Analysis and Optimizations of Java Full Garbage Collection

Haoyu Li, Mingyu Wu, Haibo Chen
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
Contact:haibochen@sjtu.edu.cn

ABSTRACT

Java runtime frees applications from manual memory management by its automatic garbage collection (GC), 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 the whole heap and induce a pause tens of times longer than normal collections, which largely affects both throughput and latency of the entire system. In this paper, we analyze the full GC performance of HotSpot Parallel Scavenge garbage collector comprehensively and study its algorithm design in depth. We find out that heavy dependencies among heap regions cause poor thread utilization. Furthermore, many heap regions contain mostly live objects (referred to as *dense regions*), which are unnecessary to collect. To solve these problems, we introduce two kinds of optimizations: allocating shadow regions dynamically as compaction destination to eliminate region dependencies and skipping dense regions to reduce GC workload. Evaluation results show the optimizations lead to averagely 2.6X (up to 4.5X) improvement in full GC throughput and thereby boost the application performance by 18.2% on average (58.4% at best).

KEYWORDS

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

ACM Reference Format:
Haoyu Li, Mingyu Wu, Haibo Chen. 2018. Analysis and Optimizations of Java Full Garbage Collection. In *APSys '18: 9th Asia-Pacific Workshop on Systems (APSys '18), August 27–28, 2018, Jeju Island, Republic of Korea.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3265723.3265735

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 [17] so that most collections only touch a small portion of the heap and finish quickly, they inevitably have to enter a phase called *full GC* to collect the whole heap space and thus incur considerable pause time. The problem is even aggravated in today's prevalent memory-intensive frameworks like Spark [19] since full GC happens more frequently and induces longer pauses.

In this paper, we provide a comprehensive analysis of the full GC part of Parallel Scavenge Garbage Collector (PSGC), the default GC in the HotSpot JVM. The full GC 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: poor thread utilization due to dependencies among regions and unsatisfying compaction arising from excessive data movement. To this end, we propose two optimization techniques. For the thread utilization problem, we introduce *shadow regions* to eliminate dependencies and in turn enable more threads to run in parallel. For the compaction efficiency problem, we provide a *region skipping* design to avoid moving regions in which most objects are alive.

We have implemented those two optimizations in the HotSpot JVM of OpenJDK 8u and compared it with the vanilla PSGC over applications from various benchmark suites. The result confirms that our optimizations improve the full GC throughput for applications by 2.6X on average (up to 4.5X) and thereby shorten the execution time of applications by averagely 18.3% (at most 58.4%).

2 BACKGROUND

2.1 Parallel Scavenge

Parallel Scavenge (PS) is the default garbage collector in the HotSpot JVM of OpenJDK 8. It collects objects in a stop-the-world fashion: when a GC phase 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.

PS 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 memory resource of the whole heap is exhausted and thus collects both spaces. To reach a satisfying application latency, young space is usually designed as a small fraction of the heap region 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 frequently in memory-intensive applications. Our evaluation on Spark with a 20GB heap shows that full GC phases 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. 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 compaction-based algorithm that copies all live objects into the beginning of the old space to vacate a vast continuous free memory space. PS implements full GC as a three-phase algorithm, including mark phase, summary phase, and compacting phase. We will briefly explain these three phases below.

Mark phase. In the first mark 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 locations are recorded in bitmaps for later use.

Summary phase. After the mark phase, the PS collector will calculate a heap summary for all live objects based on the pre-generated bitmaps. The summary phase is region-based: PS partitions the JVM heap into continuous *regions* of equal size (512KB by default in PSGC) and summarizes objects within the same region together. After the summary phase, a mapping between regions is generated so that each *source region* will have its own *destination regions*[1] for object copying.

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

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 80% of the overall full GC time, we will focus on optimizing this phase in this work. 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 corresponding source regions. Reference updates for live objects also occur during copying 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, and once a GC thread fills up a destination region, it will decrement *dcount* for all corresponding source regions. When a region's *dcount* reaches zero, all live objects within it have been evacuated so it can be reused 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 between regions can be used to construct a dependency graph to shape the execution behavior of GC threads.

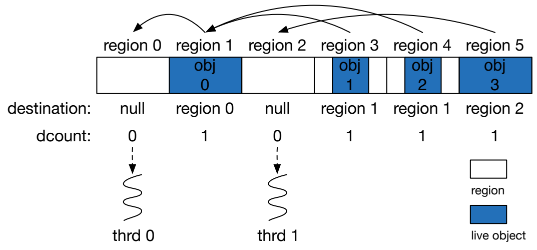

Figure 1: An example of how full GC workload goes imbalanced

Optimizations. As mentioned in Section 2.2, not all destination regions are available at the beginning of compaction, so PS can only assign those available regions to GC threads initially. This design, however, is likely to result in a load imbalance. Take Figure 1 as an example, thread 0 and thread 1 both receive one available region (region 0 and 2) initially. However, according to the region dependencies, region 1, 3 and 4 will eventually be pushed into thread 0's working stack while only region 5 will be assigned to thread 1. As a result, PS assigns four tasks to thread 0 but two tasks to thread 1, which leads to a load imbalance. To this end, full GC enables work stealing so that GC threads can steal available regions from other threads after their own working stacks are drained.

Full GC also specially handles regions at the beginning of the heap. If most objects in those regions are alive (named *dense regions* in PS), 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 [20] in the SPECjvm2008 benchmark suite [14] as an example to support our claim. Figure 2 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 resources on stealing in vain (blanks). The terrible utilization rate strongly motivates us to optimize it.

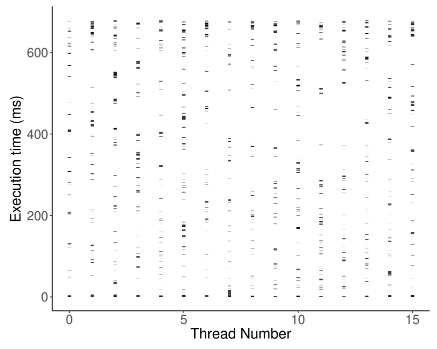

Figure 2: Thread utilization in a full GC phase of Derby

We have further profiled work stealing by dumping the working stack for every thread and find that all except one stack are empty 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 3. 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.

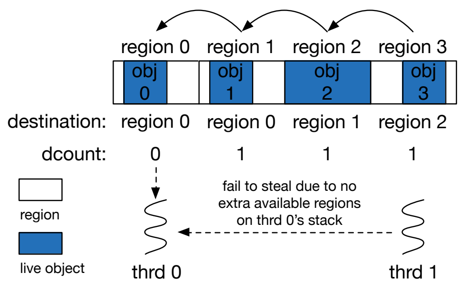

Figure 3: An example of dependency chains, the culprit for poor thread utilization

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. A key observation is that applications on Spark are usually based on huge datasets, which generate many large arrays that can easily fill up a single heap region. If those arrays remain alive during a full GC phase, regions containing them will become *dense regions* and thereby do not require compaction. The dense regions are dispersed throughout the whole heap, so many of them cannot benefit from the dense prefix optimization in PSGC and thus suffer from unnecessary compaction. Our evaluation results show the average proportions of destination regions with 100%, 95%, and 90% density but out of the dense prefix are 31.2%, 37.0%, and 37.2% respectively in Spark (running the page rank application with a 20GB heap). The results suggest great optimization potential for compaction efficiency even with a dense prefix.

4 OPTIMIZATIONS

As analyzed in Section 3, there are two major performance issues in full GC: limited thread utilization due to chained region dependencies and inefficient compaction for dense regions. To resolve these problems, we introduce shadow regions and region skipping in this section.

4.1 Shadow Region

4.1.1 Basic Idea. The goal of the shadow region optimization is to allow threads to steal unavailable regions. Specifically, when a GC thread encounters stealing failure, it will turn to unavailable regions rather than spin or sleep. Since an unavailable region still contains live objects and cannot

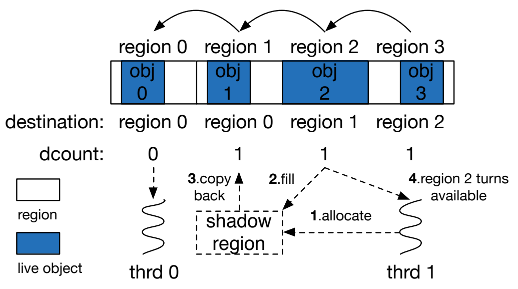

Figure 4: An example on how shadow regions resolve dependency chains

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.

Figure 4 shows an example of how shadow regions help optimize dependency chains. Suppose initially thread 0 gets region 0 while thread 1 has an empty stack. When both threads start working, thread 1 will try to steal from thread 0 but fail since the only available region is being processed by thread 0. With our optimization, thread 1 now can turn to unavailable regions instead of remaining idle. If thread 1 chooses region 1, it will allocate a shadow region for it (the dashed rectangle in Figure 4) and start to copy live objects from region 1's source, i.e., region 2. Once it has filled the shadow region, region 2 becomes available and can be exploited as a destination for other ones (region 3 in this example). Data within the shadow region will be copied back to region 1 as soon as it becomes available. This design improves the success rate of work stealing and thereby achieves better thread utilization.

4.1.2 Implementation. We implement shadow regions as a complement to work stealing. GC threads allocate a shadow region each time when they encounter stealing failure, and only become idle if they cannot find unavailable regions anymore (i.e. nearly all destination regions throughout the heap have been processed). In our implementation, a GC thread linearly scans the heap from its last processed region, picks the first unavailable one, and creates a shadow region for further processing. Since multiple GC threads may simultaneously search the heap for region processing, we introduce a slot for each region, and GC threads leverage atomic Compare-And-Swap (CAS) instructions to write their identifiers into the slot 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.

The shadow region optimization greatly improves the thread utilization but at the sacrifice of additional memory overhead and data locality. A naive implementation is to allocate a new shadow region each time a GC thread attempts to steal an unavailable region. 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. To this end, we choose to reuse shadow regions with a LIFO region stack to reduce memory consumption and improve locality. In our evaluation, this design uses on average 4.96% extra memory and takes 7.02% more time to fill up shadow regions than normal ones. As a comparison, the naive allocation policy results in 97.79% extra memory consumption and 54.97% more heap filling time for shadow regions on average. Actually, the memory overhead can be further mitigated by reusing idle heap regions in young space, which we leave to future work.

The shadow region optimization also introduces additional data copying from shadow regions back to heap regions. Nevertheless, since the objects in a shadow region are already compacted and have references updated, the extra copying operation could be done by a single memory copy instruction, which only takes up 3.5% of the region task execution time on average.

It is possible to apply more sophisticated heuristics to maximize the benefit of shadow regions. For example, GC threads can priorly 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

4.2.1 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.

4.2.2 Implementation. For simplicity, we only skip over regions where objects are all alive in our implementation. Furthermore, regions at the end of the heap must be moved forward regardless of their density to avoid very large fragmentation.

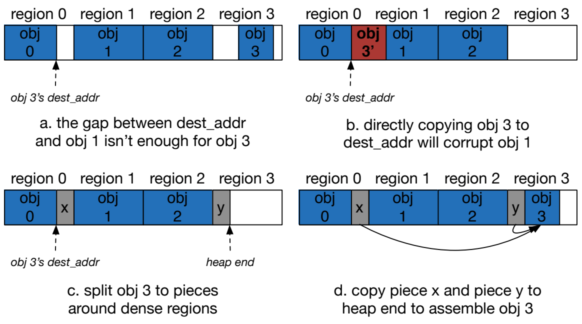
region 0
region 1
region 2
region 3
obj 0
obj 1
obj 2
obj 3
obj 3'
x
y
obj 3's dest_addr
heap end
a. the gap between dest_addr and obj 1 isn't enough for obj 3
b. directly copying obj 3 to dest_addr will corrupt obj 1
c. split obj 3 to pieces around dense regions
d. copy piece x and piece y to heap end to assemble obj 3

Figure 5: An example on handling overflowing objects

The major challenge we encountered is that some moved objects may overlap with dense regions during compaction and cause data corruption. As Figure 5.a shows, object 3 is moving to its destination *dest_addr* in region 0 whose remaining free space is not enough. If the copy really happens, object 3 will overwrite dense region 1 and corrupt object 1 (illustrated in Figure 5.b). We refer objects like object 3 as *overflowing objects*. To avoid such dangerous situations, we split overflowing objects into pieces around dense regions during compaction (Figure 5.c), and copy all pieces to the end of the heap to assemble overflowing objects after compaction (Figure 5.d). Since PSGC doesn't allow free space between live objects, we need to fill dummy objects into these pieces after processing overflowing objects. This solution is feasible and straightforward, but it requires extra copying for overflowing objects. In 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 phase. To mitigate the overhead of handling overflowing objects, we assign them to different GC threads to process them in parallel. As a result, the overhead gets reduced from 17.1% to 6.7% of the GC time.

5 EVALUATION

We have implemented our optimizations on OpenJDK 8u102-b14 with approximately 3000 lines of code. The evaluation is conducted on a machine with dual Intel ®Xeon^{TM} E5-2618L v3 CPUs (16 cores) with 64G DRAM. As a benchmark, we exploit various applications from the DaCapo [1], SPECjvm2008 [14] and JOlden [4] suites, as well as Spark[2], a memory-intensive large-scale data processing engine. Some applications in those benchmarks have a limited memory budget and never trigger a full GC, so we have excluded them. The maximum heap size for each chosen application is listed in Table 1. All results (except Figure 7) are the average of five runs.

[2]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[13].

Benchmark	Suite	Heap size (GB)
crypto.aes	SPECjvm2008	1
scimark.fft.large	SPECjvm2008	4
serial	SPECjvm2008	2
xml.validation	SPECjvm2008	0.5
derby	SPECjvm2008	2
tradebeans	DaCapo	0.5
tradesoap	DaCapo	0.5
perimeter	JOlden	1
bisort	JOlden	1
mst	JOlden	1
treeadd	JOlden	1
tsp	JOlden	1
voronoi	JOlden	1
health	JOlden	1
pagerank	Spark	20

Table 1: Benchmark heap size

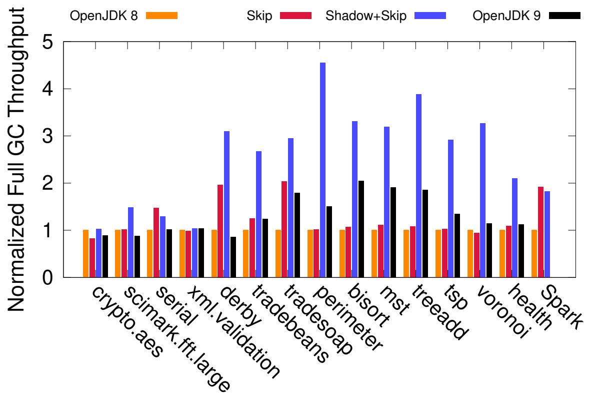

Figure 6: Full GC throughput improvement

5.1 Full GC Throughput Improvement

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

As shown in Figure 6, full GC throughput is improved in all benchmarks but crypto.aes and xml.validation. In those improved cases, the throughput improvement ranges from 1.5X (serial) to 4.5X (perimeter), and the average of all benchmarks is 2.6X. Specifically, region skipping mainly works on serial, Derby, trade, and Spark because these applications create many dense regions at runtime while others do not.

Region skipping even slightly downgrades GC performance in crypto.aes, xml.validation, 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 crypto.aes, serial, xml.validation, and Spark, 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 between Spark and other 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 chained dependencies compared with Derby.

We also provide the evaluation results of vanilla OpenJDK 9, into which Yu's work[18] has been merged. OpenJDK 9 is a newer version than OpenJDK 8, but it is not a long-term-support (LTS) version and not so welcomed as OpenJDK 8.[3] As illustrated in Figure 6, we achieve on average 2.0X higher full GC throughput than vanilla OpenJDK 9.

Lastly, we have also evaluated the thread utilization for Derby to compare that in Figure 2. As Figure 7 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. Furthermore, the thread utilization in OpenJDK 9 is still unsatisfying (about 7.7% for Derby), so our optimizations are still workable for it.

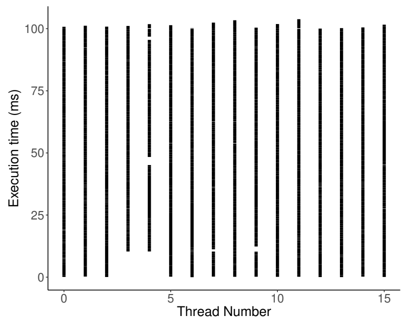

Figure 7: Optimized thread utilization for Derby

[3]In our evaluation, Spark cannot run with OpenJDK 9, so the corresponding result is missing in Figure 6.

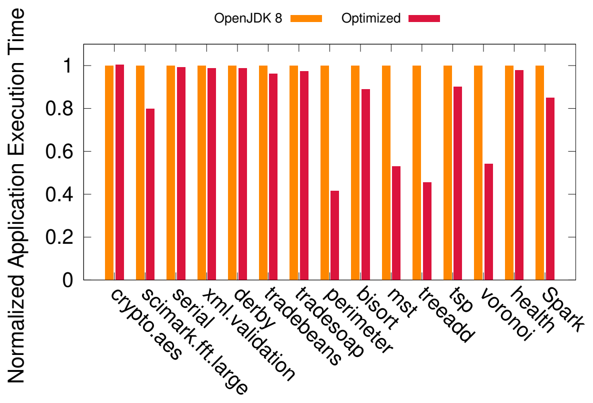

Figure 8: Application performance improvement

5.2 Application Performance Improvement

We have also evaluated the overall application performance by comparing execution time, as shown in Figure 8. To summarize, we achieve application performance improvement for 58.4% at best and 18.2% on average. 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 15% 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 trivial.

6 RELATED WORK

GC analysis and optimizations. Garbage collection is essential to Java runtime and has been analyzed and optimized for years. Gidra et al. [6, 7] provide a comprehensive analysis on PSGC and uncover several problems like NUMA-unawareness and heavily contended locks. Suo et al. [15] find that the lack of coordination between the Linux scheduler and the Java thread manager may result in scalability problems. Nguyen et al. [9] find a large amount of unnecessary object copying in big-data workload during GC and eliminate it with a new collector. Bruno et al. [2, 3] have a similar observation but use pre-tenuring to solve the problem instead. Yu et al. [18] point out an efficiency problem in the destination calculation during the compacting phase in full GC and design a calculation cache to optimize it out. Those solutions are orthogonal to our work, and some of them can be integrated with ours for larger performance improvement.

Other collectors. Recently there is growing interest in allowing GC threads and mutators to run simultaneously for better application latency [5, 11, 16]. Although those concurrent collectors reduce or even eliminate STW pauses, they may still need a final full GC phase to avoid the case where the speed of allocation is much faster than that of the collection. For example, G1GC [5], a new generation of concurrent collector in HotSpot, contains full GC and recently has been optimized to support multi-threading [10] to improve performance. Therefore, our optimizations on full GC are useful and can be adapted to other collectors. Moreover, the idea of generational collections and full GC is also adopted in other language runtimes like CLR [12] and V8 [8], so our optimizations may also be useful for them.

7 CONCLUSION

Full GC is a costly phase in Java garbage collectors which may induce prohibitive application pauses especially for large heaps. This paper analyzes full GC in the default HotSpot collector, spots major sources of inefficiency, and proposes optimizations with satisfying performance improvement.

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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Google. 2018. Chrome V8. https://developers.google.com/v8/.

[9] 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*.

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

[11] Erik Österlund and Welf Löwe. 2016. Block-free concurrent GC: stack scanning and copying. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*. ACM, 1–12.

[12] Jeffrey Richter. 2006. *CLR via c#*. Vol. 4. Microsoft Press Redmond.

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

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

[15] 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.

[16] Gil Tene, Balaji Iyengar, and Michael Wolf. 2011. C4: The continuously concurrent compacting collector. In *Proceedings of the 10th international symposium on Memory management*. ACM, 79–88.

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

[18] 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.

[19] 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*.

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