JVM Configuration Management and Its Performance Impact for Big Data Applications

Semih Sahin*, Wenqi Cao†, Qi Zhang‡, and Ling Liu§
Georgia Institute of Technology
*ssahin7@gatech.edu, †wcao39@gatech.edu, ‡qzhang90@gatech.edu, §ling.liu@cc.gatech.edu

***Abstract*—Big data applications are typically programmed using garbage collected languages, such as Java, in order to take advantage of garbage collected memory management, instead of explicit and manual management of application memory, e.g., dangling pointers, memory leaks, dead objects. However, application performance in Java like garbage collected languages is known to be highly correlated with the heap size and performance of language runtime such as Java Virtual Machine (JVM). Although different heap resizing techniques and garbage collection algorithms are proposed, most of existing solutions require modification to JVM, guest OS kernel, host OS kernel or hypervisor. In this paper, we evaluate and analyze the effects of tuning JVM heap structure and garbage collection parameters on application performance, without requiring any modification to JVM, guest OS, host OS and hypervisor. Our extensive measurement study shows a number of interesting observations: (i) Increasing heap size may not increase application performance for all cases and at all times; (ii) Heap space error may not necessarily indicate that heap is full; (iii) Heap space errors can be resolved by tuning heap structure parameters without enlarging heap; and (iv) JVM of small heap sizes may achieve the same application performance by tuning JVM heap structure and GC parameters without any modification to JVM, VM and OS kernel. We conjecture that these results can help software developers of big data applications to achieve high performance big data computing by better management and configuration of their JVM runtime.**

1. Introduction

Big data computing platforms today are represented by Hadoop HDFS/MapReduce [1], Spark [2] and Storm [3]. These platforms are delivering software as a service, written in garbage-collected languages, e.g., Scala, Java, C#, ML, JRuby, Python. The execution environment of garbage collected languages typically involves allocation of portion of memory to each application, and managing the allocated memory and garbage collection (GC) on behalf of the applications at runtime. This removes the burden of explicit memory management from the developers, e.g., handling memory leaks, removing dangling reference pointers and dead objects. Thus, garbage collected languages, such as Java, become increasingly popular and Java Virtual Machines (JVMs) are recognized as a leading execution environment for many big data software platforms today.

However, running applications on JVMs presents another layer of abstract execution environment on top of hardware virtualization (i.e., virtual machines and hypervisor) over a physical hosting server platform. Thus, the dynamic sharing of physical CPU and memory among multiple JVMs, combined with the unpredictable memory demands from applications, creates some open challenges for JVM configuration management. Most frequently asked questions include: (1) How can applications balance the sizes of their JVM heaps dynamically? (2) What type of heap structure can minimize heap space errors? (3) Which set of JVM heap parameters can we use to control the garbage collection overheads and improve application runtime performance? (4) Can we speed up the progress of application by enlarging the JVM heap size?

Researchers have attempted to address some of these questions. [4] shows that one can obtain the same performance by either explicit memory management or using garbage collection, provided that the garbage collected heap is sized appropriately with respect to the application. However, keeping the heap an appropriate size and balancing the heap sizes dynamically in a multi-application runtime environment are extremely difficult. Small heap will trigger frequent garbage collection, which results in performance degradation. Also collecting heap too frequently increases garbage collection (GC) cost. Moreover, small heap may lead to more frequent heap space errors, causing the application with dynamic memory demand to fail. On the other hand, heap can only be set to larger size when memory is plenty. Also too large a heap can induce paging, and swapping traffic between memory and disk is several orders of magnitude more expensive, significantly degrades the performance of the system. Several resource management technologies, such as memory overcommitment, heap resizing, and virtual machine memory ballooning, are proposed to address the problem of sudden changes in memory demands of applications in a consolidated environment. However, most of existing methods require modifications to JVM, guest virtual machine kernel or host OS kernel and hypervisor.

In this paper, we evaluate and analyze the effects of tuning JVM heap structure parameters and garbage collection parameters on application performance, without requiring any JVM, guest OS, host OS or hypervisor modification. Through our extensive measurement study, we show a num-

ber of valuable observations that help answering the set of questions listed above. First, we show that there is a direct correlation between application performance and its heap size. However, after heap size reaches a certain value, increasing heap size no longer improves the application performance in terms of the amount of actual work done. Second, we show that heap space error does not necessarily correspond to the conclusion that heap is full and all objects residing in heap are alive. Also, heap space error can be resolved by tuning JVM heap strcuture parameters. Third, dynamic configuring of JVM heap structure parameters and GC parameters may help minimizing GC overheads and improving application runtime performance. Our experiments also show that smaller heap sizes can achieve the same level of application performance as some of the larger heap sizes. Finally, we show that most of our measurement results are not specific to any garbage collection algorithm or any specific garbage collector implementation. We conjecture that these measurement results can help software developers of big data applications to better manage and configure their JVM runtime, achieving high performance big data computing at ease.

2. Related Work

Memory management research related to JVM performance can be broadly categorized into three categories: Memory overcommitment techniques, heap resizing algorithms and memory bloat management.

Memory overcommitment. Memory overcommitment is used at hardware virtualization layer by enabling virtual machines (VMs) to inflate and deflate memory using Balloon driver [5]. However, sharing policies are insufficient to determine when to reallocate memory and how much memory is needed. [6] proposes Memory Balancer (MEB), which dynamically monitors memory usage of each VM, and reallocates memory by predicting memory need of each VM. [7] proposed an application level ballooning technique, which enables resizing JVM heap dynamically, by modifying heap structure. However, they do not provide any sharing or resizing policy. [8] described a JVM ballooning by allocating Java objects. Balloon objects are then inflated/deflated depending on the resizing decision.

Heap Resizing. Independent but complimentary to the development of Java ballooning mechanism, several research efforts have been devoted to heap resizing policies. [9] evaluated JVM metrics as application performance indicator, and proposes a memory sharing policy by considering applications memory demands and available physical memory. [10] proposed CRAMM to track memory demands of applications at runtime, and predict appropriate heap sizes, aiming at garbage collection without paging or minimizing paging while maximizing application throughput. [11] presented a resource aware garbage collection technique with different heap resizing policies. [12] proposed Ginkgo, a policy framework to correlate application performance and its memory demand by using runtime monitoring. However, few existing efforts have conducted a systematic study on JVM configuration management with respect to heap structure parameters and GC parameters, such as showing the effect of changing the size ratios between young generation and old generation, and understanding the root cause of heap space errors.

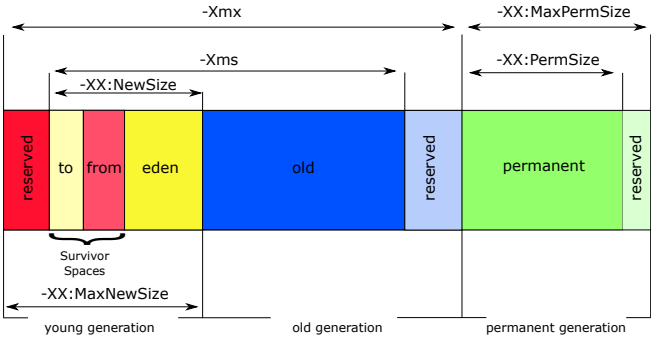


Figure 1: JVM Structure

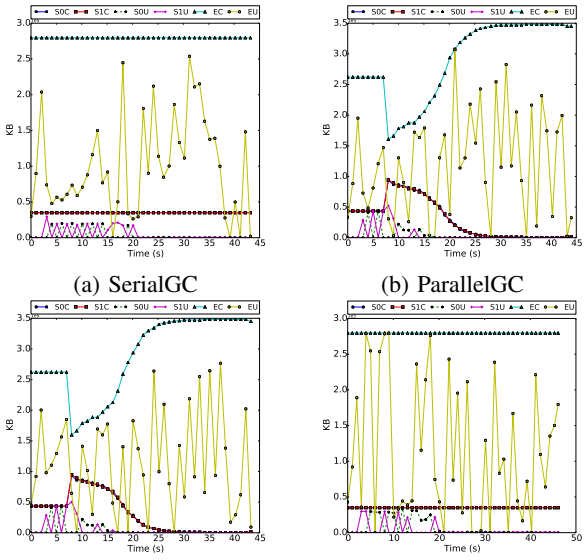
Memory Bloat Management. To improve the runtime efficiency of JVM heap, researchers have identified different classes of memory bloats caused by data type design [13], [14] and proposed new design for JVM data types that can minimize or remove undesirable memory bloats.

3. Overview and Observations

3.1. JVM Heap Structure

In JVM, memory management is done by partitioning the memory region, called JVM heap, into groups, where each group consists of one or more regions and is called a generation. Objects in one generation are of a similar age. Figure 1 shows JVM memory structure. Most commonly, there are two major types of generations: young and old. In **young generation**, recently created objects are stored. Typically, young generation is further partitioned into 3 major spaces (regions): *eden* space and two survivor spaces: *to* and *from*. New objects are first allocated in *eden* space. The allocation is done contiguously, enabling fast placement. Minor GC is triggered on allocation failure, whereas Full GC is triggered when meeting a threshold, i.e., some percentage of *old* generation has been filled. Upon a minor GC, objects that are not matured enough to move to old generation, but survived at least one garbage collection are stored in *from* survivor space. The *To* survivor space is unused.

In **old generation**, objects that are survived a certain number of garbage collection are stored. By default the old generation is twice as big as young generation. Keeping young generation small makes it collected quickly but more frequently. Old generation grows more slowly but when reaches to certain threshold value, a full (heap) garbage collection (GC) will be triggered. The reserved region in both young and old generations and the permanent generation are to allow JVM to expand its heap. The default setting of old/young capacity ratio is fixed and 2. However, capacities of spaces in young generation may vary from one garbage collector to another. For derby workload with heap size of 1024 MB, Figure 2 shows that while the default capacity ratio of eden and survivor spaces is fixed for Serial GC or Concurrent Mark Sweep (CMS) Garbage Collector, but the



(a) SerialGC (b) ParallelGC

(c) ParallelOldGC (d) ConcMarkSweepGC

Figure 2: Capacities and Utilizations of Young Generation (S0C/SIC: capacity of survivor space 0 / survivor space 1, EC/EU: eden space capacity/utilization, S0U/S1U: utilization of survivor space 0 / survivor space 1).

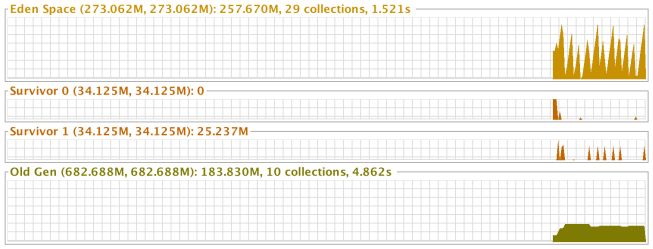
capacities are dynamically changing at runtime for Parallel GC Collector or Parallel Old Garbage Collector.

Figure3, produced using Visual VM [15], shows the utilization of eden and survivor spaces in young generation and the utilization of old generation for Dacapo h2 workload with heap sizes of 1024 MB and 384 MB. We make three observations: (1) For heap size of 1024MB, in young generation, Eden space is 94% full (257.670/273.062=0.94) and Survivor 1 is 74% full (25.237/34.125=0.739) while Survivor 0 is empty. The utilization of old generation is 27% (183,830/682.688=0.269). (2) For heap size of 384 MB, in young generation, Eden space is 85% full, while the utilization of old generation is over 81% (208.300/256.000=0.813). (3) In both cases, a number of minor GC and full GC are performed and the time spent on minor GC and full GC are also measured: 92 minor GCs are performed for heap size of 384 MB, compared to 29 minor collections for heap size of 1024 MB, though both have 10 full GCs.

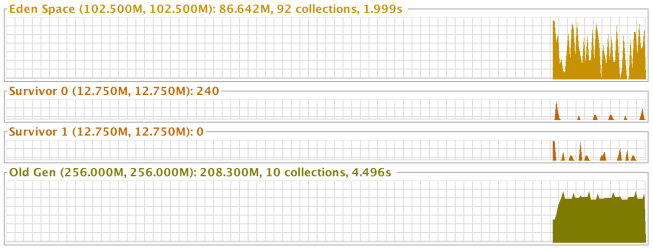
3.2. Garbage Collection (Minor GC vs Full GC)

Garbage collection is the process of identifying dead objects, which are no longer referenced by application and reclaim their heap space. Garbage collector is also responsible for object allocation and it can be performed on generations separately or on the entire heap.

Young generation Minor GC is triggered when new object allocation fails because there is insufficient space in young generation. During Minor GC, dead objects are identified, and live objects in eden space, which are not mature enough to move to *old generation*, are moved to the survivor *to* space, which was previously unusued. Live objects in the survivor *from* space are moved either to *to* space or *old generation* based on their ages. After minor GC, survivor *from* space becomes the survivor *to* space, and vice versa. In our experiments, Serial Garbage Collector is



(a) Heap Utilization with Heap Size=1024MB



(b) Heap Utilization with Heap Size=384MB

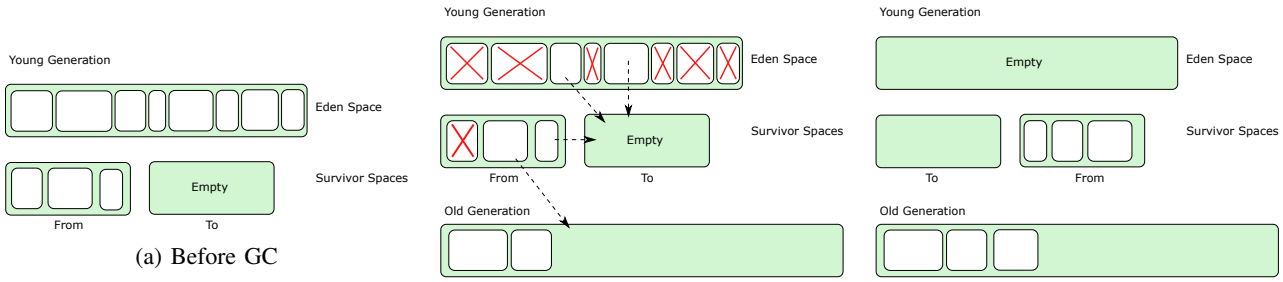
Figure 3: Heap Utilization of Dacapo h2 Workload

used during Minor GC by default. Figure 4 illustrates how it works when Minor GC is triggered. Allocation triggers minor GC in young generation when there is insufficient space in Eden to place a new object. Threshold triggers full GC in old generation when heap usage in old generation reaches a specified threshold, **Full GC** is performed on entire heap, the most costly collection compared to Minor GC. It is also more complex because there is no helper space, like survivor space for Minor GC, to enable compaction. A popular policy used in Full GC is called *mark-sweep-compact*, in which dead objects are identified first and live objects are moved to the head of old generation. Once they are placed contiguously, the rest of the heap is reclaimed. Both Minor GC and Full GC suspend applications from executing and take full control of CPU when switched on. Figure 5 shows CPU usage of application (blue curve) and garbage collection (yellow curve) running concurrently for h2 workload with heap sizes of 1024 MB and 384 MB.

3.3. Garbage Collection Overhead

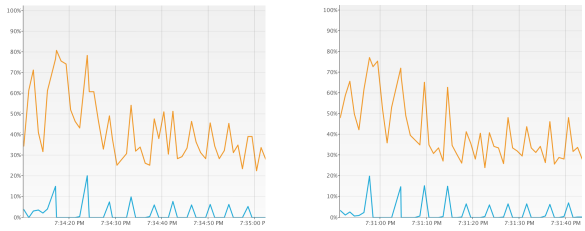
Full GC and minor GC both stop the execution of JVM applications when performing garbage collection. There are a number of factors contribute to the GC overhead. For simplicity, we focus our discussion on young generation collections. As stated earlier, minor GC is performed when an allocation fails, namely, when there is insufficient space in the eden space to meet the allocation request. Depending on the size of the new object, minor GC can be triggered with varying sizes to be collected from the eden space. Since both object allocations and GC preserve compaction of the heap regions, GC is performed only on the utilized portion of the Eden space. Hence, GC overhead depends on the frequency of allocation failure and the eden space utilization.

In general, GC is more centered on identification of alive objects, instead of dead ones. For the sake of compaction, it involves copying alive objects from eden to survivor, or from young generation to old generation. As a result, copying cost emerges as the third factor in GC overhead.



(a) Before GC (b) During GC (c) After GC

Figure 4: Garbage Collection Process (Minor GC)



(a) Heap Size = 1024MB (b) Heap Size = 384MB

Figure 5: CPU Utilization of h2 Workload

TABLE 1: Garbage Collection Details

Eden Util. (kb)	Alive Size (kb)	Garbage Size (kb)	GC Cost (msec)
139776	9877	129899	68,1877
139776	10291	129485	73,4889
139776	10691	129085	71,1739
139776	10798	128978	84,3332
157248	10163	147085	124,8909
157248	12556	144692	148,5655
157248	17472	139776	221,4381

These factors can be validated by GC report, which can be obtained by adding **-XX:+PrintGCDetails** parameter to the run command. In Table 1, we present some GC details performed on young generation for compiler.compiler benchmark application with heap size of 512MB. It shows that 1 time GC cost is proportional to utilized eden space and size of the alive data.

Parameter Tuning for Heap Structure and GC. JVM heap structure and garbage collection can be configured by tuning the following JVM parameters:

-Xmx, -Xms parameter specifies the maximum and minimum heap size that JVM can manage respectively.

-XX:NewRatio=n sets ratio of old/young generation sizes to *n*. Default NewRatio value is 2.

-XX:+UseSerialGC enables the serial garbage collector. **-XX:+UseParallelGC** activates parallel garbage collector for the young generation.

-XX:+UseParallelOldGC enables the parallel garbage collector for both of the young and old generations.

-XX:+UseMarkSweepGC activates concurrent mark sweep garbage collector for the old generation.

4. Experimental Method

4.1. Experimental Environment

In all of our experiments, we disable *UseGCOverheadLimit* parameter since we want to focus on application performance in any case unless it results in heap space error. We use Oracle's HotSpot Java Runtime Environment with version 1.8.0_65. In addition, we use jstat and VisualVM to monitor heap usage and CPU usage of our benchmark applications (see next subsection). Measurement period is set to 1 second. Unless otherwise is stated, in our experiments we use Serial Garbage Collector with default newRatio value 2. The hardware platform is a Mac OSX Yosemite 10.10.5, with two physical cores of 2.9 GHz Intel i5 and 1867 MHz DDR3 memory of 8 GB.

4.2. Test Programs

We select 4 benchmark applications from DaCapo [16] and SPECjvm2008 [17] benchmark suites without any modification. Here are the chosen benchmarks:

h2: As part of DaCapo suit, h2 executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application. Unless otherwise is stated, h2 benchmark is run 10 times.

derby: This benchmark uses an open-source database written in pure Java. It is synthesized with business logic to stress the BigDecimal library. The focus of this benchmark is on BigDecimal computations and database logic.

serial: This benchmark serializes and deserializes primitives and objects, using data from the JBoss benchmark. The benchmark has a producer-consumer scenario where serialized objects are sent via sockets and deserialized by a consumer on the same system.

compiler.compiler: This benchmark uses OpenJDK front end compiler to compile a set of .java files. The code compiled is javac itself and the sunflow sub-benchmark from SPECjvm2008.

4.3. Collectors and Configuration Parameters

In all experiments, SerialGC is used as the baseline collector. It uses a single thread with copying and compaction preserving mechanisms for young generation. It stops all applications when performing minor GC. We compare SerialGC with ParallelGC, ParallelOldGC and the Concurrent Mark-Sweep (CMS) Collector. ParallelGC is similar to SerialGC but performing minor GC using multiple threads instead of single thread. ParallelOldGC uses copying and compaction preserving mechanisms for GC in both young generation and old generation and is multi-threaded. The CMS Collector marks the reachable objects and then sweeps across the allocation region to reclaim the unmarked objects (spaces). It is non-copying and non-compacting in that it does not move allocated objects and neither compact them. In contrast, copying collectors proceed by copying reachable

objects to an empty copy space. Also upon GC, it suspends application only at the beginning and end of the collection marking and sweeping phases. CMS GC are concurrent with the application executions (no stop-the-world) and it uses multiple threads.

4.4. Metrics

The first performance metric is the ratio of old generation over the new generations. By default of 2, it states that the old generation is doubled the size of the young generation. In most of our experiments, we vary this ratio from 1 to 21. Another important metric is the GC overhead, consisting of three components: allocation failure frequency, utilization of corresponding generation, and size of alive objects that will create cost of copying. Let n denote number of GC as a frequency indicator, α_i denote the size of garbage data and β_i denote the size of alive objects in the i^{th} GC event. Let c_1, and c_2 denote machine specific constants for scanning and copying heap unit respectively. Then we can calculate GC overhead as follows:

$$GC\_Overhead = \sum^{n} c_1(\alpha_i + \beta_i) + c_2(\beta_i)$$

Other metrics in our measurement study include heap size, heap utilization, execution time, CPU utilization.

5. Experiments and Analysis

In this section, we evaluate the effect of JVM heap size, structure and GC algorithm on memory intensive benchmark applications by tuning above JVM parameters.

5.1. Heap Utilization Heap Space Error

In this set of experiments, we analyze the scenarios that causes heap space error. One may think that the reason behind heap space error is failing object allocation because the entire heap is filled and all the objects in it are alive and no additional space to reclaim. However, our experiments show that depending on the GC algorithm, the size of the survivor and eden spaces may change at runtime. Since object is allocated only in eden space or old generation, allocation may fail when eden space or old generation is filled, resulting a heap space error. However, there could be some unused survivor spaces that are not small. Table 2 illustrates this observation, where we set newRatio value to 1, and heap size to 256MB for h2 and derby benchmarks, and 128 MB for compiler.compiler.

5.2. Effect of Heap Size and GC Overhead

The most straightforward approach to eliminate Heap Space Error is to increase JVM heap size. Figure 6(a)(b) show that (i) JVM crashed for heap size of 256MB, and (ii) increasing heap size from 256MB to 384MB resolves Heap Space Error, for h2 workloads with Parallel and ParallelOld GC respectively. Figure6(c)(d) show that JVM crashed for heap size of 128MB, and (ii) increasing heap size from 128MB to 256MB resolves Heap Space Error for compiler.compiler benchmark. In all these figures, the smallest heap size is too small to run the respective benchmark, which causes heap size error and crashed.

Next we examine the effects of heap size. Increasing heap size leads to less frequent allocation failure and improves application performance by reducing GC frequency and thus overall GC overhead. Figure 7a and Figure 7b show that by increasing heap size from 256MB to 384MB, it reduces GC overhead drastically for derby and h2 workloads. We also see similar result in Figure 7d for compiler.compiler benchmark when we increase heap size from 128MB to 256MB. We also observe that after some point, increasing heap size no long reduces GC overhead, nor improve application performance. In summary, we showed that increasing heap size may eliminate Heap Space Error, and improve application performance by reducing GC overhead. However, increasing heap size may not be possible in a consolidated environment where memory is a primary bottleneck. This motivates us to examine other tuning options that explore more efficient heap utilization.

5.3. Tuning newRatio Parameter

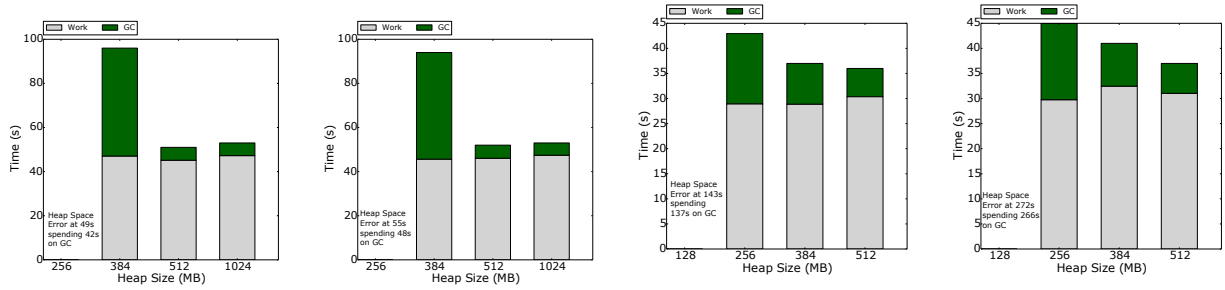
Another cause for Heap Space Error is due to poor heap utilization in JVM. This is the case when Heap Space Error is experienced but we can still observe some large but unused survivor spaces as shown in Figure 3. Our goal is to reduce the size of unused survivor spaces, so that more portion of the heap is used for allocation to objects. One way to achieve this goal is to increase the parameter newRatio value, which will increase the size of old generation and decrease the size of young generation. Since survivor spaces are sub-partitions of the young generation, increasing the parameter newRatio will decrease the size of young generation and thus the size of survivor regions. In this set of experiments, we use h2, derby, and serial benchmarks with heap of 256MB, and compiler.compiler benchmark with heap of 128MB. Figure 9a shows that increasing newRatio value from 1 to 21 improves the runtime performance of application by reducing GC overhead. Figure9b, Figure9c, and Figure9d show that GC overhead is lower (i) for derby benchmark when newRatio is 8, (ii) for serial benchmark when newRatio is 3, and (iii) for compiler.compiler when newRatio is 8. Table 3 shows the number of Minor GCs and Full GCs, the time spent on Minor GC and Full GC with varying newRatio values for derby benchmark with heap of 256MB. We observe that as Minor GC overhead increases, Full GC overhead will decrease, as we increase the parameter newRatio value. However, the total GC overhead decreases until the newRatio value reaches 8 for derby benchmark. When newRatio value increases to 13 or higher, the total GC overhead increases, showing that there exists a newRatio value that minimizes total GC overhead. This set of experiments shows that depending on the application behavior and lifetime of the objects, newRatio value can be set appropriately for achieving optimal JVM runtime performance.

5.4. Effects of Garbage Collectors

This section compares benchmark performance under varying garbage collectors. In this set of experiments, we run h2, derby and serial benchmarks with heap sizes of 1024MB and 256MB. We run compiler.compiler benchmark

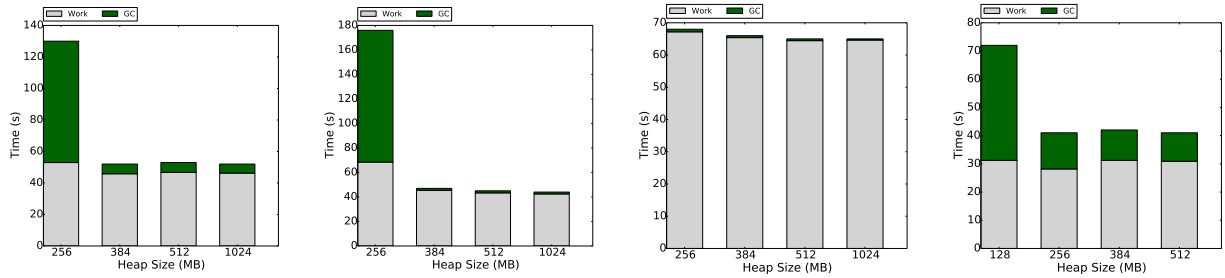
TABLE 2: Heap Utilization at the time of Heap Space Error

Benchmark	GC	Eden Cap.	Eden Used	Surv. 0 Cap.	Surv. 0 Used	Surv. 1 Cap.	Surv. 1 Used	Old Cap.	Old Used
h2	ParallelGC	44032.0	44032.0	16384.0	0	43520.0	0	131072.0	131072.0
h2	ParallelOldGC	44032.0	44032.0	16384.0	0	43520.0	0	131072.0	131072.0
derby	ParallelOldGC	88064.0	88064.0	19968.0	0	20992.0	0	131072.0	131040.6
compiler.compiler	ParallelGC	22528.0	22528.0	21504.0	0	21504.0	0	65536.0	65493.5
compiler.compiler	ParallelOldGC	22528.0	22527.3	21504.0	0	21504.0	0	65536.0	65533.6



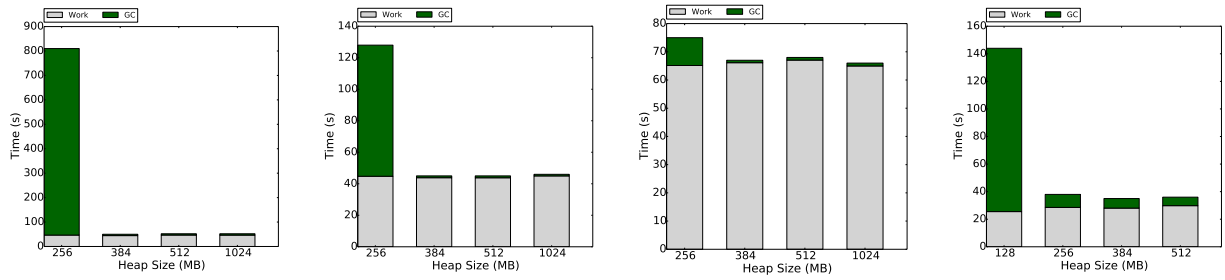
(a) h2-ParallelGC (b) h2-ParallelOldGC (c) compiler.compiler-ParallelGC (d) compiler.compiler-ParallelOldGC

Figure 6: Heap Space Error Elimination by Increasing Heap Size



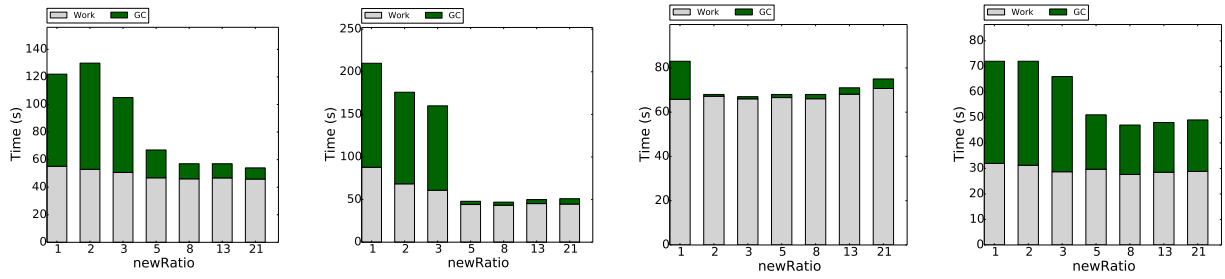
(a) h2 (b) derby (c) serial (d) compiler.compiler

Figure 7: Heap Size vs Running Time (SerialGC)



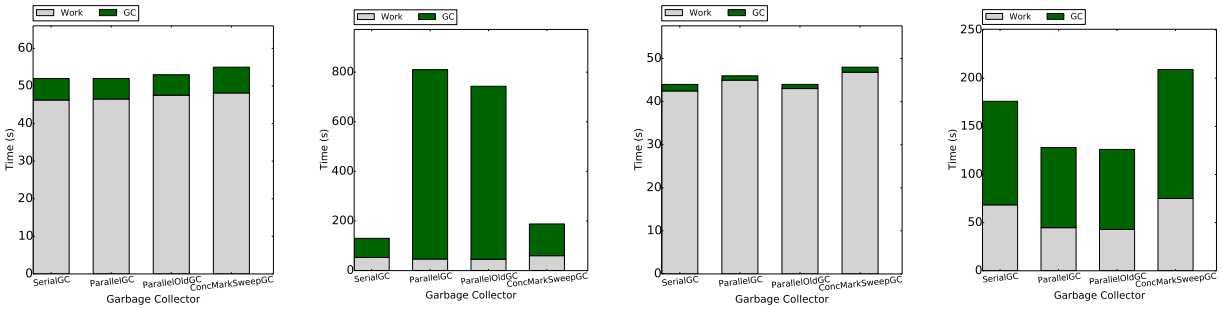
(a) h2 (b) derby (c) serial (d) compiler.compiler

Figure 8: Heap Size vs Running Time (ParallelGC)



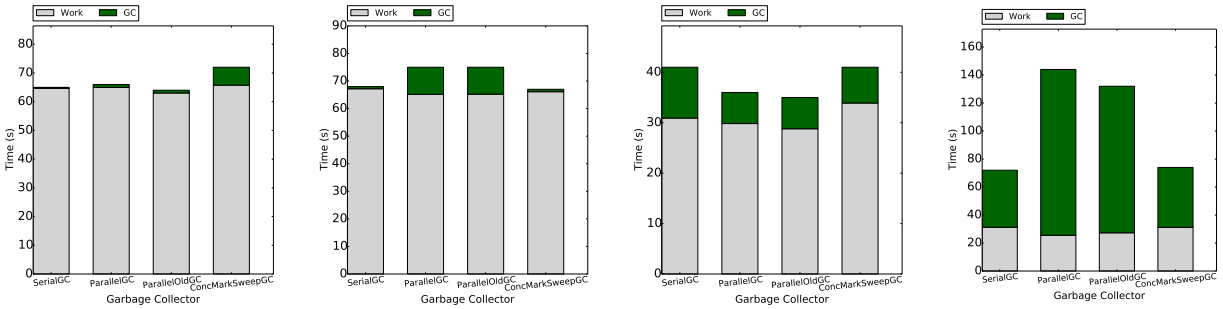
(a) h2 (b) derby (c) serial (d) compiler.compiler

Figure 9: newRatio vs Running Time



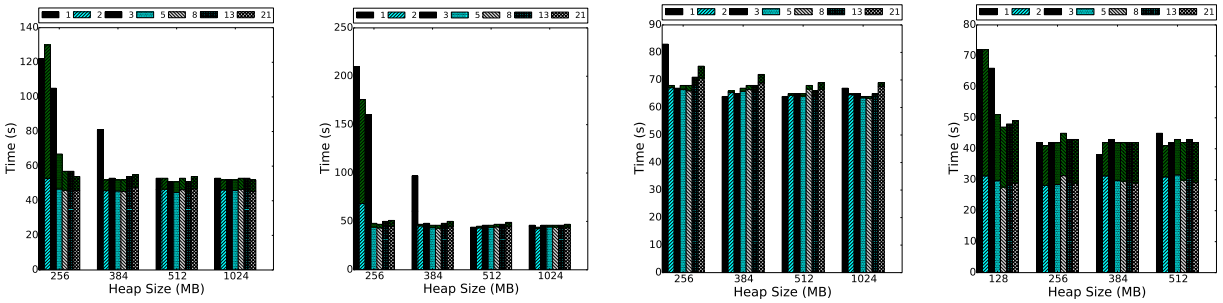
(a) h2 - 1024 MB (b) h2 - 256 MB (c) derby - 1024 MB (d) derby - 256 MB

Figure 10: Garbage Collector vs Runtime (h2 and derby)



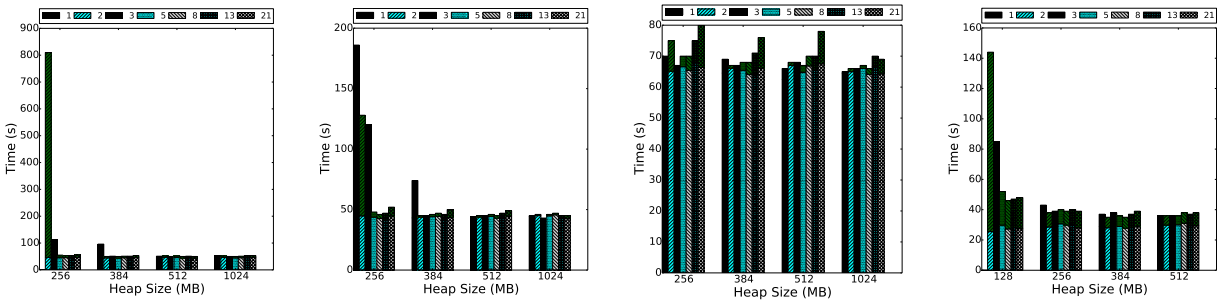
(a) serial - 1024 MB (b) serial - 256 MB (c) compiler - 512 MB (d) compiler - 128 MB

Figure 11: Garbage Collector vs Runtime (serial and compiler)



(a) h2 (b) derby (c) serial (d) compiler.compiler

Figure 12: Heap Size & Structure vs Actual Work & GC Overhead (SerialGC)



(a) h2 (b) derby (c) serial (d) compiler.compiler

Figure 13: Heap Size & Structure vs Actual Work & GC Overhead (ParallelGC)

TABLE 3: GC Overhead with Varying NewRatio Values

newRatio	# of YoungGC	YoungGC Cost	# of FullGC	FullGC Cost	Total GC Cost
1	18	0.383	824	121.720	122.103
2	36	0.510	752	107.006	107.516
3	63	0.617	717	98.442	99.059
5	1192	2.315	12	1.499	3.815
8	1788	2.946	6	0.704	3.650
13	2788	4.365	4	0.357	4.722
21	4344	5.926	4	0.354	6.280

with heap size of 512MB and 128MB. Figure 10a and Figure 10b show that similar performance can be obtained when large heap is used for h2 benchmark. However, when heap is small, Serial and ConcMarkSweep GCs outperform Parallel and ParallelOld GCs. Figure10c shows similar results for derby benchmark with large heap. However, in contrast to h2 benchmark, when heap is small, Figure 10d shows Parallel and ParallelOld GCs outperform Serial and ConcMarkSweep GC. For compiler.compiler benchmark, Figure 11a and Figure 11b show that serial benchmark exhibits similar behavior as to h2. Figure11d shows that with small heap, Serial and Mark-Sweep GCs have lower GC Overhead and higher performance for compiler.compiler benchmark. Furthermore, for compiler.compiler workload, performance may differ even with larger heaps as shown in Figure 11c. In summary, the performance of collectors may vary with respect to different heap size and different application behavior. Additionally, heap space error can be a result of poor selection of garbage collector or poor setting of the newRatio parameter in addition to out of heap memory. Note that eden and survivor regions are fixed in size for Serial GC and ConcMarkSweep GC, but they are dynamically changing in Parallel and ParallelOld GCs.

5.5. Effect of Heap Size on Applications

This section investigates whether GC overhead and actual work of an application are highly correlated or completely independent processes. Given that object allocation and data access are parts of application's actual work, we want to analyze the effect of heap size, newRatio values on the time spent for performing actual work. In this set of experiments, we run h2, derby, and serial benchmarks with heap size varying from 256, 384, 512, to 1024 MB, and run compiler.compiler benchmark with heap size varying from 128, 256, 384 to 512 MB. We also vary newRatio value from 1 to 21. Figure12a, Figure12b, Figure12c and Figure12d present results for h2, derby, serial and compiler.compiler benchmarks respectively. These figures show time spent on performing actual work for specific application is the same regardless of the heap size or heap structure. The reason is that the data access cost is fixed, since they are all regular memory access, and the cost of object allocation is fixed because of compaction preserving feature of JVM. Figure13 shows similar results when using ParallelGC.

6. Conclusion

We have studied the effects of tuning JVM heap structure parameters and garbage collection parameters on application performance, without requiring any JVM, guest OS, host OS or hypervisor level modification. Our extensive measurement study shows a number of interesting observations: (1) Increasing heap size does not increase application performance after a certain point. (2) Heap space error does not necessarily indicate heap is full and all objects in the heap are alive and heap space errors can be resolved by tuning JVM parameters. (3) By tuning JVM heap structure and GC parameters, we can achieve the same application performance using smaller heap sizes. Most of our measurement results are not specific to any garbage collection algorithm or any specific garbage collector implementation. We conjecture that our results can help software developers of big data applications to achieve higher performances by better management and configuration of their JVM runtime.

Acknowledgment

This research has been partially support by the National Science Foundation under Grants IIS-0905493, CNS-1115375, NSF 1547102, SaTC 1564097, and Intel ISTC on Cloud Computing, and an RCN BD Fellowship, provided by the Research Coordination Network (RCN) on Big Data and Smart Cities. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the RCN or National Science Foundation.

References

[1] "Apache hadoop map reduce," http://hadoop.apache.org.

[2] "Apache spark project," http://spark.apache.org.

[3] Storm, "Apache Storm project," 2016, retrieved March 2016 from http://storm-project.net/.

[4] M. Hertz, Y. Feng, and E. D. Berger, "Garbage collection without paging," in *ACM SIGPLAN PLDI*, 2005.

[5] C. A. Waldspurger, "Memory resource management in vmware esx server," *SIGOPS Oper. Syst. Rev.*, 2002.

[6] W. Zhao, Z. Wang, and Y. Luo, "Dynamic memory balancing for virtual machines," *SIGOPS Oper. Syst. Rev.*, 2009.

[7] T.-I. Salomie, G. Alonso, T. Roscoe, and K. Elphinstone, "Application level ballooning for efficient server consolidation," ser. EuroSys, 2013.

[8] R. Mcdougall, W. Huang, and B. Corrie, "Cooperative memory resource management via application-level balloon," 2011, uS Patent App. 12/826,389.

[9] N. Bobroff, P. Westerink, and L. Fong, "Active control of memory for java virtual machines and applications," in *ICAC*, 2014.

[10] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss, "Cramm: Virtual memory support for garbage-collected applications," ser. OSDI, 2006.

[11] M. Hertz, S. Kane, E. Keudel, T. Bai, C. Ding, X. Gu, and J. E. Bard, "Waste not, want not: Resource-based garbage collection in a shared environment," *SIGPLAN Not.*, 2011.

[12] M. Hines, A. Gordon, M. Silva, D. da Silva, K. D. Ryu, and M. Ben-Yehuda, "Applications know best: Performance-driven memory overcommit with ginkgo," in *CloudCom*, 2011.

[13] N. Mitchell and G. Sevitsky, "The causes of bloat, the limits of health," ser. OOPSLA, 2007.

[14] Y. Bu, V. Borkar, G. Xu, and M. J. Carey, "A bloat-aware design for big data applications," *SIGPLAN Not.*, 2013.

[15] "Visual vm," https://visualvm.java.net/.

[16] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The dacapo benchmarks: Java benchmarking development and analysis," in *ACM SIGPLAN OOPSLA*, 2006.

[17] Standard Performance Evaluation Corporation, "Specjvm2008," 2016, https://www.spec.org/jvm2008/.