regions*, therefore obliging that the sub-graph must be reachable by the root-set (threads' stacks, globals and statics) in order to have heap space effectively assigned.

Another important aspect of *LAG1* is associating the identifier of the created *container-region* with each object in the *LAG1-RS*. With this technique, *LAG1* does not lose the associative relationship between the head of a sub-graph and the *container-region* that will hold the objects. It also provides a way for *LAG1* to propagate this identifier to this root's followers or children (i.e., the objects belonging to the root's sub-graph).

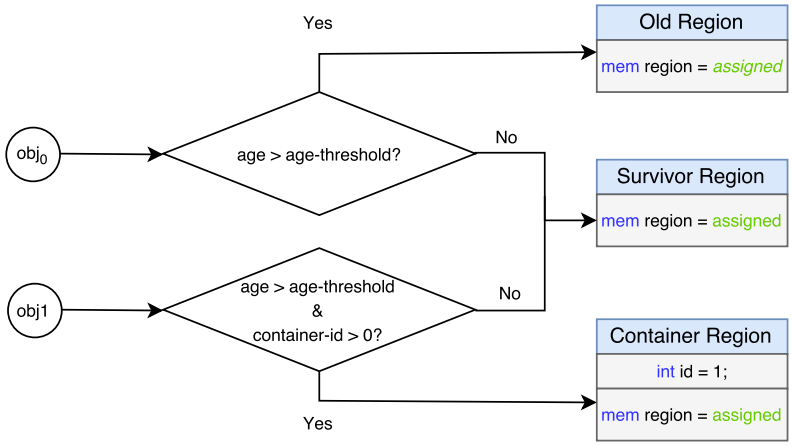
Promotion of *LAG1* sub-graph The first phase of a GC cycle is to trace from the root-set, i.e., the threads' stack frames, the global variables and the static variables in the system. The order of the operations is non-important since garbage collectors are designed to be throughput-oriented during collection phases. This policy does not conform with *LAG1* policy that every object should be located near its siblings. To avoid this, *LAG1* inserts a checkpoint phase before letting the rest of the tracing algorithm do its work. The checkpoint phase consists on propagating the container region identifier that the *LAG1-RS* created to its followers. This will associate any follower of a certain object in the *LAG1-RS* with the identifier for the same *container-region* of the parent, recursively. In *LAG1*, this phase is called *pre-marking*. *LAG1* also provides a work-stealing model for parallel garbage collectors during the *pre-marking* phase to speed-up computation. Although *pre-marking* may add additional overhead to the GC, experiments show that heads of object sub-graphs are allocated rarely during the application. Thus, the overhead is negligible for a long running application.

After the *pre-marking* phase, the second phase of the GC, which consists on the tracing and promotion of root-set followers, can proceed normally with no constraints. *LAG1* no longer intervenes on the GC phases until it sees a live object associated with a *container-region* and targeted for promotion to a *tenured* space. For these objects, it targets its destination space to the *container-region* instead of the default old region. Since *container-regions* are on the same space as the old regions, *LAG1* complies with the object age while still providing a target space next to its siblings of "tenured" age in a transparent manner.

Figure 3 shows a rundown of the operations executed as part of gang promotion. In Figure 3 (a), a GC Thread is shown accessing the thread-local array of a mutator thread, which previously saved obj_1 and obj_2. The thread then fetches the obj_1 reference, creates a new *container-region* with an unique identifier and propagates this identifier to the siblings of obj_1. Figure 3 (b), on the other hand, shows the promotion step. It illustrates a regular object obj_0 being promoted, with no need to check if it has any *container-id* association (in the *pre-marking* phase, *LAG1-RS* objects and its siblings were associated with their *container-region*). But, since obj_1 has an association with a *container-region* it goes through a different condition. Therefore, obj_1 is copied to the memory region of its *container-region* if it is old enough.



(a) *pre-marking* phase



(b) Promotion of pre-marked *LAG1* object

Fig. 3: A rundown of the operations executed for gang promotion

4 Deployment of gang promotion on Hotspot JVM

LAG1 is implemented on OpenJDK 8 HotSpot JVM. The OpenJDK HotSpot JVM [2] is the state-of-the-art Java virtual machine used in most Java deployments. It is a highly portable and highly optimized virtual execution environment for Java-bytecode based languages (Java, Scala, Clojure, etc.). The new by-default garbage collector is the Garbage First (G1) GC [8], a low-pause collector, with a soft real-time pause guarantee, while still achieving high throughput. G1 is the baseline garbage collector of this work's prototype, thus we take advantage of some of its features such as: the generational heap space and its region-based division of the heap, meaning that the heap space is split into small fixed-sized regions.

In this section we describe our modifications to the Java runtime sub-system of HotSpot JVM (Section 4.1) and how we modified G1 to implement *LAG1* (Section 4.2).

4.1 Java Runtime Instrumentation

In our prototype, we tackled the problem of identifying relevant object sub-graphs (Section 3.1) in the Java runtime. The Java runtime sub-system of HotSpot is divided into three components: i) the assembly interpreter, ii) a lightly optimizing bytecode compiler (C1), iii) a highly optimizing bytecode compiler (C2). C1 is better suited for client-machine applications, thus we disregarded its application. We only considered the assembly interpreter and C2, the latter for methods with high invocation count.

Instrumentation for *LAG1* was tackled on the allocation site of the root of a relevant object sub-graph (e.g., a data structure). Relevant object types are left for the user to decide, using a command line option. For example, if the relevant object sub-graph is a `LinkedList` structure, the user should specify the full qualified class name (i.e., `java.util.LinkedList`). Since class loading is prior to object allocation for any given type, we first register the user-specified class to be *LAG1-RS* by placing a bit on the virtual machine class-representation. Thus, during allocation, all we do is a fast check for the bit on the class to be installed on the object. If it is present, then we add the object address to a thread-local indexed array. This requires only three operations at assembly-level, a compare (for the presence of the bit in the class), a load (to load the object address in the thread-local array) and an increment (to increment the thread-local array index).

4.2 LAG1 — Locality-aware extension of G1

To implement *container-regions*, *LAG1* takes advantage of the regionalized architecture of the heap that G1 already provides. Since old regions (G1 heap regions belonging to the old generation) are already present in G1, *container-regions* are specialized old regions. The reason for this decision is that *LAG1*

handles large object sub-graphs, preferably long-lived, thus it would be impracticable to use the young generation regions.

During minor GC (a GC that collects only the young regions), before the tracing of the reachable live object graph is initiated, *LAG1* checks if there are saved references in the *LAG1-RS*. If there are references, the *pre-marking* phase is initiated. The *pre-marking* phase comprises both creating *container-regions* for *LAG1-RS* objects and propagating the *container-region* identifier to its followers. This identifier is simply an integer to index the *container-region* array. Additionally, *pre-marking* also includes tagging relevant objects. The identifier integer and tag bit are installed on unused bits in the header, such as illustrated in Figure 4, where it shows the *pre-marking* of an object header. In Figure 4 some lower-level details are shown, regarding the header of a Java object, with the important bits in the *tag-bit* (an unused bit in the HotSpot JVM, which we use to mark a *LAG1-RS* object and its sub-graph) and in the *container-region* identifier.

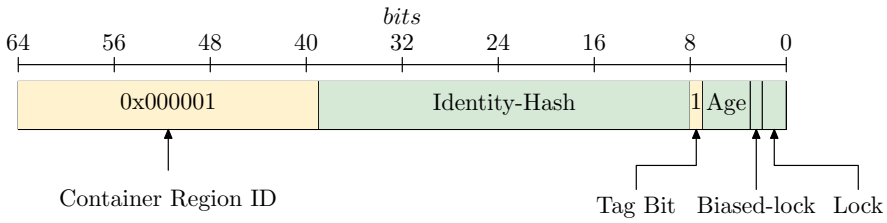


Fig. 4: The *pre-marking* phase tagging an object with a container region identifier

While implementing *LAG1*, we took into consideration that may exist *LAG1-RS* objects already promoted in a previous GC. This means that may exist new objects (allocated since the last GC), children of a *LAG1-RS* object already promoted in an earlier stage. To mark these newly allocated objects with the *container-region* identifier, we intercepted the remembered-set operations of G1 (also called *old-to-young* on other garbage collectors [9]) and added one more instruction to install the identifier of the referent (the parent in an older generation) on the follower in the young generation. Therefore, there is no possibility of losing the follower to another space by not being *pre-marked* in time, because the remembered-set operations are always executed before the promotion of the followers. Another favorable aspect of this approach is that it no longer requires a checkpoint barrier before regular tracing, such as explained in Section 3.2.

The last stage for *LAG1* is to promote (i.e., copy) objects according to the *container-region* identifier. Since G1 already checks the object's age to decide if it should promote to a survivor region or to an old region, *LAG1* only adds an additional check. The check consists on looking at the object header and see if it has a container region identifier installed. It is a fast bit mask operation, so no overhead is inflicted.

5 Evaluation

To evaluate *LAG1*, we considered the fact that hot object sizes in Java are as big as L1[7] and L2[8] cache line sizes, and thus very few of them fit in those caches. Therefore, our experiments consisted in observing the virtual memory performance, more specifically the dTLB (CPU-level) and the *page-table* (Kernel-level) system structures. Also, we evaluated our modifications to the OpenJDK 8 HotSpot JVM, in the form of the application throughput. The next sections present the setup we used (Section 5.1), the program locality achieved with our solution (Section 5.2), and the high-level behaviour of the application (Section 5.3).

5.1 Evaluation Setup

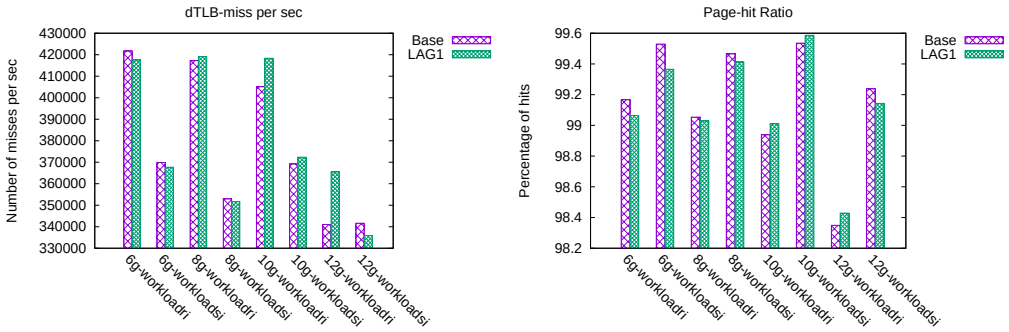
Experimental runs were executed on a 4-core machine with 8 logical cores, 3 levels of cache with a 8MB L3 and 16GB of memory, running a 64-bit Linux 4.4.0 kernel. To test the locality effects in system-level memory structures, such as the dTLB and page-table, we resorted to performance monitoring counters in the Linux tools package[9]. The target of our experiments was HBase [1], a widely used large-scale data store for Big Data processing, using YCSB [7] as a client application. YCSB is a highly configurable cloud benchmarking tool, widely used to benchmark large-scale data stores. The following paragraphs present the configurations we used on YCSB to benchmark HBase running our modified JVM.

YCSB can be configured with a large number of parameters, including: number of operations, number of records to load on the data store, the ratio of operations for each action (insert, update, read, scan) and the size of each record. The size of each record was fixed to 1KB for all experiments. Also, the number of operations to perform on the data was also fixed to $1 * 10^5$. On the other hand, the number of records to load and the ratio of operations was varied. Since the JVM was configured to have a maximum size of 12GB for the Java heap, the number of records (*load*) used was: i) 6GB, ii) 8GB, iii) 10GB and iv) 12GB. For this evaluation, the configuration for YCSB consisted of two workloads with memory loading characteristics: i) a read-intensive (RI) workload and ii) a scan-intensive (SI) workload. The detailed characteristics of each workload is described below.

Workload RI 70% of reads, 25% of scans and 5% of updates

Workload SI 25% of reads, 70% of scans and 5% of updates

[7] L1 is the 1st level of CPU cache: 32KB in size and 64B per line in modern models
[8] L2 is the 2nd level of CPU cache: 256KB in size and 64B per line in modern models
[9] http://linux.die.net/man/1/perf



(a) dTLB-misses per second (lower is better) (b) Page-hit ratio (higher is better)

Fig. 5: Locality on system-level memory structures

5.2 Program Locality in System

In this section we show the improvements that our solution has in system-level memory structures. The focus is given to the dTLB and the page-table, because Big Data Java applications (which handle large sub-graphs of objects) will not see a big improvement in CPU caches given their size.

Key-value stores, such as HBase [1], usually use multi-level map where, given a *table* name, a *row* name and a *field* name, a value can be inserted, read or updated [19]:

```
map <table-name, map<row-key, sortedmap<field-key, data>>>    (1)
```

With *LAG1* we expect that the *field-keys* and the actual data, represented in Equation (1), be closer in memory. Figure 5 shows the results obtained, for each pair `<size of data-set>-<workload type>`, where the bars for *Base* refer to the baseline JVM and *LAG1* our modified JVM. We begin our analysis with the observation that, the first step in virtual-memory address translation will start with: a dTLB load, then a page-table query (if the dTLB load misses) and then, if the requested address is not in the page-table, a page-fault is issued. Figure 5 (a) shows that the dTLB misses per second is stable for workloads that do not cause pressure in the heap, i.e., 6GB and 8GB of dataset size. Therefore, variations in the page-table, shown in Figure 5 (b), are mostly related to external factors (e.g., OS virtual-memory policies, GC, etc.).

However, as the size of the dataset — workloads of 10GB and 12GB of dataset size — gets closer to the Java heap size (12GB), we begin to see the dTLB is stressing. That means the dTLB cache no longer has the capability of saving that many translated virtual-memory addresses to physical addresses, thus this mechanism no longer becomes important. The responsibility is passed

to the page-table, where the OS will do a *page-walk*[10]. At this point, we see that with *LAG1* the page-table hit-ratio, shown in Figure 5 (b), is increased in comparison with the baseline JVM. This is more evident in read-intensive (RI) than scan-intensive (SI) workloads, because scan-intensive workloads read multiple values sequentially. And, as referred previously, Java objects may be large in size, when compared with system-level memory structures, thus spanning multiple page entries (and consequently, multiple dTLB entries). The test with 12GB of dataset size and a scan-intensive workload (*12g-workloadsi*) is the only that does not follow the pattern, but that is because it already has low dTLB-misses as shown in Figure 5 (a).

5.3 Application Behaviour

In this section, we present the results for the application throughput when running HBase with *LAG1*. The results are from the timeseries output of YCSB, which ran 100 000 (100k) operations on an HBase instance with a load of 6GB, 8GB, 10GB and 12GB records. We first ran a warm-up phase over the entries, therefore all results are the best obtained across 3 tests, in the percentile shown. The workloads were the same as in Section 5.1.

Figures 6 and 7 show the comparison of throughput between *LAG1* and the baseline JVM. It can be observed that, although *LAG1* added complexity to the baseline JVM, for all tests it did not influence throughput significantly (and in some cases, nothing at all). We believe that this is a positive result, because improvements in program locality outweigh the added complexity, and that future research could benefit from focusing on program locality aspects.

6 Related work

Research in automatic memory management has proven that there is no unique solution that fits all classes of applications. The best choice of GC is, in many cases, application and input-dependent [25, 24]. This has spanned a vast collection of algorithms, in many cases combinations of older ones, which can be stacked with application-specific profiles [23].

Parallel, stop-the-world algorithms have been making a successful entry in the field of big-data applications, since they can efficiently collect a whole heap within shorter pauses and do not require constant synchronization with the mutator, as it is the case with concurrent collection [9]. However, Java-supported Big Data applications in general, and storage in particular, stress the GC with lack of locality in large heaps and bloat of objects. This is mainly tackled using three kinds of approaches [17, 5, 15]: i) avoiding per-object headers and imposing new memory organizations at the framework-level, ii) speeding-up garbage collection by identifying objects that are created and destroyed together and,

[10] A page-walk consists on querying page-table entries, to see if the address the CPU is trying to load is present in physical memory

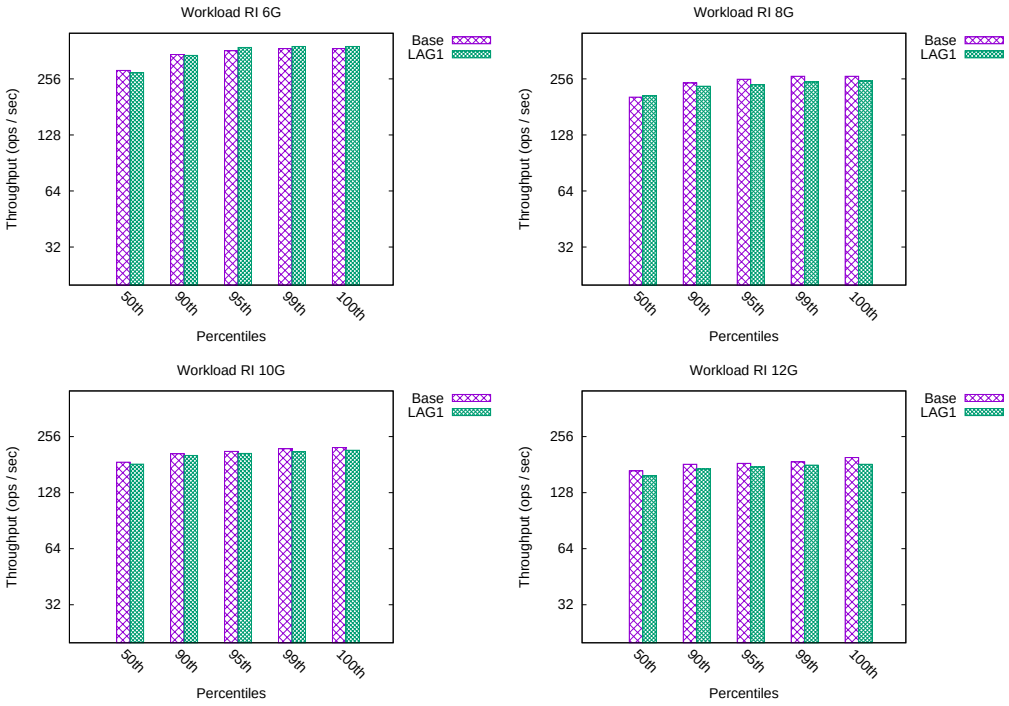


Fig. 6: Throughput on HBase with Workload RI for a variety of datasets

iii) coordinating the stop-the-world moment in inter-dependent JVM instances. Because most works focus on reducing overheads by dramatically changing the layout of objects and out-of-heap specially crafted structures, these solutions need changes both to the compiler and the GC system or rely on complex static analysis which is hard to prove correct and complete.

Facade [17] is a compiler and augmented runtime that reduces the number of objects in the heap by separating data (fields) from control (methods) and putting data in an off-heap structure without the need to maintain the bloat-causing header. Hyracks [5] is a graph processing framework that also uses a scheme where small objects are collapsed into special-purpose data structures. Because this is done at the framework-level, and not at the JVM-level, it is difficult to reuse the approach. Overhead can also be caused by GC operations running uncoordinated inter-dependent JVM instances [15]. When each of these instances needs to collect unreachable objects, if it does so regardless of each other; this can lead to significant pause times.

On the other hand, previous work about object ordering schemes [16, 6, 11] have shown that taking advantage of placement strategies, can increase locality in system-level memory structures and achieve better performance, especially when using guided techniques for optimal object placement. However, current

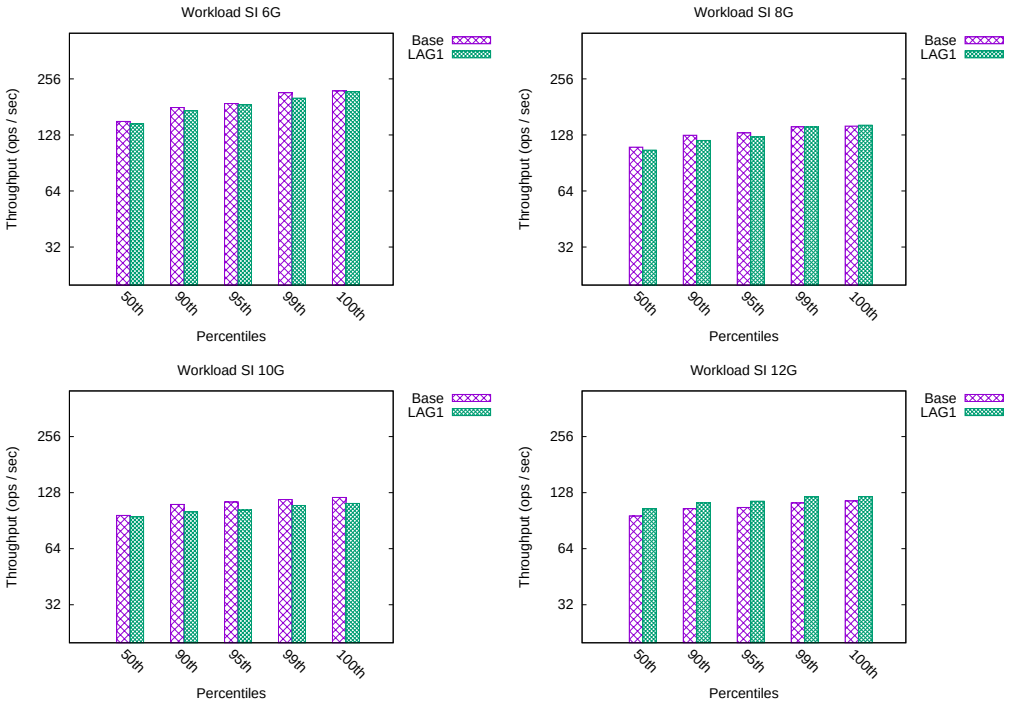


Fig. 7: Throughput on HBase with Workload SI for a variety of datasets

approaches rely either on static analysis of fine-tuned dynamic profiling to avoid an excessive overhead. Instead, *LAG1* only relies on the user to specify the class of objects that hold the data, since it is already a low overhead solution.

NG2C [4] is a new GC algorithm that combines pretenuring with user-defined dynamic generations. It allocates objects with similar lifetime profiles in the same generation; by allocating such objects close to each other, *i.e.* in the same generation, it avoids object promotion (copying between generations) and heap fragmentation (which leads to heap compactions) both responsible for most of the duration of HotSpot GC pause times. Compared to *LAG1*, NG2C takes another approach to the issue of object locality, which may result in objects that point to each other being dispersed in memory. In the long run, contrary to *LAG1*, this may lead to extra page faults and degradation of locality on system-level memory caches.

7 Conclusion

Several Big Data frameworks and storages are executed on a managed runtimes, taking advantage of parallel garbage collection and Just-In-Time (JIT) compilation. However, modern parallel memory management and throughput-oriented

techniques can hinder locality. Our approach was to promote objects' co-locality which minimizes the number of memory pages used, taking more advantage of system-level data and translation caches. This was done with an extension to the Garbage First (G1) GC promotion mechanism and algorithmic modifications to the runtime system, which we named *LAG1*.

The results provide positive conclusions on the use of *LAG1* on state-of-the-art JVM, the OpenJDK 8 HotSpot. First, we showed that the promotion efforts to co-locate highly-related object sub-graphs favourably increase page-table hits with real world executions. This is evident in large datasets with demanding workloads for the available memory, a common practice today. Second, we demonstrated that program locality outweighs added complexity on the runtime system with locality-aware policies. This was demonstrated with stable throughput across a variety of workloads and dataset sizes.

In the future, we would like to assess how the improvements provided by *LAG1* can also enhance performance transversally to other work on Java VM-based mechanisms and middleware, whose operation is also heavily dependent on object graph locality and on performing graph transversals, e.g., object replication [27, 28], checkpoint and replay, [21, 20], and dynamic software update [18].

Finally, although RAM memory is cheaper nowadays, the dataset sizes are growing faster than the available memory in cloud systems. Vendors cannot always comply with the agreed SLAs, because of the chaotic layout of objects in memory, when the latter is under pressure. It is our belief that given our results, future research could be more focused on program locality aspects. On the other hand, we are also focused on future work, including the evaluation with more specialized hardware, such as NUMA architectures, on larger datasets.

Acknowledgements: This work was supported by national funds through Fundação para a Ciência e a Tecnologia with reference PTDC/EEI-SCR/6945/2014, and by the ERDF through COMPETE 2020 Programme, within project POCI-01-0145-FEDER-016883. This work was partially supported by Instituto Superior de Engenharia de Lisboa and Instituto Politécnico de Lisboa. This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013.

References

1. http://hbase.apache.org/, visited feb 16, 2017
2. http://openjdk.java.net/, visited feb 16, 2017
3. http://www.oracle.com/technetwork/database/database-technologies/nosqldb/overview/index.html, visited feb 16, 2017
4. Bruno, R., Oliveira, L.P., Ferreira, P.: Ng2c: Pretenuring garbage collection with dynamic generations for hotspot big data applications. In: Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management. pp. 2–13. ISMM 2017, ACM, New York, NY, USA (2017), http://doi.acm.org/10.1145/3092255.3092272
5. Bu, Y., Borkar, V., Xu, G., Carey, M.J.: A bloat-aware design for big data applications. In: Proceedings of the 2013 International Symposium on Memory Management. pp. 119–130. ISMM '13, ACM (2013)

6. Chen, W.k., Bhansali, S., Chilimbi, T., Gao, X., Chuang, W.: Profile-guided proactive garbage collection for locality optimization. In: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 332–340. ACM (2006)
7. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM symposium on Cloud computing. pp. 143–154. ACM (2010)
8. Detlefs, D., Flood, C., Heller, S., Printezis, T.: Garbage-first garbage collection. In: Proceedings of the 4th International Symposium on Memory Management. pp. 37–48. ISMM '04, ACM, New York, NY, USA (2004), http://doi.acm.org/10.1145/1029873.1029879
9. Gidra, L., Thomas, G., Sopena, J., Shapiro, M.: A study of the scalability of stop-the-world garbage collectors on multicores. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 229–240. ASPLOS '13, ACM (2013)
10. Gidra, L., Thomas, G., Sopena, J., Shapiro, M., Nguyen, N.: Numagic: a garbage collector for big data on big NUMA machines. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 661–673. ACM (2015)
11. Huang, X., Blackburn, S.M., McKinley, K.S., Moss, J.E.B., Wang, Z., Cheng, P.: The garbage collection advantage. In: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications - OOPSLA '04. p. 69. ACM Press, New York, New York, USA (2004)
12. Ilham, A.A., Murakami, K.: Evaluation and optimization of java object ordering schemes. In: Electrical Engineering and Informatics (ICEEI), 2011 International Conference on. pp. 1–6. IEEE (2011)
13. Jones, R., Hosking, A., Moss, J.E.B.: The Garbage Collection Handbook: The Art of Automatic Memory Management. Chapman & Hall/CRC, 1st edn. (2011)
14. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review 44(2), 35–40 (2010)
15. Maas, M., Asanović, K., Harris, T., Kubiatowicz, J.: Taurus: A holistic language runtime system for coordinating distributed managed-language applications. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 457–471. ASPLOS '16, ACM, New York, NY, USA (2016)
16. Moon, D.A.: Garbage collection in a large lisp system. In: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming. pp. 235–246. ACM, New York, NY, USA (1984)
17. Nguyen, K., Wang, K., Bu, Y., Fang, L., Hu, J., Xu, G.H.: FACADE: A compiler and runtime for (almost) object-bounded big data applications. In: ASPLOS. pp. 675–690. ACM (2015)
18. Pina, L., Veiga, L., Hicks, M.W.: Rubah: DSU for java on a stock JVM. In: Black, A.P., Millstein, T.D. (eds.) Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014. pp. 103–119. ACM (2014), http://doi.acm.org/10.1145/2660193.2660220
19. Redmond, E., Wilson, J.R.: Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement. Pragmatic Bookshelf (2012)
20. Silva, J.M., Simão, J., Veiga, L.: Ditto - deterministic execution replayability-as-a-service for java vm on multiprocessors. In: Eyers, D.M., Schwan, K. (eds.)

Middleware. Lecture Notes in Computer Science, vol. 8275, pp. 405–424. Springer (2013)
21. Simão, J., Garrochinho, T., Veiga, L.: A checkpointing-enabled and resource-aware java virtual machine for efficient and robust e-science applications in grid environments. Concurrency and Computation: Practice and Experience 24(13), 1421–1442 (2012), https://doi.org/10.1002/cpe.1879
22. Simão, J., Veiga, L.: Adaptability driven by quality of execution in high level virtual machines for shared cloud environments. Comput. Syst. Sci. Eng. 28(6) (2013)
23. Singer, J., Brown, G., Watson, I., Cavazos, J.: Intelligent selection of application-specific garbage collectors. In: Proceedings of the 6th international symposium on Memory management. pp. 91–102. ACM (2007)
24. Soman, S., Krintz, C.: Application-specific garbage collection. J. Syst. Softw. 80, 1037–1056 (July 2007), http://dx.doi.org/10.1016/j.jss.2006.12.566
25. Tay, Y.C., Zong, X., He, X.: An equation-based heap sizing rule. Performance Evaluation 70(11), 948–964 (Nov 2013)
26. Ungar, D.: Generation scavenging: A non-disruptive high performance storage reclamation algorithm. ACM Sigplan Notices 19(5), 157–167 (1984)
27. Veiga, L., Ferreira, P.: Incremental replication for mobility support in OBIWAN. In: ICDCS. pp. 249–256 (2002), https://doi.org/10.1109/ICDCS.2002.1022262
28. Veiga, L., Ferreira, P.: Poliper: policies for mobile and pervasive environments. In: Kon, F., Costa, F.M., Wang, N., Cerqueira, R. (eds.) Proceedings of the 3rd Workshop on Adaptive and Reflective Middleware, ARM 2003, Toronto, Ontario, Canada, October 19, 2004. pp. 238–243. ACM (2004), http://doi.acm.org/10.1145/1028613.1028623
29. Wilson, P.R., Lam, M.S., Moher, T.G.: Effective static-graph reorganization to improve locality in garbage-collected systems. SIGPLAN Not. 26(6), 177–191 (May 1991)