SAGP: A Design of Swap Aware JVM GC Policy

Qichen Chen
Seoul National University
Seoul, Korea
charliecqc@dcslab.snu.ac.kr

ABSTRACT

JVM based Applications usually suffers a severe suspend from the "Stop-The-World" pauses during Garbage Collection(GC). The suspend time could even be much longer when some of the GC targets are systematically swapped out to a permanent storage device. Additionally, we discovered that the default Parallel-Compact GC policy shifts contents that are not GC targets to reduce segmentations, which make the situation worse if these contents have been already swapped out to slow devices such as HDD. We present a new garbage collect policy that can recognize the swapped out contents and will remove them from the shifting targets. From our demonstrative experiment, the new policy can theoretically reduce the latency of collecting 10 GiB contents from 671s to 190s.

KEYWORDS

Java, JVM, Garbage Collection, Swap System

ACM Reference Format:
Qichen Chen. 2018. SAGP: A Design of Swap Aware JVM GC Policy. In *Proceedings of ACM Middleware conference (Middleware'18).* ACM, New York, NY, USA, Article 4, 3 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

Data-intensive workload processing platforms such as Spark, MapReduce are widely used as the big data era comes. In these platforms, Data is distributed to each node, processed there and aggregated to the client. Considering to the efficiency, each node should provide an acceptable performance while processing as much data as possible.

Given the popularity and powerfulness of Java, most of these platforms are constructed based on JVM, which manages a heap space to store data objects. GC (Garbage Collection) [1] frequently occurs on the heap space to recycle the unused objects, while JVM could be stopped during the GC, which introduce a "Stop The World" pause to the client. The STW problem will become much more serious when part of the GC targets is swapped out to the hard disk, since during the GC, those targets should be swap in to memory again and the slow disk I/O latency will be the bottleneck. a STW pause might be fatal to the distributed data processing platform, thus Some tunning guides of JVM based distributed systems[5] suggest to avoid system swap due to its high cost. In addition, the default full GC policy of OpenJDK 8: parallel compact collector[?], will slide the Non-GC target to eliminate internal heap segmentation thus constantly moving data from disk to memory. Suspending system swap might be a good solution to solve the performance issue, however, preventing swap also means limit the data size that can be processed in one node into the node's DRAM capacity, which causes introducing more processing nodes and increase the total cost.

To handle this issue, previous studies [8] proposed a way to use huge address space for JVM that choose other node's empty memory as its swap space. Matthew Hertz et al [7] proposed the solution that bookmarking the evicted page and keep them untouched during gc procedure, this function is supported by modifying the Linux kernel, However kernel modification is usually difficult and easy to affect other applications. besides, Zhenyun Zhuang et[9] al proposed an idea that using THP to reduce system memory allocation pressure for removing the GC overhead.

In our research, we're going to present a swap aware JVM GC policy with two main schemes to improve the GC performance as follows:(1) we use a reference count to track every object access, then during the summary phase of parallel compact procedure, we remove the regions that with small access count from shifting target, in case they are out of the memory with high probability.(2) We read linux pagemap[3] info to identify whether each page that belongs to accessed object is swapped out or not, if a page is swapped out, then we take it into consideration during the summary phase.

2 DESIGN AND IMPLEMENTATION

Our newly designed GC policy is based on the parallel compacting policy. There are three phases in the policy.

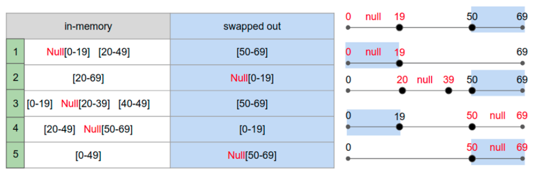
First phase is the marking phase, each generation is directly reachable from the application code is divided among garbage collection as live, the data for the region it is in is updated with information about the size and location of the object.

Second, the summary phase operates on regions. From the previous collections, it is probably that there are some portions in the left side contains most of the live object and the left dead objects in those portions are not worth to compact them. As a result, it firstly computes the density of each generation, starting with the most left region, moving to a certain point where recovering the regions right to the point are worth of the compacting cost. In this case, the point is called as the dense prefix, no object that in the left part of the dense prefix will be compact. Then the summary phase recalculates new location of each region that need to be shifted

Lastly, in the compaction phase, the garbage collection threads use the result of summary phase to identify regions need to be filled, and copy data into the region independently.

From the details of the parallel compact policy, it is clear that besides the liveness of object, the system level page swapness has not been taken into consideration by JVM level GC policy. To see

Middleware'18, December 2018, Rennes, France
© 2016 Copyright held by the owner/author(s).
ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4



	in-memory	swapped out
1	Null[0-19] [20-49]	[50-69]
2	[20-69]	Null[0-19]
3	[0-19] Null[20-39] [40-49]	[50-69]
4	[20-49] Null[50-69]	[0-19]
5	[0-49]	Null[50-69]

Figure 1: evaluation on swap impaction

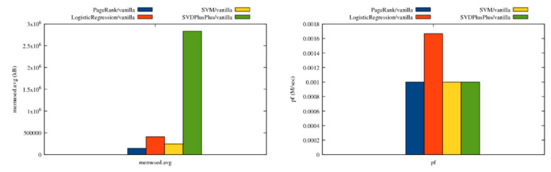
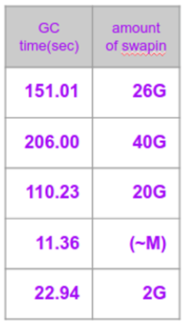


Figure 3: SVD++ Page fault count and memory access count



GC time(sec)	amount of swapin
151.01	26G
206.00	40G
110.23	20G
11.36	(~M)
22.94	2G

Figure 2: Evaluation result

the impact on the GC performance when portion of the heap is out of memory, we designed the experiment as follows:

2.1 Evaluation Design for verifying swap impaction

We performed all measurements on a 2.1 GHz Intel Xeon E5 linux machine with 64 GB of RAM and 60 GB of local swap space. To make sure swap happens, we fix the JVM max heap size to 115 GB, which is much larger than the RAM size. To observe the influence to GC performance by the swapped objects, we designed a simple program. in this program, we firstly allocate 70 objects, each of them is 1 GB, sequentially, then we keep accessing 10 of them in order to bring the swapped object back to the memory, afterwards we mark 20 of the allocated objects as unreachable by assigning NULL to their reference. We doing so is for creating recycle targets for the upcoming GC. At last we continue to allocate new object, as the old generation is full now, a full GC will be triggered and we measure the GC latency as our evaluation metrics.

Figure 2 shows us the 5 scenarios we have designed, the number there represents the object index, while the "NULL" shows the corresponding object is going to become unreachable. Each of the scenarios depicts the object distribution priors to the full GC starts.

Figure 3 shows the GC time and swapped in size during GC of each case. From Figure 3 we can tell that the case 1-3 takes much longer time due to its recycling target is swapped out or for compacting the remaining part, extra data swap in occurs and consumes much time. On the other hand, neither the compacting nor the recycling target is swapped out, which make these 2 cases the fastest ones of all scenarios.

2.2 Swap Aware GC Policy

From the last section, we can draw a conclusion that the existed parallel compact policy is not friendly to the scenario where part of the recycle target or the compacting target is out of the memory. To solve the problem, we are going to propose 2 schemes, which can avoid swapped out pages being involved into region compaction.

The first solution we are going to take is using a reference count to identify the hotness of each object. Since the Linux Swap system using LRU list to manage pages, objects with large access count will be resident in RAM with high possibility. In opposite, recently less accessed objects are most likely swapped out to disk. Currently, we have already analyzed how to get the reference variable to certain heap object from the local variable table. The future work is adding reference count through the reference variable to finally identify objects' hotness.

The second solution is using linux pagemap to identify each page's swappness during summary phase. As we mentioned above, at the beginning of summary phases, scanning on all regions is executed. We implant the pagemap read for region's virtual address into the region scan process, then we can get the number of the pages that are swapped out in each region. Finally, we employ the information when doing dense prefix computation, and get the most appropriate result considering the amount and location of swapped objects.

2.3 Evaluation Plan

Our Plan of evaluation is as follows: We will use workloads that meet the following requirements to validate the performance of the two solutions described in the previous section. (1)High memory intensity (2)High locality

One of the examples is pseudo-JBB[4], which is an SPEC benchmark for Java, and another candidate is JCheck[2], which runs hashtable operations. In addition, the graph computation and machine learning algorithms of Sparkbench[6] could be selected as out target workload. We are going to use these benchmarks to evaluate our proposed design with those running on original OpenJDK 8 with the parallel compaction GC policy.

So far, we have already completed a baseline experiment on some of the workloads in Sparkbench. Evaluation metrics include memory usage and page fault,. From Figure 3, We could know that SVDPlusPlus has an access locality because it has fewer page faults comparing to its long execution time and high memory usage. Therefore, we plan to evaluate the solutions with SVDPlusPlus later on. Also, we have completed the measurement of swap overhead on SVDPlusPlus. The final goal is to optimize GC policy to converge on performance when there is no swapping.

3 CONCLUSIONS

In this paper, we clarify the problem that current JVM GC policies are not aware of linux system level swap. We analyzed the problem through simple validate program and found that in the parallel compact policy, swapped out objects may be involved to become the recycling targets or compacting targets. we proposed 2 solutions aiming to solve this problem, we also proposed a detailed evaluation plan to validate our solutions in the future.

4 ACKNOWLEDGMENT

I am grateful to HyoJeong Lee as she made a huge contribution to this job. I am also thanks to YongSeok Son for his contribution. This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No.2018R1C1B5085640) and (No.2015M3C4A7065646)

REFERENCES

[1] [n. d.]. Java Grabage Collection basics. http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html
[2] [n. d.]. *JCheck*. http://openjdk.java.net/projects/code-tools/jcheck/.
[3] [n. d.]. pagemap, from the userspace perspective. https://www.kernel.org/doc/Documentation/vm/pagemap.txt
[4] [n. d.]. *pseudJBB*. http://www.spec.org/jbb2005/.
[5] Srigurunath Chakravarthi. 2010. Tuning Hadoop for Performance. hadoopsummit2010
[6] Min et al Li. 2017. *SparkBench: a spark benchmarking suite characterizing large-scale in-memory data analytics*. Cluster Computing 20.3 (2017): 2575-2589.
[7] Emery D.Berger Matthew Herz, Yi Feng. 2005. Garbage Collection Without Paging. PLDI\T1\textquoteright05,12\T1\textendash15June2005,Chicago,Illinois,USA
[8] Ronald Veldema. 2007. Supporting Huge Address Spaces in a Virtual Machine for Java on a Cluster. LanguagesandCompilersforParallelComputing. SpringerBerlinHeidelberg,2007:187-201.
[9] Haricharan Ramachandra Badri Sridharan Zhenyun Zhuang, Guong Tran. 2016. OS-Caused Large JVM Pauses: Investigations and Solutions. JournalofSoftwareVolume11,Number10,October2016