ScissorGC: Scalable and Efficient Compaction for Java Full Garbage Collection

Haoyu Li, Mingyu Wu, Bingyu Zang, Haibo Chen
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
{haoyu.li,byzang,haibochen}@sjtu.edu.cn,mingyuwu93@gmail.com

Abstract

Java runtime frees applications from manual memory management through automatic garbage collection (GC). This, however, is usually at the cost of stop-the-world pauses. State-of-the-art collectors leverage multiple generations, which will inevitably suffer from a full GC phase scanning and compacting the whole heap. This induces a pause tens of times longer than normal collections, which largely affects both throughput and latency of applications.

In this paper, we comprehensively analyze the full GC performance of the Parallel Scavenge garbage collector in HotSpot. We find that chain-like dependencies among heap regions cause low thread utilization and poor scalability. Furthermore, many heap regions are filled with live objects (referred to as *dense regions*), which are unnecessary to collect. To address these two problems, we provide *ScissorGC*, which contains two main optimizations: dynamically allocating shadow regions as compaction destinations to eliminate region dependencies and skipping dense regions to reduce GC workload. Evaluation results against the HotSpot JVM of OpenJDK 8/11 show that *ScissorGC* works on most benchmarks and leads to 5.6X/5.1X improvement at best in full GC throughput and thereby boost the application performance by up to 61.8%/49.0%.

CCS Concepts • **General and reference → Performance**; • **Software and its engineering → Garbage collection**; **Multithreading**;

Keywords Full garbage collection, Java virtual machine, Performance, Parallel Scavenge, Memory management

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
VEE '19, April 13–14, 2019, Providence, RI, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6020-3/19/04…$15.00
https://doi.org/10.1145/3313808.3313820

1 Introduction

Java is steadily adopted by various kinds of applications due to its virtues such as powerful functionalities, strong reliability, and multi-platform portability, mainly thanks to its underlying runtime, the Java Virtual Machine (JVM). Automatic memory management, or garbage collection (GC), is a crucial module provided by the JVM to free programmers from manual deallocation of memory and thereby guarantees both usability and memory safety. However, GC comes with a cost. Most state-of-the-art garbage collectors leverage the stop-the-world (STW) method for collection: when GC starts, application threads will be suspended until all dead objects have been reclaimed. Although mainstream collectors exploit the generational design [25] so that most collections only touch a small portion of the heap and finish quickly, they inevitably have to enter a stage called *full GC* to collect the whole heap space and thus incur considerable pause time. State-of-the-art concurrent collectors like G1 [6] also contain a similar full GC cycle to handle the case where memory resource is nearly exhausted [18]. The problem is even aggravated in today's prevalent memory-intensive frameworks like Spark [29] in that they have a large demand for memory, which induces more frequent full GC cycles and longer pauses.

In this paper, we provide a comprehensive analysis of the full GC part of Parallel Scavenge Garbage Collector (PSGC), the default collector in the HotSpot JVM. We find that full GC spends the most time on the *compacting phase*, which moves objects to the head of the heap to vacate a large continuous free space. The compaction algorithm of PSGC divides the heap into *regions* and assigns them as *tasks* to multiple GC threads for concurrent processing. Our analysis uncovers two problems in this algorithm. First, the multi-threading compaction suffers from poor thread utilization due to chain-like dependencies among regions. Second, memory-intensive applications introduce a large number of regions filled up with live objects (referred to as *dense regions*), and blindly compacting them cannot collect any memory space within those regions but only cause data copy overhead. Those two problems make the original algorithm non-scalable and inefficient.

To this end, we propose *ScissorGC*, a new compaction algorithm with two corresponding optimization techniques. For the thread utilization problem, we introduce *shadow regions* to eliminate region-wise dependencies and in turn enable

more threads to run in parallel. For the compaction efficiency problem, we provide a *region skipping* design to avoid compacting and moving dense regions. We have also discussed the design tradeoff of *ScissorGC* and provided several techniques to mitigate the overhead *ScissorGC* introduces.

We have implemented *ScissorGC* in the HotSpot JVM of OpenJDK 8 and compared it with the vanilla PSGC over applications from various benchmark suites. The evaluation result confirms that our optimizations make full GC much more scalable and improve the full GC throughput for most applications by up to 5.6X and thereby shorten the execution time of most applications by up to 61.8%. We have also ported *ScissorGC* into the latest OpenJDK 11 to integrate with other optimizations on full GC [27] and improve the full GC throughput by up to 5.1X and the application throughput by up to 49.0%. The source code of *ScissorGC* is available at **https://ipads.se.sjtu.edu.cn:1312/opensource/scissorgc**.

To summarize, the contributions of this paper include:

- A detailed analysis of the compaction algorithm of full GC to uncover the sources of its inefficiency. (§3)
- *ScissorGC*, a new compaction algorithm with two corresponding optimization techniques to resolve the performance problems. (§4)
- A thorough evaluation with various applications to confirm our optimizations actually work. (§5)

2 Background

2.1 Parallel Scavenge

Parallel Scavenge Garbage Collector (PSGC) is the default garbage collector in the HotSpot JVM of OpenJDK 8. It collects objects in a stop-the-world fashion: when a GC cycle starts, all mutator threads must be paused and GC threads will take over. Mutators cannot be executed until all GC threads have finished their work. This design avoids complicated coordination between mutators and GC threads and in turn reaches satisfying GC throughput, but it may greatly affect the application latency.

```
1  class Region {
2    // Fields in vanilla PSGC
3    int dcount;
4    int live_obj_size;
5    char* dest_addr;
6    Region*[] src_regions;
7    ...
8
9    // Added by ScissorGC
10   int shadow;
11   int status;
12 }
```

Figure 1. The definition of *Region* in PS full GC

PSGC divides the heap into two spaces: a *young space* for object creation and an *old space* to store long-lived objects. The GC algorithm is also two-fold. *Young GC* is triggered when the young space is used up and only collects the young space. *Full GC* happens when the memory resource of the whole heap is exhausted and collects both spaces. To reach a satisfying application latency, young space is usually designed as a small fraction of the entire heap so that young GC happens frequently but finishes quickly. However, when full GC must be initiated, mutators will experience a much longer pause. Worse yet, full GC happens quite frequently in memory-intensive applications. Our evaluation on Spark with a 20GB heap shows that full GC cycles happen every 10 seconds and last for 3.35 seconds on average. Meanwhile, the worst-case pause time is 11.4X that of a young GC cycle. Therefore, the primary concern of our work is to mitigate the prohibitive pauses caused by full GC.

2.2 Full GC algorithm

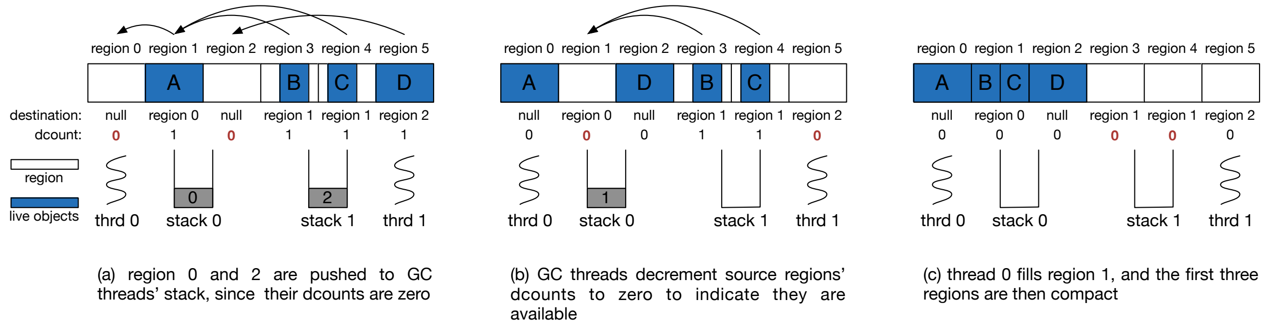
Full GC is a multi-threaded, compaction-based algorithm that copies all live objects into the beginning of the old space to vacate a large continuous free memory space. PSGC implements full GC as a three-phase algorithm, including marking phase, summary phase, and compacting phase. These phases are region-based: PSGC partitions the JVM heap into continuous *regions* of equal size (512KB by default in PSGC) and compacts live objects within the same region together. The data structure of *Region* is shown in Figure 1. We next briefly explain these three phases below.

Marking phase. In the first marking phase, GC threads will search for live objects from known *roots*, such as on-stack references and static variables. All reachable objects from roots will be marked as alive, and their sizes will be added to the *live_obj_size* of the very regions.

Summary phase. After the marking phase, PSGC will calculate a heap summary for all regions based on their *live_obj_size*. After the summary phase, a mapping between regions is generated with the *source_regions* field so that each *source region* will have its own *destination regions*[1] for object copying. The summary phase also generates per-region indices, i.e., *dest_addr*, with which GC threads can calculate the destination address of any live objects inside a region.

Compacting phase. Live objects will not be moved or modified until the last compacting phase. Since the compacting phase is costly and usually accounts for over 70% of the overall full GC time, we will focus on optimizing this phase in this work. The compacting phase is still region-based: each destination region stands for a *task*, and GC threads will concurrently fetch destination regions and fill them up with live objects from their corresponding source regions. Reference updates for live objects also occur after copying

[1] A source region can have one or two destination regions.

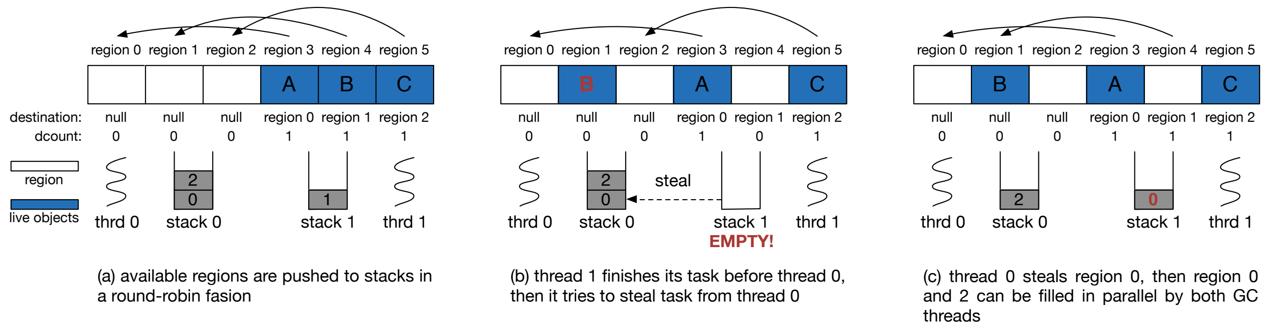


(a) region 0 and 2 are pushed to GC threads' stack, since their dcounts are zero

(b) GC threads decrement source regions' dcounts to zero to indicate they are available

(c) thread 0 fills region 1, and the first three regions are then compact

Figure 2. An example of the compacting phase in full GC



(a) available regions are pushed to stacks in a round-robin fasion

(b) thread 1 finishes its task before thread 0, then it tries to steal task from thread 0

(c) thread 0 steals region 0, then region 0 and 2 can be filled in parallel by both GC threads

Figure 3. An example of the work stealing in full GC

them to destination regions. Since destination regions themselves are also source regions for others, the GC threads must not process them until all live objects within them are evacuated to their destinations. This processing order is preserved by maintaining a variable for each region named destination count, or *dcount*, to memorize how many destination regions (excluding itself) depend on it. The summary phase will calculate the initial *dcount* for each region. In the compacting phase, once a GC thread fills up a destination region, it will decrement *dcount* by one for all corresponding source regions. When a region's *dcount* reaches zero, all live objects within it have been evacuated so it can serve as a destination region. It will thereby be pushed into the working stack of the GC thread which decrements its dcount to zero and poised to accept live objects from its source regions. Those dependencies among regions can be used to construct a dependency graph to shape the execution behavior of GC threads.

We provide an example in Figure 2 to further illustrate the compaction algorithm. This example consists of six heap regions and two available GC threads. At the summary phase, since region 0 and region 2 contain no live objects, they depend on nothing and get a zero *dcount*. Other regions, on the contrary, keep a non-zero *dcount* and their own dependencies. When the compacting phase starts, regions with a zero *dcount* (i.e., region 0 and 2) will be assigned to the working stack of thread 0 and 1 respectively for processing (Figure 2.a). Those two GC threads will copy live objects from other regions to the under-processed ones, according to the per-region indices generated in the summary phase. After region 0 is processed by thread 0, the *dcount* of region 1 will be decreased to 0, which makes region 1 available for processing. Since region 1 initially depends on region 0 and should serve as a source region for region 3 and 4, it will be pushed to the stack of thread 0 (as shown in Figure 2.b). On the contrary, when thread 1 finishes processing region 2, region 5 will not be pushed to the working stack as it is the last region in this example and no region relies on it. Similarly, when the task on region 1 is complete, region 3 and 4 will not be processed; all live objects have been compacted at that time and the compacting phase terminates (Figure 2.c).

Optimizations. Since not all destination regions are available at the beginning of compaction, PSGC can only assign those available regions to GC threads initially. This design, unfortunately, is likely to result in a load imbalance. Consider the example in Figure 3.a, where region 0, 1, and 2 are relied on by subsequent regions (region 3, 4, and 5). Since the first three regions are available at the beginning of compaction, they will be assigned to two GC threads in a round-robin manner. If thread 1 runs much faster than thread 0 and finishes copying objects from region 4 to region 1 in advance,

it will become idle since no regions depend on region 4 (Figure 3.b). To avoid starvation of GC threads, full GC enables work stealing so that GC threads can steal available regions from other threads after their own working stacks are fully drained. In this example, thread 1 will fetch region 0 from thread 0's working stack to achieve load balance (Figure 3.c).

Full GC also specially handles regions at the beginning of the heap. If most objects in those regions are alive (named *dense regions* in PSGC), the benefit of moving them forward is diminishing. Consequently, full GC will instead organize them as a *dense prefix* and avoid moving any objects therein. Although dense prefixes can avoid data copying, they can only be used to optimize regions at the head of a heap.

3 Analysis

3.1 Thread Utilization

Although the compacting phase supports multi-threading, our results show that it suffers from low thread utilization and induces poor scalability in many applications. We evaluate Derby [31] in the SPECjvm2008 benchmark suite [23] as an example to support our claim. Figure 4 shows the execution behavior of 16 GC threads (x-axis) in the compacting phase over time (y-axis). The result indicates that all threads only spend a small fraction of time (8.0% on average) working on compaction (colored bars) while wasting CPU resources on stealing in vain (blanks). The terrible utilization rate also causes poor scalability (shown in Section 5.5) and strongly motivates us to optimize it.

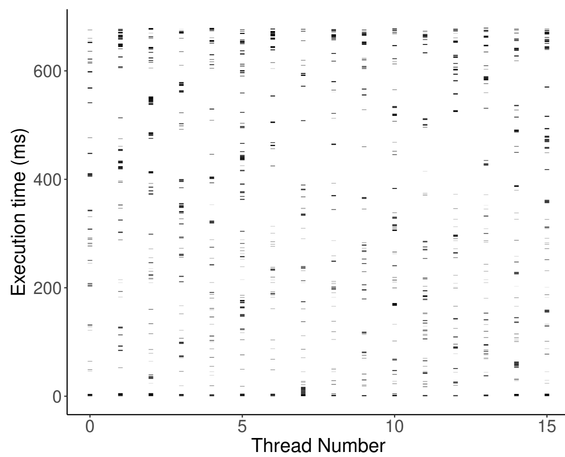


Figure 4. Thread utilization in the full GC compacting phase of Derby

We have further profiled work stealing by dumping the working stack for every thread and find that all except one stack are empty for most of the time. The non-empty stack, however, has only one task being processed, which is not available for stealing. As mentioned in Section 2.2, a destination region cannot be processed until its *dcount* reaches zero. Therefore, the Derby scenario happens when the dependency graph contains long *dependency chains* as illustrated in Figure 5. This simplified case comprises four regions and all regions have only one destination, which forms a dependency chain, where only one region is available for processing at any time. Consequently, other threads will always fail to steal due to lack of available tasks.

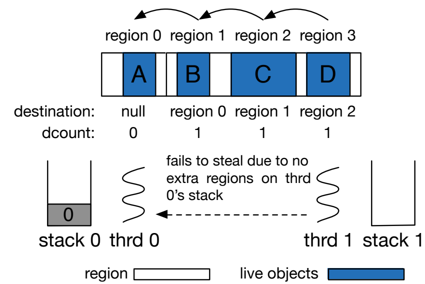


Figure 5. An example of dependency chains, the culprit for poor thread utilization

3.2 Compaction Efficiency

Memory-intensive applications like Spark have a keen demand for memory resources and request tens or hundreds of gigabytes for their Java heaps, which introduces tens of thousands of destination regions during full GC. To process such a number of regions without inducing large pauses, the compaction efficiency must be greatly optimized.

We have used Spark as an example of memory-intensive applications to study their memory behaviors. When running the built-in page rank application on Spark with a 20GB heap, we collect the region-related statistics at each full GC cycle and the observations are described below.

Dense regions are quite common. In our evaluation, 49.4% of heap regions on average are dense regions. We have even observed extreme cases where dense regions occupy over 80% of the whole heap. Consider region 1 in Figure 2.a, since it only contains live objects, compacting it is meaningless for it does not help to reclaim any memory space. Even worse, the compaction algorithm has to move the whole region forward and cause considerable data copy overhead (copying to region 0 in this example). Therefore, specially processing those dense regions can greatly optimize the compaction efficiency.

Previous optimizations fail to help. PSGC has provided the dense prefix optimization (mentioned in Section 2.2) to avoid compacting dense regions at the beginning of the heap. Unfortunately, dense regions are dispersed throughout the whole heap, so many of them cannot benefit from the optimization and thus suffer from unnecessary compaction. Our evaluation shows that 60.5% of dense regions are outside the dense prefix, which suggests great potential for further optimizations.

Large objects are the culprit for dense regions. We have also looked into each dense region to study its composition. We observed that 74.5% of dense regions are part of large objects. Those large objects are usually data arrays, such as arrays of doubles, integers, long integers, and Scala *Tuples*[2]. This happens because applications like Spark are usually based on huge datasets, which generate many large arrays whose size can span multiple heap regions. The largest array object in our test consumes more than 18MB heap space, which introduces 36 consecutive dense regions. Furthermore, those objects live long because Spark allows users to cache huge datasets in the heap memory for future reuse [28], so they will survive many GC cycles and finally induce a large number of dense regions in full GC. The cache behavior is very common in today's memory-intensive frameworks [4, 7, 11].

To conclude, memory-intensive applications usually introduce many dense regions in full GC and compacting them is not effective to reclaim memory resource. However, the current compaction algorithm still blindly processes most dense regions as normal ones, which induces poor compaction efficiency. This observation motivates us to avoid compacting and moving dense regions for better performance.

4 Optimizations

As analyzed in Section 3, there are two major performance issues in the compaction algorithm of full GC: limited thread utilization due to chain-like region dependencies and inefficient compaction for dense regions. To resolve these problems, we introduce *ScissorGC* in this section, which contains two optimization techniques: shadow regions and region skipping.

4.1 Shadow Region

Basic idea. The goal of the shadow region optimization is to allow threads to steal unavailable regions. Specifically, when a GC thread encounters a stealing failure, it will turn to unavailable regions rather than spin or sleep. Since an unavailable region still contains live objects and cannot serve as a destination for the moment, the GC thread needs to allocate a shadow region as its substitute and directly copy live objects into the shadow one. Live objects within the shadow region will be copied back once the destination count of the corresponding stolen region reaches zero. This design improves the success rate of work stealing and thereby achieves better thread utilization.

Implementation. We implement shadow regions as a complement to work stealing. In PS full GC, work stealing is implemented as *StealTask*: when a GC thread cannot find any available regions to process, it will fetch a StealTask to start stealing. If the GC thread manages to steal a region from others, it will process the region and drain its own working stack (line 3-5 of Algorithm 1). When encountering a stealing failure, GC threads will spin or fall asleep in the original design (line 9-10). We instead have modified its logic to search for an unavailable region to fill with the help of shadow regions (line 6-8).

To fill an unavailable region, GC threads first acquire a shadow region (line 16) and copy live objects from source regions to the shadow one until it is full (line 17-19). This

Algorithm 1 The algorithm of shadow region

```
1:  function StealTask
2:      while there is still other task running do
3:          if T steals an available region then
4:              fill region
5:              drain the working stack of T
6:          else if T finds an unavailable region then
7:              FillShadowRegion(region)
8:              drain the working stack of T
9:          else
10:             T spins or falls asleep
11:         end if
12:     end while
13: end function
14:
15: function FillShadowRegion(region)
16:     region.shadow = acquire a shadow region
17:     while region.shadow is not full do
18:         next_src = the next source region of region
19:         MoveObject(next_src, region.shadow)
20:         DecrementDestinationCount(next_src)
21:     end while
22:     if region.dcount == 0 then
23:         CopyBack(region)
24:     end if
25: end function
26:
27: function DecrementDestinationCount(region)
28:     region.dcount--
29:     if region.dcount == 0 then
30:         if region.shadow == NULL then
31:             push region to the working stack of T
32:         else if region.shadow is full then
33:             CopyBack(region)
34:         end if
35:     end if
36: end function
37:
38: function CopyBack(region)
39:     copy the data of region.shadow back to region
40:     release the shadow region region.shadow
41:     region.shadow = NULL
42: end function
```

[2] *Tuples* are prevalently used in Scala. Programmers can easily define a two-dimensional tuple with code like "x = (10, 20)".

step also memorizes the shadow region in a newly-added per-region field, *shadow*. When all live objects of a source region have been evacuated to the corresponding shadow regions, the GC thread should decrement the destination count for this source region (line 20), and push the region into its working stack if it becomes available (i.e. dcount becomes 0). After filling the shadow region, GC threads will copy data therein back to its corresponding heap region if that region has become available (line 22-24). Otherwise, the data will be copied back once the heap region turns available (line 32-34).

Discussions. Since multiple GC threads may fetch the same unavailable region simultaneously (line 6-7 and line 30-31), ScissorGC atomically marks regions to avoid repeated processing. To this end, ScissorGC introduces a variable named *status* for each region (see Figure 1), and GC threads leverage atomic Compare-And-Swap (CAS) instructions to update this variable to mark the corresponding region as being processed. Those atomic instructions guarantee that each destination region is processed exactly once by GC threads, and they are cheap compared to other synchronization primitives such as locks.

To handle the requests for shadow region allocation, a straw-man design is to allocate off-heap memory. However, this strategy may consume too much memory, and the write latency of newly allocated shadow regions is observably slower than that of heap regions due to worse data locality. Fortunately, we find an interesting feature in young GC, which guides us to reuse regions in the young space for less memory consumption and better locality.

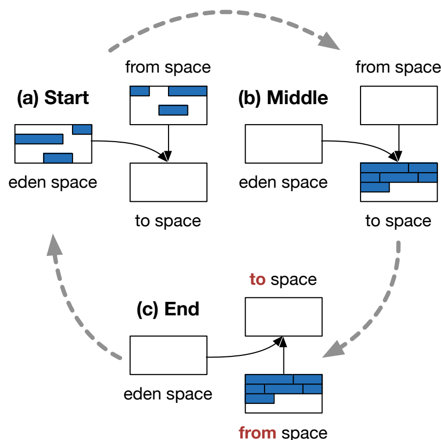


Figure 6. A brief introduction to young GC

The young space in PSGC is divided into three parts: an *eden space* to serve for memory allocation, and two *survivor spaces* to store objects which have survived at least one GC cycle.

Consider Figure 6.(a), when young GC starts, one empty survivor space (known as *to space*) will accept objects from the eden space and the other non-empty survivor space[3] (known as *from space*). During this young GC cycle, live objects in from space will be evacuated, then it becomes empty while *to space* contains live objects (shown in Figure 6.b).

Therefore, the role of those two survivor spaces will be swapped at the end of young GC (as shown in Figure 6.c), and live objects will be copied to the new *to space* in the next young GC cycle. The observation is that only one survivor space at a time actually contains live objects. We thereby always leverage the empty survivor space to allocate shadow regions (line 16). In the case where the survivor space is not enough, we will fall back to the slow path and allocate some extra shadow regions out of Java heap memory. Once objects in a shadow region have been copied back to heap regions, it will be recycled into a LIFO region stack for future reuse (line 40). These optimizations can improve both memory efficiency and data locality (shown in Section 5.6).

To find an unavailable region to fill, a GC thread linearly scans the heap from its last processed region and picks the first unavailable one (line 6). It is possible to apply more sophisticated heuristics to maximize the benefit of shadow regions. For example, GC threads can choose a region heavily depended by other ones so as to generate available regions as many as possible for other threads to steal. This design indeed reduces the usage of shadow regions to avoid additional overhead, but it may also introduce heavy computation and more metadata maintenance. Therefore, we decide to exploit the previously mentioned mechanism due to its satisfying efficiency.

4.2 Region Skipping

Basic idea. The basic idea of region skipping is to avoid data movement for dense regions. To achieve this goal, we need to find out all the dense regions and skip over them when establishing mappings between destination and source regions in the summary phase. Those regions will not be moved in the subsequent compacting phase, and the compaction efficiency can be boosted if a considerable number of regions is skipped.

Implementation. Algorithm 2 presents the pseudocode of the region skipping. In the summary phase, PSGC traverses all the regions to establish the source-to-destination mapping (line 3-12). We have modified the summary phase to record all the dense regions (line 2), and exclude them from the mapping so that they will not be moved by the subsequent compacting phase. The branch code in line 5 only allows regions which are not dense to become a source region for others, while the function *SkipDenseRegion* in line 10 avoids assigning a dense region as a destination by skipping its address range.

[3]When objects have survived many young GC cycles, they will be copied to *old space* instead. We elide this case here for simplicity.

Algorithm 2 The algorithm of region skipping

```
1:  function SummaryPhase(beg_region, end_region)
2:      find out all the dense regions
3:      cur_addr = the start address of beg_region
4:      for region = beg_region → end_region do
5:          if region is not dense then
6:              cur_region = the region contains cur_addr
7:              add region to cur_region.src_regions
8:              region.dest_addr = cur_addr
9:              cur_addr += region.live_obj_size
10:             cur_addr = SkipDenseRegion(cur_addr)
11:         end if
12:     end for
13:     shadow_area = acquire a shadow area
14: end function
15:
16: function MoveObject(src_region, dest_region)
17:     repeat
18:         live_obj = next live object in src_region
19:         if live_obj is an overflowing object then
20:             store live_obj in shadow_area as new_obj
21:         else
22:             move live_obj to dest_region as new_obj
23:         end if
24:         UpdateReference(new_obj)
25:     until dest_region is full or src_region is empty
26: end function
27:
28: function UpdateReference(obj)
29:     for all ref in obj do
30:         *ref = CalcObjDestAddr(*ref)
31:     end for
32: end function
33:
34: function CalcObjDestAddr(obj)
35:     region = the region that contains obj
36:     if region is dense then
37:         return obj
38:     else if obj is an overflowing object then
39:         return heap end +
40:                 offset of obj in shadow_area
41:     end if
42:     offset = live bytes before obj in region
43:     result = region.dest_addr + offset
44:     return SkipDenseRegion(result)
45: end function
46:
47: function SkipDenseRegion(addr)
48:     while addr is in a dense region do
49:         addr += size of the dense region
50:     end while
51:     return addr
52: end function
```

After moving a live object in the compacting phase (the function *MoveObject*), the GC thread needs to update its object references to the calculated destination addresses of their referents (line 24). Since objects in dense regions will not be moved, the destination calculation just returns their original addresses (line 36-37). Otherwise, *ScissorGC* computes the destination address and adjusts the result to skip dense regions (line 42-44).

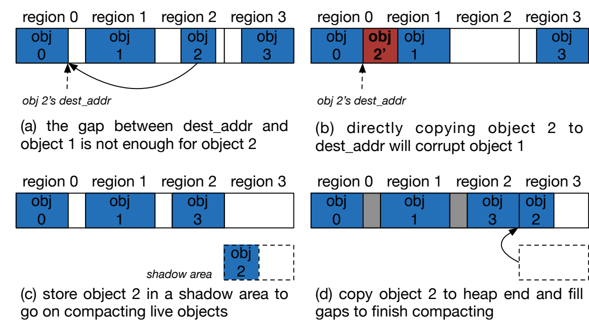


Figure 7. An example on handling overflowing objects

The major challenge we encountered in implementing region skipping is that the calculated destination address of some objects may collide with the skipped dense regions during compaction, which will cause data corruption. As Figure 7.a shows, object 2 should have been moved to its destination *dest_addr* in region 0 whose remaining free space is not enough. If the copy really happens, object 2 will overwrite dense region 1 and corrupt object 1 (illustrated in Figure 7.b). We refer objects like object 2 as *overflowing objects*.

As illustrated in Algorithm 2, we address the issue of overflowing objects by first acquiring a shadow area in the summary phase (line 13), whose allocation strategy is similar to that for shadow regions. In the compacting phase, overflowing objects will be moved to the shadow area instead of their calculated destination address (line 19-20). Before the compaction ends, they will be copied to the heap end (line 38-41). In the example of Figure 7, object 2 will be copied to the shadow area (Figure 7.c) and finally moved to region 3 before the compacting phase finishes (Figure 7.d). Since PSGC does not allow uninitialized memory in the middle of the heap, the memory space at the calculated destination address should be filled with dummy objects.

Discussions. According to our evaluation, overflowing objects are common in applications that skip many regions. For example, Spark generates about 150 overflowing objects on average for every full GC cycle. Those overflowing objects result in considerable overhead. First, the usage of shadow regions introduces additional data copy operations. Since the summary phase has maintained the destination address for each overflowing object, we assign them to different GC threads to copy them in parallel. The copy overhead is about

4.5% of the GC time. Second, overflowing objects cause fragmentations since their original destination addresses cannot be reused and will be filled with dummy objects as shown in Figure 7.d. The resulted memory waste consumes averagely 0.15% of the entire heap size, which is trivial (further discussed in Section 5.6).

Another source of overhead comes from additional calculations in the summary phase and reference update since we have to identify dense regions and specially handle overflowing objects. However, the overhead is also trivial as we illustrate in Section 5.4.

5 Evaluation

We have implemented *ScissorGC* on OpenJDK 8u102-b14 with approximately 3000 lines of code. The evaluation is conducted on a machine with dual Intel ®Xeon™ E5-2618L v3 CPUs (2 sockets, 16 physical cores with SMT enabled) and 64G DRAM. As for benchmarks, we exploit all applications from the DaCapo [1][4], and some from SPECjvm2008 [23] and JOlden [5] suites, as well as Spark[5], a memory-intensive large-scale data processing engine. The maximum heap size for each application is set to 3X of its respective minimum heap size and listed in Table 1. For the DaCapo applications, we select the default workload and do not configure specific input sizes for SPECjvm2008 benchmarks. All results (except Figure 10) are the average of ten runs after a warm-up run. And we have specified the 95% confidential intervals of results in Figure 11, 11, 13, and 14.

Benchmark	Suite	Heap size (MB)
crypto.aes	SPECjvm2008	405
scimark.fft.large	SPECjvm2008	6750
serial	SPECjvm2008	4050
xml.validation	SPECjvm2008	1140
derby	SPECjvm2008	4950
avrora	DaCapo	10
batik	DaCapo	120
eclipse	DaCapo	285
fop	DaCapo	75
h2	DaCapo	720
jython	DaCapo	75
luindex	DaCapo	21
lusearch	DaCapo	21
pmd	DaCapo	150
sunflow	DaCapo	60
tradebeans	DaCapo	180
tradesoap	DaCapo	180
xalan	DaCapo	35
perimeter	JOlden	810
bisort	JOlden	720
mst	JOlden	2180
treeadd	JOlden	2700
tsp	JOlden	2550
voronoi	JOlden	900
health	JOlden	675
pagerank	Spark	43008

Table 1. Benchmark heap size

5.1 Full GC Throughput Improvement

The performance of full GC is measured by GC throughput, which is computed by the heap size before compaction divided by GC execution time. We mainly use vanilla OpenJDK 8 as our baseline. To understand the effects of the two optimization techniques respectively, we also provide the throughput result with only region skipping enabled.

As shown in Figure 8.a, full GC throughput is improved in all benchmarks but fop and pmd, which loses 0.09% and 0.04% throughput respectively. These two applications, as well as jython, do not invoke any full GC except the single System.gc() that DaCapo harness performs before iterations, so *ScissorGC* hardly works on them. For those improved cases, the throughput improvement ranges from 1.01X (crypto.aes) to 5.59X (perimeter).

Specifically, region skipping mainly works on scimark.fft.large, serial, Derby, and Spark because these applications create many dense regions at runtime while others do not. Region skipping even downgrades GC performance in jython, treeadd and voronoi, as the extra calculating and copying overhead offsets its benefit. To avoid such performance loss, we have implemented an adaptive policy to enable region skipping only when over one-third of destination regions are dense regions.

On the other hand, the shadow region improves the full GC throughput for most benchmarks except for luindex, Spark, crypto.aes, and xml.validation as these applications originally achieve satisfying thread utilization, which leaves little room for optimization. Since the dependencies among regions form a dependency graph, we have used *normalized critical path length*, which is calculated by the length of the longest dependency chain in the graph divided by the number of regions, to describe the difference in those benchmarks. In our test, the normalized length of the critical path is less than 0.05 for these four applications. As a comparison, the normalized length for Derby is 0.46. This metric suggests that they are hardly affected by chain-like dependencies compared with Derby.

We have also evaluated the thread utilization for Derby to compare that in Figure 4. As Figure 10 indicates, all threads spend most of the time working on destination (or shadow) regions, and the average thread utilization reaches 95.3%, which is 11.9X of that without shadow region optimization.

[4] We exclude the benchmark tomcat, since it has an underlying problem [13].

[5] We run a pagerank application in Spark 2.3.0 in the local mode, with a dataset consisting of a graph with 5 million edges sampled from the Friendster social network[22].

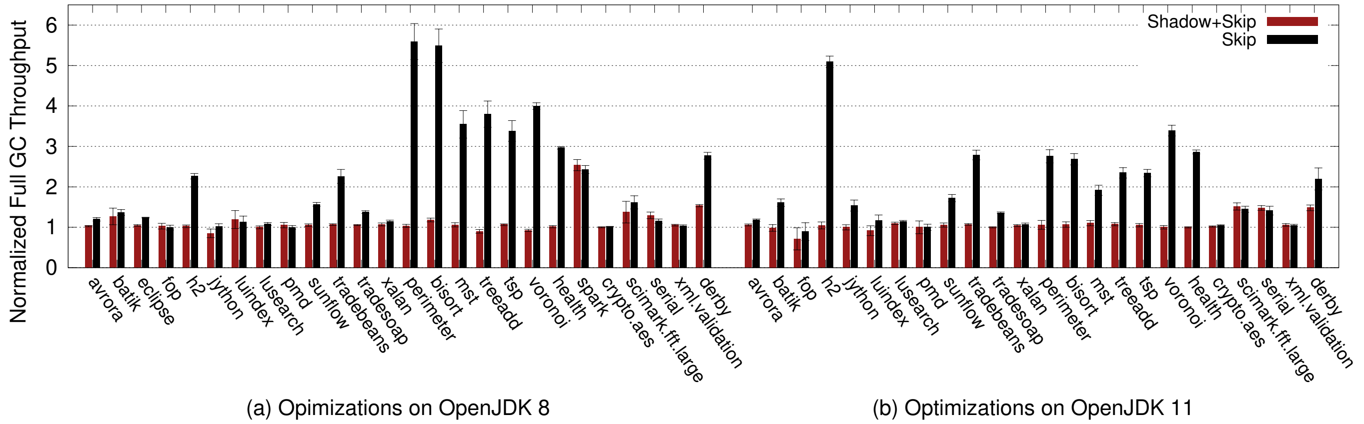


Figure 8. Full GC throughput improvement

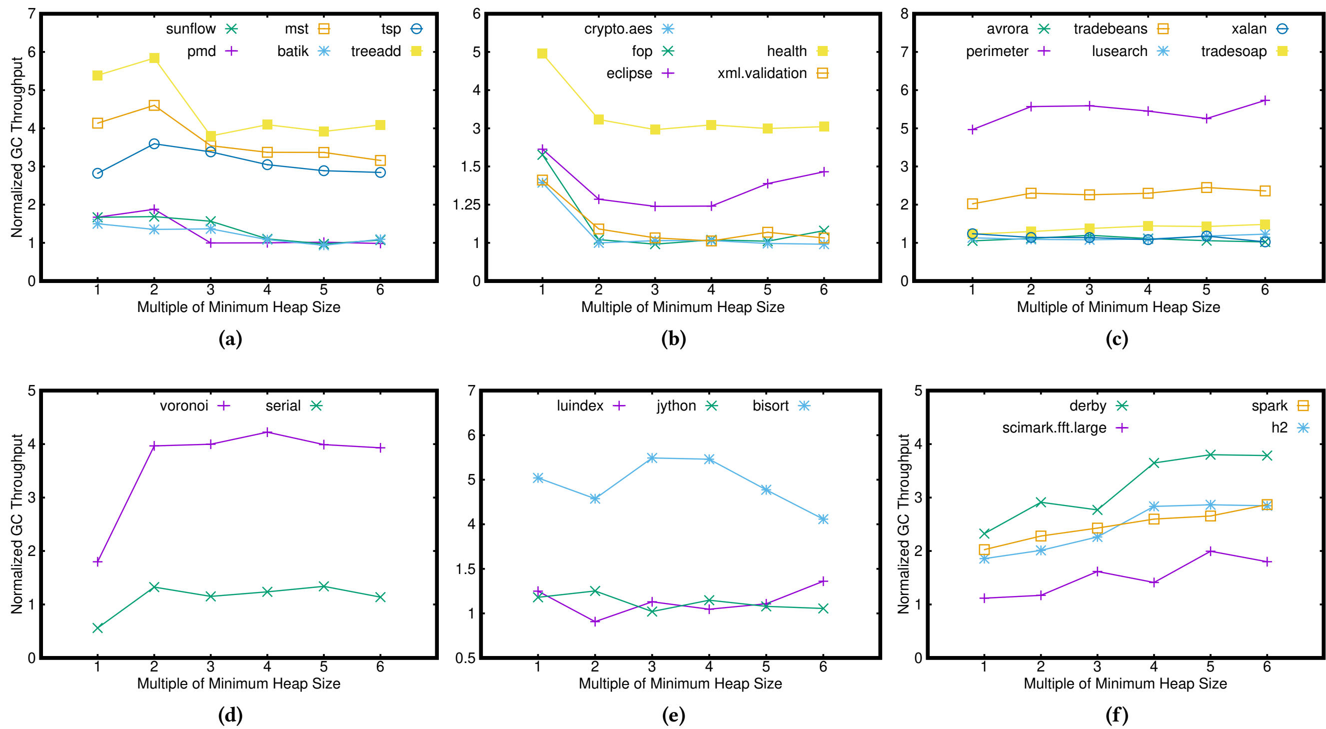


Figure 9. Full GC throughput under different multiples of minimum heap size

Figure 9 focuses on the impact of varying heap sizes on full GC throughput like previous work does [9, 19]. According to their shapes, we divide the curves into six groups as Figure 9.(a)-(f). In most cases, the throughput becomes relatively stable once the heap size exceeds a certain multiple of the minimum requirement, e.g., 2X or 3X. But for smaller heaps, GC throughput fluctuates. For example, curves in Figure 9.(a)/(b) reach their apices at 2X/1X of the minimum heap size since smaller heaps tend to introduce more frequent full GC, larger numbers of dense regions and severer region dependencies. But for some cases (like Figure 9.(d)/(f)), performance at the minimum heap size is worse, because *ScissorGC* will allocate excessive shadow regions and run out of empty regions in young space, which hurts the performance.

5.2 Application Performance Improvement

We have also evaluated the overall application performance by comparing execution time, as shown in Figure 11.a. To summarize, *ScissorGC* speeds 22 of 26 applications up from

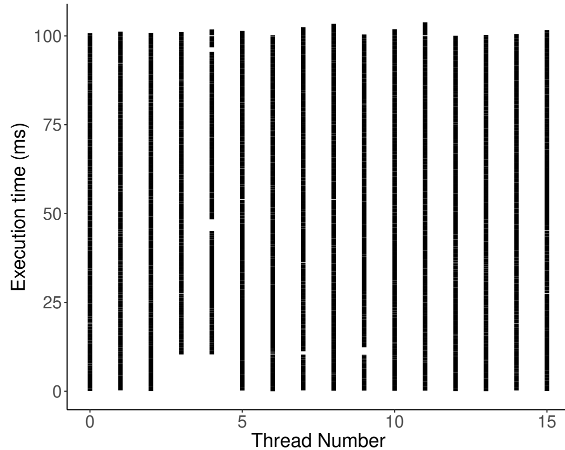


Figure 10. Optimized thread utilization for Derby

0.5% to 61.8%. Generally, applications with larger working sets benefit more from our optimizations on full GC. For example, Spark exploits the Java heap as a data cache for the datasets, which will occupy a significant portion of the memory space and thereby induce relatively frequent full GC phases. Consequently, Spark gains 20.2% speedup thanks to our optimizations. On the contrary, applications like Derby have relatively small working set and induce few full GC phases, so the improvement on application performance is limited.

5.3 Integrated with Other Optimizations

Prior work [8, 27] has studied the performance of full GC and provides various optimizations. However, our optimization techniques are orthogonal to those optimizations and can be easily integrated with them. To support our claim, we have ported our optimizations on the latest OpenJDK 11[6], whose full GC is improved with the proposal from Yu et al. [27], where the reference updating in the compacting phase is accelerated with a result cache. Since our optimizations have nothing to do with the reference updating logic, it is quite simple to port ours into OpenJDK 11. Figure 8.b[7] indicates that our *ScissorGC* can further improve the throughput of full GC in OpenJDK 11 for 23 benchmarks from 1.04X to 5.09X, while Figure 11.b shows the performance of 20 applications is improved from 0.2% to 49.0%.

5.4 Full GC Breakdown Analysis

Figure 12 shows the comparison of breakdown for the three phases in full GC between the vanilla and our *ScissorGC*. The left bar of every benchmark stands for vanilla while the right one stands for *ScissorGC*. As we mentioned in Section 2, the compacting phase dominates full GC, with 69.0% of the execution time on average. This number drops to 33.5% thanks to our optimizations. The compacting phase of 23 benchmarks speeds up by a range from 1.01X (xml.validation) to 9.2X (bisort). Although our region skipping optimization complicates the summary phase, the overhead is trivial considering its proportion to the overall GC time. Our evaluation indicates that the summary phase takes averagely 8.5% of all three phases with optimizations in *ScissorGC* (6.2% for vanilla), so the overhead in the summary phase does not affect the performance of full GC much. Meanwhile, as the execution time of the compacting phase dramatically decreases, the marking phase plays a much more important role and contributes to at most 63.3% (for voronoi) of the overall full GC pause, which makes it a potential optimization target in the future.

5.5 Scalability

The chain-like region dependencies are the culprit for limited GC thread utilization in compaction, resulting in poor GC performance and scalability. Since our optimizations have solved the thread utilization issue, we have also studied how *ScissorGC* improves the GC scalability by evaluating the GC throughput for all benchmarks with different numbers of GC threads (from 1 to 32). To eliminate performance fluctuation introduced by non-deterministic scheduling from the Linux scheduler, we have modified both JVMs to bind GC threads into a fixed core with *setaffinity*. If the number of GC threads is no more than 8, those threads will be bound to the same NUMA node, but on different physical cores. For 16 GC threads, they are bound to different physical cores respectively, but in two NUMA nodes. For 32 GC threads, they are pinned to different hardware threads. The result for scalability is shown in Figure 13[8][9].

ScissorGC has similar performance to the vanilla JVM at 1 thread as work stealing and shadow regions cannot work in the single-thread configuration. Furthermore, the overhead of overflowing objects cannot be mitigated by multi-thread parallelism and counteracts the benefit brought by region skipping. As the number of GC threads increases, *ScissorGC* notably improves the GC throughput compared with the vanilla JVM for all benchmarks except for avrora, batik, crypto.aes, and xml.validation. The later two cannot be optimized by shadow regions because the vanilla JVM already achieves relatively satisfying scalability due to much shorter critical paths (analyzed in Section 5.1). As for avrora, the heap only contains 2 4 regions, so the improvement brought by shadow regions is still marginal. Some applications which cannot benefit from the shadow region optimization (like

[6] The version of OpenJDK 11 is *OpenJDK11+28*

[7] Spark and eclipse cannot run with vanilla OpenJDK 11 in our environment, so we exclude it from the OpenJDK 11 evaluation.

[8] xml.validation lacks data at 1 thread because there is no full GC in that configuration.

[9] We select the representative benchmarks from each suite because some curves are in similar shapes. And the complete results can be found at our opensource repo.

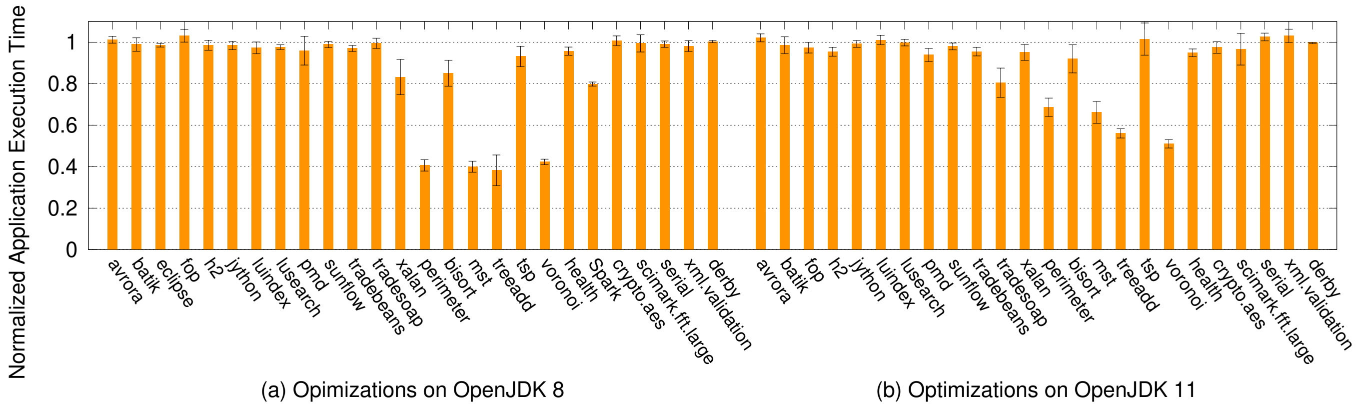


Figure 11. Application performance improvement

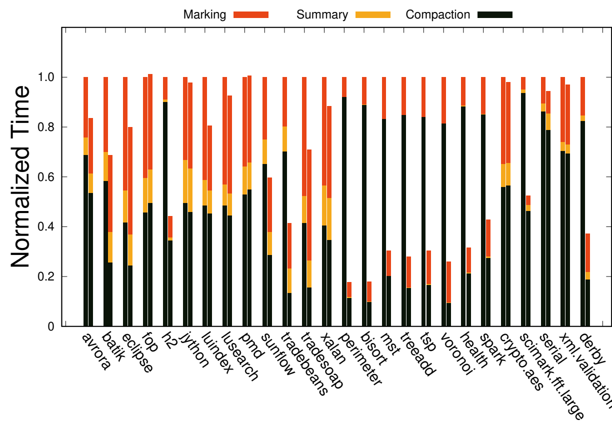


Figure 12. Full GC Breakdown Comparison

Spark and scimark.fft.large) still enjoy larger speedup with more GC threads, as the overhead of handling overflowing objects for region skipping is amortized by multiple threads.

Most benchmarks with *ScissorGC* can scale to 8 cores but some meet an inflection point in 16 or 32 threads. The 16-thread configuration introduces more cross-NUMA memory accesses, while the 32-thread one co-locates two GC threads on a single physical core to leverage SMT. Both NUMA and SMT have been proved harmful for garbage collections and applications in Java [8, 9, 21], which we leave as our future work for further analysis and optimizations.

To conclude, *ScissorGC* makes full GC much more scalable as it eliminates region dependencies and greatly improves GC thread utilization with shadow regions. It thereby improves the overall GC performance in the multi-core environment.

5.6 Memory Overhead

As handling overflowing objects in region skipping would leave fragmentations near dense regions, which results in memory waste, we have evaluated the *fragmentation ratio* for each benchmark by dividing the total size of fragments by the size of the whole heap. The result in Figure 14 only includes a part of benchmarks since the others result in few dense regions and the region skipping optimization is never triggered. In contrast, benchmarks with excessive dense regions, such as derby and Spark, have a relatively higher ratio. Nevertheless, even the highest fragmentation ratio (for serial) is only 1.22% while the average ratio is 0.07%. As a comparison, the dense prefix optimization in vanilla JVM would generate 1.3% fragmentation on average.

We also study the off-heap memory overhead led by shadow regions and compare it with a naive implementation which allocates new memory space each time a GC thread requires a shadow region. Since *ScissorGC* mainly leverages unused memory in the young space to allocate shadow regions, all benchmarks show weak demand for off-heap memory. In our evaluation, most benchmarks cause zero off-heap memory overhead, as the young space suffices for shadow regions. Only three benchmarks (batik, serial, and scimark.fft.large) require off heap memory to serve as shadow regions since they have a larger memory requirement to store overflowing objects. However, the off-heap memory overhead is only 0.07%, 1.03%, and 0.32% of the heap size respectively. As a comparison, the naive allocation policy (mentioned in Section 4.1) results in 97.79% extra memory consumption of the entire heap size. The overflowing objects usually introduce higher memory pressure than shadow regions in that the memory space they consume cannot be recycled until the compaction finishes. The objects in a shadow region can be copied back once the corresponding heap region becomes available. However, the overflowing objects can only be copied to the heap end when the whole compacting phase is nearly over. Therefore, the memory space consumed by the overflowing objects cannot be recycled and reused during compaction and thereby cause more memory overhead.

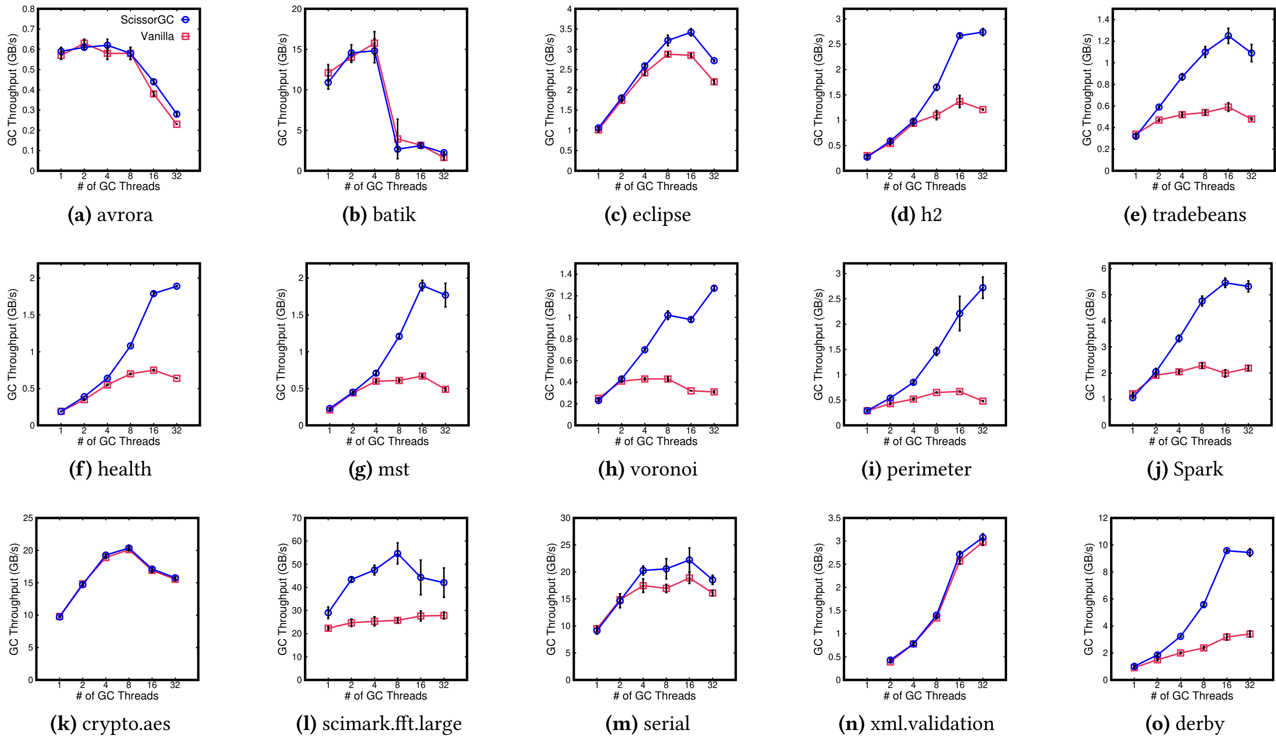


Figure 13. Full GC Scalability

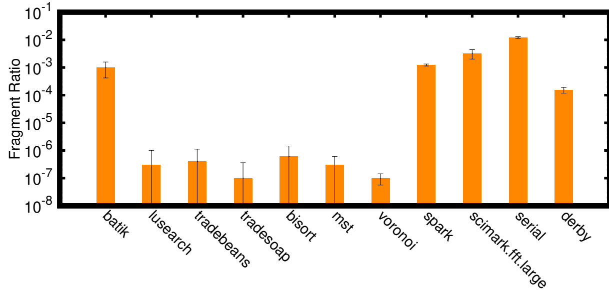


Figure 14. Fragmentation ratio with region skipping

The reduced memory consumption also improves the data locality in copying. Compared with the naive implementation which takes averagely 54.97% more time to fill up a shadow heap region than normal ones, *ScissorGC* only spends 7.02% more time thanks to better locality.

6 Related Work

6.1 GC Scalability

The scalability of a garbage collector is crucial for its performance on large-scale multi-core machines. Gidra et al. [8] find that PSGC is NUMA-unaware and exploit heavily-contended locks for synchronization and in turn provide corresponding optimizations. Suo et al. [24] point out that the unfair lock acquisitions in the JVM and imperfect scheduling decisions from the Linux CFS scheduler will hurt scalability, and propose optimizations such as annotating GC tasks with scheduling hints. Zhou et al. [30] present software configurable locality policies to improve the GC scalability for many-core processors. Our work mainly studies the scalability of the full GC algorithm, which is neglected by most prior work, and finds that the low thread utilization is the main culprit.

Multi-threading garbage collectors have introduced work stealing mechanism for better load balancing and scalability. However, work stealing also introduces considerable overhead and needs to be optimized. Qian et al. [20] find that GC threads will suffer from consecutive futile stealing attempts, so they force those threads to terminate to reduce the interferences among threads. Suo et al. [24] further maintain an active GC thread pool to eliminate meaningless steals to terminated GC threads. Wessam [12] exploits a master thread to adaptively decide the number of GC threads active for stealing based on the size of remaining tasks. Gidra et al. [9] propose *local mode* for NUMA machines to forbid GC threads to steal references from remote NUMA nodes so as to avoid costly cross-node memory access. Our work mainly focuses on enabling more available tasks to improve the success rate of stealing, and we also leverage lock-free mechanisms to reduce the stealing overhead.

6.2 GC on Memory-intensive Workload

Memory-intensive workload greedily requests memory from Java heap and induces much more frequent garbage collections than others. Therefore, optimizations on GC will have a significant impact on the overall performance. Since the allocation pattern of memory-intensive workload does not conform to the generational hypothesis, the region-based garbage collector [2, 3, 10, 17] is proposed to avoid unnecessary object copies in generational GC. Nguyen et al. [17] propose an epoch-based collector to collect objects only when an epoch ends. Gog et al. [10] define regions with different lifetime for allocation and fast collection. Bruno et al. [2, 3] divide the Java heap into regions with different *ages* and allocate objects to different regions according to their estimated lifespan and programmers' annotations. Espresso [26] instead considers efficient hybrid heap management atop a mix of non-volatile memory (NVM) and DRAM and tailors the GC algorithm for long-lived objects on NVM. Our work mainly studies the excessive dense regions in full GC introduced by memory-intensive workload. Li et al. [14] point out that the dense prefix in the stock JVM cannot optimize dense regions which are not in the head of a heap, while *ScissorGC* further analyzes the composition of dense regions and leverages shadow regions to improve the performance of region skipping.

Other optimizations focus on optimizing GC in memory-intensive workload, but in various scenarios. Yu et al. [27] focus on optimizing memory-intensive applications on memory-hungry platforms like Xeon Phi, where GC plays a more important role. Maas et al. [15, 16] study the GC behavior of memory-intensive applications for distributed scenarios, such as a Spark cluster, and propose several policies to coordinate GC among JVMs in different machines. Those optimizations are orthogonal to ours, and we have integrated our *ScissorGC* with the work from Yu et al. to gain larger speedup.

7 Conclusion

Full GC is a costly stage in Java garbage collectors which may induce prohibitive application pauses especially for large heaps. This paper mainly analyzes the most time-consuming compacting phase in full GC in the Parallel Scavenge garbage collector and uncovers two performance problems: non-scalability due to region-wise dependencies and inefficient compaction resulted with unnecessary data movement. To overcome those problems, this paper proposes *ScissorGC*, a new compaction algorithm with two corresponding optimizations: *shadow regions* to eliminate dependencies and enable more threads to run simultaneously, and *region skipping* to avoid moving regions filled up with live objects. *ScissorGC* has been implemented on two versions of HotSpot JVM of OpenJDK, and the evaluation shows that *ScissorGC* helps to improve both scalability and efficiency of compaction in full GC.

Acknowledgments

We sincerely thank our shepherd Jan Vitek and the anonymous reviewers for their constructive comments. This work is supported in part by China National Natural Science Foundation (No. 61672345) and National Key Research & Development Program of China (No. 2016YFB1000104). Haibo Chen is the corresponding author.

References

[1] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, October 22-26, 2006, USA*. ACM, 169–190.

[2] Rodrigo Bruno and Paulo Ferreira. 2017. POLM2: automatic profiling for object lifetime-aware memory management for hotspot big data applications. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. ACM, 147–160.

[3] Rodrigo Bruno, Luís Picciochi Oliveira, and Paulo Ferreira. 2017. NG2C: pretenuring garbage collection with dynamic generations for HotSpot big data applications. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*. ACM, 2–13.

[4] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst. 2010. HaLoop: efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 285–296.

[5] Brendon Cahoon and Kathryn S McKinley. 2001. Data flow analysis for software prefetching linked data structures in Java. In *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*. IEEE, 280–291.

[6] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management*. ACM, 37–48.

[7] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. 2012. Spinning fast iterative data flows. *Proceedings of the VLDB Endowment* 5, 11 (2012), 1268–1279.

[8] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. 2013. 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*. ACM, 229–240.

[9] Lokesh Gidra, Gaël Thomas, Julien Sopena, Marc Shapiro, and Nhan Nguyen. 2015. NumaGiC: a Garbage Collector for Big Data on Big NUMA Machines. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 661–673.

[10] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingan, Derek Murray, Steven Hand, and Michael Isard. 2015. Broom: sweeping out garbage collection from big data systems. In *Proceedings of the 15th USENIX conference on Hot Topics in Operating Systems*. USENIX Association, 2–2.

[11] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework.. In *OSDI*, Vol. 14. 599–613.

[12] Wessam Hassanein. 2016. Understanding and improving JVM GC work stealing at the data center scale. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*. ACM, 46–54.

[13] JDK Bug System. 2016. [JDK-8155588] org.apache.jasper.JasperException: Unable to compile class of JSP. https://bugs.openjdk.java.net/browse/JDK-8155588.

[14] Haoyu Li, Mingyu Wu, and Haibo Chen. 2018. Analysis and Optimizations of Java Full Garbage Collection. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*. ACM, 18.

[15] Martin Maas, Krste Asanović, Tim Harris, and John Kubiatowicz. 2016. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. *ACM SIGOPS Operating Systems Review* 50, 2 (2016), 457–471.

[16] Martin Maas, Tim Harris, Krste Asanovic, and John Kubiatowicz. 2015. Trash Day: Coordinating Garbage Collection in Distributed Systems.. In *HotOS*.

[17] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A high-performance big-data-friendly garbage collector. In *Proc. the 12th USENIX Conference on Operating Systems Design and Implementation*.

[18] OpenJDK. 2018. JEP 307: Parallel Full GC for G1. http://openjdk.java.net/jeps/307.

[19] Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. 2010. Schism: Fragmentation-tolerant Real-time Garbage Collection. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 146–159. https://doi.org/10.1145/1806596.1806615

[20] Junjie Qian, Witawas Srisa-an, Du Li, Hong Jiang, Sharad Seth, and Yaodong Yang. 2015. SmartStealing: Analysis and Optimization of Work Stealing in Parallel Garbage Collection for Java VM. In *Proceedings of the Principles and Practices of Programming on The Java Platform*. ACM, 170–181.

[21] Yaoping Ruan, Vivek S Pai, Erich Nahum, and John M Tracey. 2005. Evaluating the impact of simultaneous multithreading on network servers using real hardware. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 33. ACM, 315–326.

[22] SNAP. 2014. Friendster. http://snap.stanford.edu/data/com-Friendster.html.

[23] SPEC. 2008. SPECjvm2008. https://www.spec.org/jvm2008/.

[24] Kun Suo, Jia Rao, Hong Jiang, and Witawas Srisa-an. 2018. Characterizing and optimizing hotspot parallel garbage collection on multicore systems. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 35:1–35:15.

[25] David Ungar. 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *ACM Sigplan notices*, Vol. 19. ACM, 157–167.

[26] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. 2018. Espresso: Brewing Java For More Non-Volatility with Non-volatile Memory. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 70–83.

[27] Yang Yu, Tianyang Lei, Weihua Zhang, Haibo Chen, and Binyu Zang. 2016. Performance Analysis and Optimization of Full Garbage Collection in Memory-hungry Environments. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, 123–130.

[28] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*.

[29] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*.

[30] Jin Zhou and Brian Demsky. 2012. Memory management for many-core processors with software configurable locality policies. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 3–14.

[31] Paul C Zikopolous, George Baklarz, and Dan Scott. 2005. *Apache derby/IBM cloudscape*. Prentice Hall PTR.