Performance Analysis and Optimization of Full Garbage Collection in Memory-hungry Environments *

Yang Yu‡¶, Tianyang Lei§, Weihua Zhang‡¶†, Haibo Chen§, Binyu Zang§

‡ School of Computer Science, Fudan University
¶ Shanghai Key Laboratory of Data Science, Fudan University
§ Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
† Parallel Processing Institute, Software School, Fudan University
{yu_y13, zhangweihua}@fudan.edu.cn, {sky1young, haibochen, byzang}@sjtu.edu.cn

Abstract

Garbage collection (GC), especially full GC, would non-trivially impact overall application performance, especially for those memory-hungry ones handling large data sets. This paper presents an in-depth performance analysis on the full GC performance of Parallel Scavenge (PS), a state-of-the-art and the default garbage collector in the HotSpot JVM, using traditional and big-data applications running atop JVM on CPU (e.g., Intel Xeon) and many-integrated cores (e.g., Intel Xeon Phi). The analysis uncovers that unnecessary memory accesses and calculations during reference updating in the compaction phase are the main causes of lengthy full GC. To this end, this paper describes an incremental query model for reference calculation, which is further embodied with three schemes (namely optimistic, sort-based and region-based) for different query patterns. Performance evaluation shows that the incremental query model leads to averagely 1.9X (up to 2.9X) in full GC and 19.3% (up to 57.2%) improvement in application throughput, as well as 31.2% reduction in pause time over the vanilla PS collector on CPU, and the numbers are 2.1X (up to 3.4X), 11.1% (up to 41.2%) and 34.9% for Xeon Phi accordingly.

1. Introduction

Managed programming languages like Java have been steadily adopted in parallel computing due to its ease of programing thanks to its inherit threading, portability and automatic memory management. Actually, many big-data frameworks like Hadoop and Spark [26, 30] use Java virtual machines (JVMs) to run their tasks. There is also a steady momentum to adopt Java-like programming language to high-performance computing (HPC) domains to increase program productivity [17, 27, 29].

While the volume of memory for a single machine has been steadily increasing, memory is still a scarce resource. First, many data-intensive Java applications with a large working set suffer from a general phenomenon called memory bloat [6] due to processing a large amount of data. This is especially true for a shared-cluster design inside many companies like Google [31] such that each application is only accompanied with a limited amount of memory. Second, the increasing number of cores per-machine usually results in limited per-core memory. This problem especially exists in the Many Integrated Core (MIC) architecture (e.g., Intel Xeon Phi), which has an excessive number of cores/hardware threads sharing only a small amount of memory (e.g., 60 cores/240 threads sharing only 8GB memory).

Efficiently running Java applications on such memory-hungry environments requires efficient garbage collectors. There have been a number of algorithmic designs that try to mitigate memory pressure and improve scalability for such environments [4, 23, 25]. Recent work also shows that GC would occupy non-trivial proportion of execution time [13] and the accumulated stragglers due to GC would lead to increased overall execution time [15] as well as amplified tail latency [9] for big-data applications.

In this paper, we present an in-depth study on the performance behavior of Parallel Scavenge, the default garbage collector for HotSpot JVM in OpenJDK. PS is a throughput-oriented, stop-the-world garbage collector that uses a variant of Mark-Compact algorithm [14]. While less frequently, full GC would cause longer pause time than minor GC and thus incur more performance impact to applications. With the assist of a detailed profiling, our analysis shows that full GC may still cause a non-trivial impact to application performance (e.g., more than 50% for *JOlden.TreeAdd*).

A detailed profiling reveals that the bottleneck lies within the reference updating period of the compacting phase. The

* This work is supported by National High Technology Research and Development Program of China (No. 2012AA010905), NSFC (No. 61572314 and 61370081), National Youth Top-notch Talent Support Program of China and Singapore NRF (CREATE E2S2). Related patches have been upstreamed to OpenJDK (JDK-8146987). Yang Yu was a visiting student at IPADS, SJTU when doing this work.

VEE '16, April 2–3, 2016, Atlanta, Georgia, USA.

ACM 978-1-4503-3947-6/16/04. . . $15.00.
http://dx.doi.org/10.1145/2892242.2892251

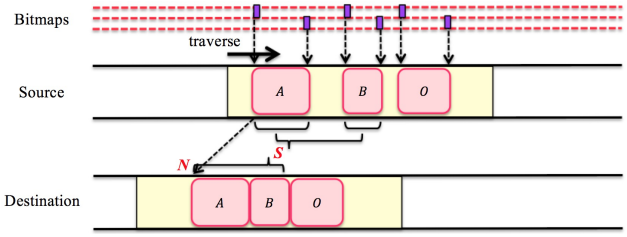


Figure 1: Update references using mark bitmaps

updates need to query two globally-preserved bitmaps mapping to the heap space that is segregated into multiple regions. The bitmap is searched from a specific region start as a reference is updated, which results in a lot of redundant calculations and memory accesses and contribute to more than 70% of full GC time. To this end, we propose a general *incremental query* (IQ) model to dynamically reuse the result of last query, which can significantly minimize the ranges for bitmap searching. According to different query patterns of runtime object references, we further design three IQ-based schemes (namely optimistic, sort-based and region-based) to maximize the reusability and query efficiency.

We have implemented the above designs in the HotSpot virtual machine of OpenJDK 7u and 8and perform a set of experiments using both standard benchmarks like *JOlden* [7], *Dacapo* [5], *SPECjvm2008* [1] as well as real-world big-data frameworks like *Spark* [26] and *Giraph* [3]. Our evaluation shows that our incremental query achieves up to 2.9x and 3.4x speedup in full GC throughput and 57.2% and 41.2% improvement in application throughput on a Xeon CPU server and a Xeon Phi coprocessor accordingly.

This paper makes the following contributions:

- A thorough profiling-based analysis of the full GC in Parallel Scavenge, which uncovers that reference updating in the compact phase is the most time-consuming bottleneck (§ 2).
- An incremental query model and three different schemes to accelerate the bitmap searching and improve the full GC throughput in Parallel Scavenge (§ 3).
- A detailed performance evaluation confirms the effectiveness of our design on both Intel Xeon and Xeon Phi MIC architectures (§ 4).

2. Performance Analysis of Parallel Scavenge

2.1 Parallel Scavenge

The HotSpot VM in OpenJDK leverages the well-known *generational collection* [14] to segregate the heap into multiple areas on the basis of different objects' ages, i.e., *young*, *old* and *permanent* generations. Parallel Scavenge is the default garbage collector with a *stop-the-world* fashion, i.e., the application will be suspended during GC. Here, we briefly introduce minor collection and review the design of full GC in greater details.

Young/Minor GC: The young collector in PS uses a copying algorithm by dividing heaps into several areas: an *eden* space and two *survivor* spaces. Most objects are initially allocated in the eden space (except for some large

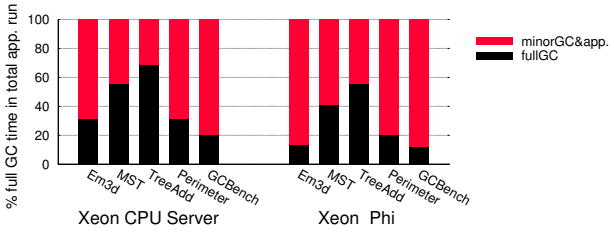


Figure 2: Proportion of full GC time in total application run

ones). After a minor collection, some objects are moved to one of the survivor spaces, with other aged live objects being promoted to the old generation.

Full GC: The full GC leverages a mark-compact algorithm to collect the whole heap. The compaction process slides all live objects towards the starting side, thus could effectively avoid fragmentation and allow the *bump-the-pointer* technique for efficient memory allocation.

The full collection consists of three phases: *marking*, *summary* and *compacting*. In the *marking phase*, the collector utilizes an important data structure called *mark bitmap* to map the whole heap space. PS maintains two bitmaps, one for the beginning address of an object and the other for the end address. The corresponding bits in the two bitmaps are set to identify if an object is alive. The heap space is partitioned into a lot of regions (each with a size of 4KB). A piece of metadata is maintained for each region. As an object within a region is marked alive, the corresponding metadata for the region will be updated.

The *summary phase* is responsible for calculating the new location for each compacting region and the live objects within it, e.g., the new location of the region that a live object will be copied to. This phase is done very quickly and contributes little to the full GC time.

The essential work of the *compacting phase* is to move live objects to their new locations and update all the references contained in the objects. Initially, a set of empty regions are maintained as the destination regions, whose corresponding source regions have already been determined during the summary phase. The collector sequentially moves the live objects from the source region to the destination region.

For each live object, all its references will be updated as long as it is moved to the destination region. As illustrated in Figure 1, for a referenced object, e.g., *O*, its new location could be calculated by using the mark bitmaps and the metadata of the region it resides in. Since the region metadata records the new location of the first live object in the region (denoted as *N*), any live object in this region could be located by adding the total live objects' size between the first object and itself, e.g, *S*, to *N*. *S* can be computed by searching the mapping range on the mark bitmaps of this region. After the compacting phase, all live objects are compacted to the beginning of the space.

2.2 Impact of GC on Big Data Applications

While GC usually should have only small impact on application performance, there are several cases where the GC time would constitute a non-trivial portion of application execution time. Actually, a recent study shows that TPC-H Q17 and "shopper" workflows in Naiad [19] spend 20∼40%

of their total runtime on GC regardless of the heap size for young generation [13]. Besides, the full GC is always claimed to be a key constraint to the throughput of stop-the-word collectors [16, 20, 28] like the Parallel Scavenge.

To confirm this, we measured a set of data-intensive Java applications to demonstrate the significant impact of full GC on both normal CPU and Xeon Phi (detailed evaluation setup in § 4). As shown in Figure 2, a considerable proportion of full GC time can be observed for all the applications with multiple GC threads on both CPU and Xeon Phi. The full GC time of *TreeAdd* could even exceed half of the whole execution time. This is because that an insufficient heap space caused by the memory-hungry environment may bring a high probability of full GC.

2.3 Detailed Performance Analysis

To further discover the most time-consuming part in the full GC of Parallel Scavenge, we designed and implemented a detailed profiling tool to attribute the full GC execution into individual operations. For simplicity, we utilize one GC thread to focus on the key operational logic.

Algorithm 1 Calc_new_pointer

Require: $addr \leftarrow$ old referenced object address
1: $region \leftarrow getRegion(addr)$
2: $dest \leftarrow region.destination()$
3: **if** $region.allAlive()$ **then** # dense path
4: **return** $dest + offset\_in\_region(addr)$
5: **else** # sparse path
6: **return** $dest + region.partial\_obj\_size() + live\_words\_in\_range(region.partial\_obj\_end, addr)$
7: **end if**

We first differentiate the full GC time in terms of the three different phases. Figure 3 shows that the compacting phase constitutes a majority of the total time for almost all applications, while the marking and summary phases only constitute a very small percentage. As mentioned in section 2.1, the compacting phase mainly does two thing: copy objects to their new locations and update all the references. From the graph we can see a much larger proportion for the reference updating (*sparse_update* and *dense_update*), which is reasonable because each time a reference is updated, the mark bitmaps need to be traversed from the region start for calculation of the new location, no matter how far the object lies from the start. Moreover, the amount of references is usually far greater than that of live objects in a Java application.

When profiling deeply into the reference updating procedure, we find that most time is spent on calculating the new location of the referenced objects. The new address is calculated by accumulating the live objects' sizes within the range, as illustrated in Algorithm 1, there are two paths: *dense path* and *sparse path*. When all data in the region is alive, it enters the dense path, where the new offset of the object is exactly the same with the offset in the current region. Otherwise, it enters the sparse path, in which the *live_words_in_range()* method will search the bitmaps from the end of the object that partially extends onto the region to the current object's location for total live sizes. Figure 3 reveals that for most benchmarks, the time spent on the sparse path significantly exceeds that on the dense path.

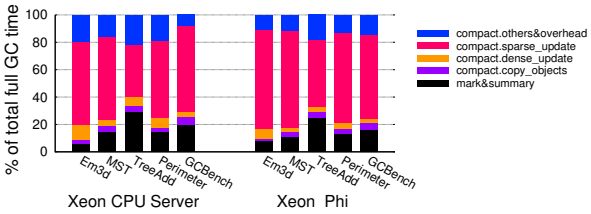


Figure 3: Decomposition of full GC time

Using our profiling tool, we can observe a substantial proportion of the "sparse path" of total full GC time. Hence, it is critically important to optimize the sparse path in the compacting phase for an overall reduction of full GC time.

3. Incremental Query

3.1 Basic Idea

According to our analysis, the key issue to optimize the full GC is to minimize the unnecessary memory accesses and calculations during reference updating period. However, the original *sparse path* in PS collector's compacting GC is very time-consuming and inefficient. For example, as shown in Figure 1, for two sequentially searched objects *A* and *B* in the same region, the range from the region start to *A* is repeatedly searched for both *A* and *B*.

To eliminate such redundant searches, we propose an incremental query scheme, which dynamically reuses the previous result for the next calculation. This idea is based on a key observation on Parallel Scavenge that each query is initialized from a fixed region start and confined within the current region.

Specifically, when the objects that two sequentially updated references point to lie in the same source region, the calculation for the latter object's new offset could be reduced by keeping track of the former's result. With the former object's address and new offset (i.e., the total size of live objects from region start), the latter object only needs to traverse the range from the former object, which could reduce a lot of redundant memory accesses and calculations.

Our basic idea is illustrated in Algorithm 2. The *beg_addr* and *end_addr* refer to the start and end of the searching range, respectively. In line 2 we first check if the *beg_addr* of current object matches that of last recorded object, if matches, they are considered in the same region. Line 3∼13 compare the *end_addr* and *last_end_addr* to determine the new searching range. For example, in line 5, the current *end_addr* lies to the right of *last_end_addr*, it thus only needs to search the live words between them, then line 7 will add the result *delta* to *last_result* to get the total live words within the range from *beg_addr* to the current *end_addr*. Finally, it updates the variables for the reusing of next object.

However, how to effectively reuse previous results is not straightforward but depends on the query patterns of the references before the compaction. Based on our observations, the query patterns can be divided into two categories. The first one is a local pattern that the sequentially referenced objects tend to reside in the same region. For this pattern, the results of last queries could thus be easily reused. The second one is a random pattern that the referenced objects always lie in random regions, which makes it incapable to

Algorithm 2 Calculate live words within a range

```
Require: beg_addr, end_addr
 1: retrive last_beg_addr, last_end_addr, last_result
 2: if beg_addr = last_beg_addr then
 3:     if end_addr = last_end_addr then
 4:         live_bits ← last_result
 5:     else if end_addr > last_end_addr then
 6:         delta ← live_words_in_range(last_end_addr, end_addr)
 7:         live_bits ← last_result + delta
 8:     else
 9:         delta ← live_words_in_range(end_addr, last_end_addr)
10:         live_bits ← last_result − delta
11:     end if
12:     update last_end_addr, last_result with delta
13:     return live_bits
14: end if
15: live_bits ← live_words_in_range(beg_addr, end_addr)
16: update last_beg_addr, last_end_addr, last_result
17: return live_bits
```

reuse last results directly. Most applications are mixed with these two query patterns, differentiated by their respective proportions. To this end, we further propose three optimizing schemes accordingly to handle different situations.

3.2 Optimistic IQ

Targeting the applications with a high proportion of local query patterns, we propose a straightforward implementation of our basic idea: the optimistic incremental query. This approach complies with Algorithm 2, with each GC thread maintaining only one global result of last query for all the regions. Besides, when the object lies between the region start and the last object, which corresponds to the condition in line 8∼10 of Algorithm 2, its distances to both sides will be checked to make sure the shorter path is selected.

The optimistic IQ relies heavily on the local pattern to take good effect, while its advantage lies in the minimal overhead for both memory utilization and calculation.

3.3 Sort-based IQ

For the applications with most random query patterns, the optimistic IQ is not the best choice since the sequentially referenced objects tend to reside in various regions. We thus propose an approach called sort-based incremental query, whose key idea is to dynamically reorder the references based on their addresses with a lazy update.

The sort-based IQ employs a buffer with a fixed size. All the references are first filled into the buffer in batches before their updating. Each time the buffer fills up, the references within the buffer will be reordered according to the region index, and then be updated in their new sequences after the sorting. The buffer size is typically set close to the size of an L1 cache line for good locality.

With the sorting scheme, the references in the same region are gathered periodically, making it possible to effectively apply Algorithm 2. However, this approach may impose some overhead due to the extra sorting procedure.

3.4 Region-based IQ

Based on the insights of the two schemes as well as their deficiencies, we propose a region-based incremental query to embrace the best of the two schemes. This approach maintains a result of last query for each region per GC thread, thus can reasonably fit for both local and random query patterns. The region-based IQ applies Algorithm 2 within a region's bound.

We additionally employ a *slicing* scheme based on the concern that in some cases, the searching distance for a reference could span a large portion of the region width even with the reuse of the last result. Therefore, we reduce the searching range by dividing each region into multiple slices, maintaining the result of last query for each slice. For each referenced object, the region-based IQ checks the distances between it and last queried objects of two slices respectively: the current slice it resides in and the neighboring slice on the other side. The shortest searching path can thus be guaranteed with the slice-grained reuse of last queried result.

Minimize Overhead The region-based IQ is more aggressive than other two schemes and may impose some memory overhead due to the employment of slicing, especially with multiple GC threads. To minimize this overhead, we use a 16-bit integer to store the calculated size of live objects since it must be smaller than the region size (e.g., 4KB). Moreover, considering the region start will never be altered, we thus replace the 64-bit full-length address of the last queried object with the address offset to the region start, which only needs 12 bits and can perfectly fit in a 16-bit integer as well. By this way, the memory overhead is minimized to 0.09% of the entire heap size using the region-based IQ with one slice for each GC thread.

3.5 Parallelism

In our design, the results of last queries maintained by each GC thread are independent from each other, which can fundamentally eliminate the contention for updating the results after a bitmap searching completes. Our approach has no side effect to the multi-threading mechanism of the Parallel Scavenge GC.

4. Evaluation

4.1 Experimental Environments

Hardware/software platforms: We have implemented our optimization for OpenJDK 7u and later port it to version 8. The porting effort is trivial, which shows the portability of our optimization across versions. We evaluate our optimization on both Intel Xeon and Xeon Phi platform. The Xeon Phi Java support is based on OpenJDK 7u from our previous work [29]. The Xeon platform is an Intel Xeon E5-2620 CPU server with 6 cores at 2.0GHz frequency and 32GB memory space. The Xeon Phi coprocessor has 60 in-order cores at 1GHz, each of which has 4 hardware threads. There is no traditional shared last-level cache and the memory size is 8GB.

Benchmarks: We use standard benchmarks like JOlden, Dacapo, SPECjvm2008 as well as emerging big-data frameworks like Spark and Giragh for our study. We evaluate the memory-intensive applications from the suites to evaluate the GC behavior. As memory bloat is common for big-data applications [6, 13], we set the heap size (e.g, 1GB) close to

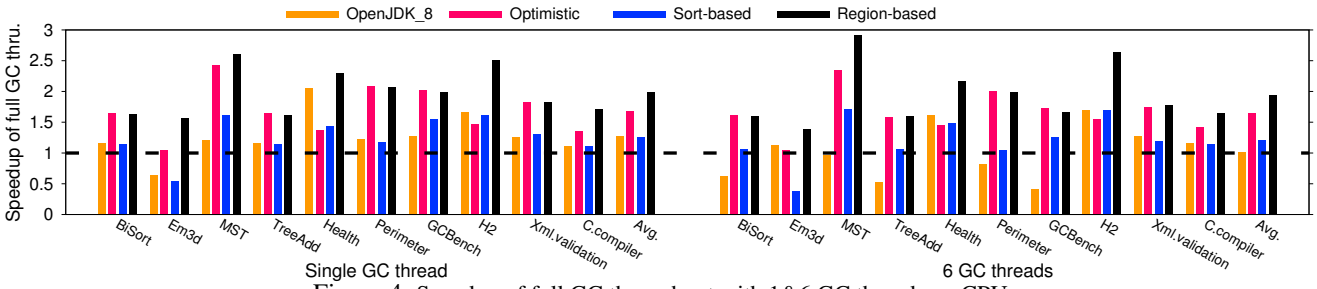


Figure 4: Speedup of full GC throughput with 1&6 GC threads on CPU.

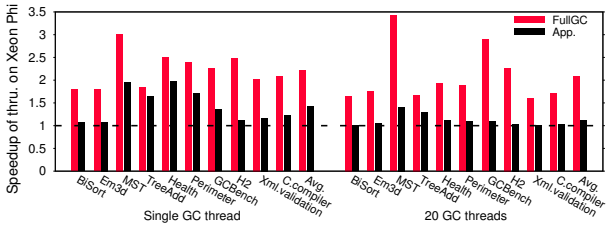
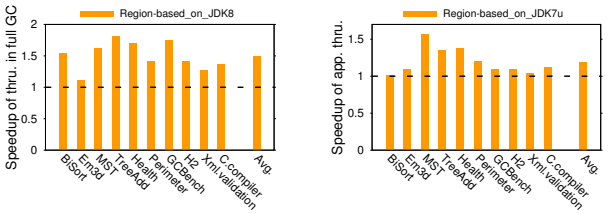


Figure 5: Speedup of full GC & app. thru. with 1&20 GC threads using region-based IQ on Xeon Phi



(a) Full GC thru. relative to JDK 8 (b) App. thru. relative to JDK 7u

Figure 6: Speedup of throughput with six GC threads on CPU

the workload size to emulate a memory-hungry environment for most of them to tax the GC, similar to prior work [11]. Besides, we evaluate Spark and Giraph with larger input and heap sizes (e.g., 10GB) as well to confirm the effectiveness of our optimization alone with increase of heap size.

GC setup: We present the throughput comparison of both full GC and application running with single and six GC threads on CPU and twenty for Xeon Phi. Twenty is the most appropriate thread count for Xeon Phi since the Parallel Scavenge does not scale well under a large number of GC threads [10]. Meanwhile, the memory overhead for the region-based IQ is about 3∼4% with twenty threads which is modest and acceptable. The performance is normalized to that of the original OpenJDK 7u version. We also provide the reduction in full GC time and the total pause time.

Application	Heap Size (GB)	Description
JOlden.BiSort	1	A bitonic sort
JOlden.Em3d	1	Models electromagnetic waves
JOlden.MST	1	Computes minimum spanning tree
JOlden.TreeAdd	1	Recursive DFS of a binary tree
JOlden.Health	3	Simulates health-care system
JOlden.Perimeter	1	Computes perimeter in image
GCBench	1	Build arrays and trees
Dacapo.H2	1	Executes transactions in H2
SPECjvm2008.X.v.	1	Validates an XML tree
SPECjvm2008.C.c.	1	Test javac compiler
Spark.pagerank	1&10	Rank websites
Giraph.sssp	1&10	Shortest path compution

Table 1: Benchmarks. (X.v & C.c in SPECjvm2008 refer to Xml.validation & Compiler.compiler)

4.2 Improvement on Standard Benchmarks

4.2.1 Speedup in Full GC

Figure 4 depicts the speedup of full GC throughput on Xeon server. The results cover the three different schemes (region-based with 2 slices). Besides, OpenJDK 8 has made some revision in region management by separating each region into multiple blocks. To study its performance impact, we provide a performance comparison with our optimization.

We can see that the optimistic and region-based IQ could both achieve significant speedup for most benchmarks. The two schemes perform similarly except for Em3d, Health, H2 and C.compiler. This is because these four applications mainly consist of random query patterns during compaction, which do not fit well for optimistic IQ. The sort-based IQ has inferior performance compared to the other two, which indicates: 1) a majority of sequentially updated references conform to a local pattern; 2) the overhead of sorting is significant, which makes the sort-based IQ less attracting.

When scaling up to six GC threads, there shows no significant reduction in speedup. This confirms that our approach has no side effect to the multi-threading mechanism. The region-based IQ achieves up to 2.9x and averagely 1.9x speedup for full GC throughput on CPU. Besides, by using the optimization in section 3.4, the memory overhead can be reduced to only 1.1% with six GC threads.

As illustrated Figure 5, the best-performed region-based IQ achieves a full GC speedup of 3.4x for *MST* and averagely 2.1x with twenty GC threads on Xeon Phi. The results are mostly consistent to those of the Xeon server.

Portability of our optimization: We also compared the port of our region-based IQ to OpenJDK 8 with the vanilla GC. As shown in Figure 4, the new revision of OpenJDK 8 performs even worse with multiple GC threads for many benchmarks than the original 7u version. However, as shown in Figure 6a, our region-based IQ could still achieve significant performance improvement, which can reach up to 1.8x for *TreeAdd* and averagely 1.5x speedup. This result confirms the portability of our optimization.

4.2.2 Application Performance Improvement

The three approaches have incremental performance, e.g., region-based IQ outperforms the other two. Despite some minor overhead, we believe that in most cases, the region-

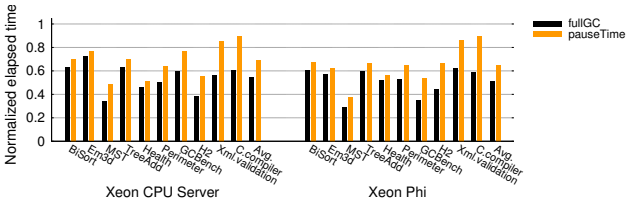
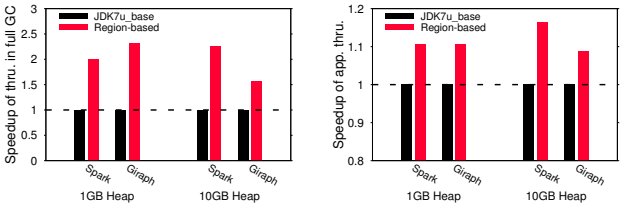


Figure 7: Normalized elapsed time. Lower is better



(a) Full GC (b) Application

Figure 8: Throughput speedup of big-data applications with varying input and heap sizes

based IQ performs best and should be the default choice. Thus, we mainly use the it to demonstrate the speedup in the applications' throughput.

Figure 6b plots the improvement with six GC threads on CPU. While the proportions of full GC time vary across different applications, the performance improvement is remarkable for most benchmarks, which is up to 57.2% and averagely 19.3% with six GC threads.

As shown in Figure 5, the region-based IQ can bring up to 41.2% improvement in the application's throughput for the MST program with twenty GC threads on Xeon Phi. In general, we could achieve averagely 11.1% improvement for the applications under the multi-GC-threading environment.

Reduction in pause time: Figure 7 depicts the reduction in full GC time and total pause time by using the region-based IQ with multiple GC threads, compared to the original Parallel Scavenge in OpenJDK 7u. Due the significant performance improvement in full GC, the total pause time is notably reduced, which can reach averagely 31.2% and 34.9% for CPU and Xeon Phi respectively.

4.3 Improvement on Big-data Applications

We further use Spark and Giraph to show the effectiveness of our optimization for real-world applications, with both normal (i.e., 1GB) and larger (i.e., 10GB) heap sizes. The input data size is also increased, which is from 100MB to 2.6GB for Spark, and 400MB to 4GB for Giraph. The normalized throughputs relative to the original PS collector in OpenJDK 7u are provided in Figure 8 for both full GC and application execution.

A significant performance speedup comparable to the standard benchmarks in full GC can be observed with varying heap sizes. For the application execution time, the improvement can steadily reach about 10%, which is typically up to 16.3% for Spark with 10GB heap size.

Our optimization is orthogonal to distributed execution. To confirm this, we conducted a small-scale evaluation on a 5-node cluster. Each machine has two 10-core Intel Xeon E5-2650 v3 processors and 64GB DRAM. We ran Spark PageRank with 100 million edges and configured 10GB heap size on each node. We recorded the accumulated full GC time for all nodes and the elapsed application time on master. The improvement on full GC and application throughput is 63.8% and 7.3%, respectively. The speedup is smaller due to network communication becomes a more dominating factor during distributed execution. Yet, a still significant reduction in full GC time may help a lot in reducing tail latency [9, 15].

5. Related Work

There is a long thread of research to improve the performance of garbage collection, especially the mark-compact collector [2, 8, 18, 24]. For example, Abuaiadh et al. [2] proposed a heap compaction algorithm for SMP platforms, which saves information for a pack of objects instead of using standard forwarding pointer mechanism for updating references. Chung et al. [8] proposed a sweeping approach that traverses only the live objects so that the sweeping time depends only on the number of live objects in the heap. Our approach also only scans live objects but further reuses the prior scanning result. Morikawa et al. [18] described an adaptive scanning method between scanning bitmap and scanning heap for Lisp2 compactor collector. However, they did not consider the reuse of previous scanning result. Our optimization aligns with this thread of research by pinpointing a major bottleneck of the compaction phase and designing a novel solution to significantly boost the performance.

Some efforts have been made on the throughput-oriented Parallel Scavenge garbage collector. Gidra et al. [11, 12] studied the scalability issues of Parallel Scavenge on NUMA systems, and presented a NUMA-aware design which can maximize the memory access locality during collection. Some recent work advocates of using coordination to mitigate garbage collection for big data runtime [15, 19]. Specifically, Murray et al. [19] leveraged event information to partition objects into different regions for better memory management. Maas et al. [15] showed that coordination of GC among multicore nodes may reduce the impact of GC on big data performance. Our work is orthogonal to such work and could further reduce the GC impact.

Xu et al. [21, 22] performed a lot of work based on the problem of runtime bloat. They witnessed significant performance impact and severe pressure on the garbage collector, which is caused by the memory bloat in a managed runtime like JVM. They proposed an application-level approach to tackle this problem, such as a bloat-aware design paradigm towards the development in GC enabled languages to eliminate the bloat for large-scale data-intensive applications [6].

6. Conclusion

This section presented a detailed study on the full GC performance using standard and big-data applications. Our study found that full GC still contributes a non-trivial portion of application execution time. Our profiling found that the reference updating in the compaction phase is a major performance killer. To this end, this paper presented an incremental query model with three schemes. Our evaluation confirms the benefit of our optimization.

References

[1] SPECjvm2008. https://www.spec.org/jvm2008/, 2015.

[2] D. Abuaiadh, Y. Ossia, E. Petrank, and U. Silbershtein. An efficient parallel heap compaction algorithm. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 224–236, New York, NY, USA, 2004. ACM. ISBN 1-58113-831-8. . URL http://doi.acm.org/10.1145/1028976.1028995.

[3] Apache. Apache giraph: an iterative graph processing system built for high scalability. http://giraph.apache.org/.

[4] S. M. Blackburn and K. S. McKinley. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 22–32, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. . URL http://doi.acm.org/10.1145/1375581.1375586.

[5] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4. . URL http://doi.acm.org/10.1145/1167473.1167488.

[6] Y. Bu, V. Borkar, G. Xu, and M. J. Carey. A bloat-aware design for big data applications. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, pages 119–130, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2100-6. . URL http://doi.acm.org/10.1145/2464157.2466485.

[7] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in java. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, PACT '01, pages 280–291, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1363-8. URL http://dl.acm.org/citation.cfm?id=645988.674177.

[8] Y. C. Chung, S.-M. Moon, K. Ebcioğlu, and D. Sahlin. Reducing sweep time for a nearly empty heap. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 378–389, New York, NY, USA, 2000. ACM. ISBN 1-58113-125-9. . URL http://doi.acm.org/10.1145/325694.325744.

[9] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, Feb. 2013. ISSN 0001-0782. . URL http://doi.acm.org/10.1145/2408776.2408794.

[10] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. Assessing the scalability of garbage collectors on many cores. *SIGOPS Oper. Syst. Rev.*, 45(3):15–19, Jan. 2012. ISSN 0163-5980. . URL http://doi.acm.org/10.1145/2094091.2094096.

[11] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. 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*, ASPLOS '13, pages 229–240, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1870-9. . URL http://doi.acm.org/10.1145/2451116.2451142.

[12] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen. 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*, ASPLOS '15, pages 661–673, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2835-7. . URL http://doi.acm.org/10.1145/2694344.2694361.

[13] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingan, D. Murray, S. Hand, and M. Isard. Broom: Sweeping out garbage collection from big data systems. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 2–2, Berkeley, CA, USA, 2015. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2831090.2831092.

[14] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011. ISBN 1420082795, 9781420082791.

[15] M. Maas, T. Harris, K. Asanovic, and J. Kubiatowicz. Trash day: Coordinating garbage collection in distributed systems. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 1–1, Berkeley, CA, USA, 2015. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2831090.2831091.

[16] S. Microystems. Memory management in the java hotspot virtual machine, 2006.

[17] J. E. Moreira, S. P. Midkiff, M. Gupta, P. Wu, G. Almasi, and P. Artigas. Ninja: Java for high performance numerical computing. *Sci. Program.*, 10(1): 19–33, Jan. 2002. ISSN 1058-9244. . URL http://dx.doi.org/10.1155/2002/314103.

[18] K. Morikawa, T. Ugawa, and H. Iwasaki. Adaptive scanning reduces sweep time for the lisp2 mark-compact garbage collector. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, pages 15–26, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2100-6. . URL http://doi.acm.org/10.1145/2464157.2466480.

[19] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. . URL http://doi.acm.org/10.1145/2517349.2522738.

[20] R. M. Muthukumar and D. Janakiram. Yama: A scalable generational garbage collector for java in multiprocessor systems. *IEEE Trans. Parallel Distrib. Syst.*, 17 (2):148–159, Feb. 2006. ISSN 1045-9219. . URL http://dx.doi.org/10.1109/TPDS.2006.28.

[21] K. Nguyen and G. Xu. Cachetor: Detecting cacheable data to remove bloat. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 268–278, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2237-9. . URL http://doi.acm.org/10.1145/2491411.2491416.

[22] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu. Facade: A compiler and runtime for (almost) object-bounded big data applications. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 675–690, New York, NY,

USA, 2015. ACM. ISBN 978-1-4503-2835-7. . URL http://doi.acm.org/10.1145/2694344.2694345.

[23] N. Sachindran, J. E. B. Moss, and E. D. Berger. Mc2: High-performance garbage collection for memory-constrained environments. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 81–98, New York, NY, USA, 2004. ACM. ISBN 1-58113-831-8. . URL http://doi.acm.org/10.1145/1028976.1028984.

[24] V. Sarkar and J. Dolby. High-performance scalable java virtual machines. In *Proceedings of the 8th International Conference on High Performance Computing*, HiPC '01, pages 151–166, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-43009-1. URL http://dl.acm.org/citation.cfm?id=645447.652938.

[25] S. Soman, C. Krintz, and L. Daynès. Mtm2: Scalable memory management for multi-tasking managed runtime environments. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 335–361, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70591-8. . URL http://dx.doi.org/10.1007/978-3-540-70592-5_15.

[26] Spark. Apache spark is a fast and general engine for large-scale data processing. http://spark.apache.org/, 2015.

[27] G. L. Taboada, S. Ramos, R. R. Expósito, J. Touriño, and R. Doallo. Java in the high performance computing arena: Research, practice and experience. *Sci. Comput. Program.*, 78(5):425–444, May 2013. ISSN 0167-6423. . URL http://dx.doi.org/10.1016/j.scico.2011.06.002.

[28] D. Vengerov. Modeling, analysis and throughput optimization of a generational garbage collector. In *Proceedings of the 2009 International Symposium on Memory Management*, ISMM '09, pages 1–9, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-347-1. . URL http://doi.acm.org/10.1145/1542431.1542433.

[29] Y. Yu, T. Lei, H. Chen, and B. Zang. Openjdk meets xeon phi: A comprehensive study of java hpc on intel many-core architecture. In *Parallel Processing Workshops (ICPPW), 2015 44th International Conference on*, pages 156–165. IEEE, 2015.

[30] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2228298.2228301.

[31] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 379–391, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1994-2. . URL http://doi.acm.org/10.1145/2465351.2465388.