OS-Caused Large JVM Pauses: Investigations and Solutions

Zhenyun Zhuang*, Cuong Tran, Haricharan Ramachandra, Badri Sridharan
LinkedIn Corporation, 2029 Stierlin Court Mountain View, CA 94043 United States

* Corresponding author. Email: zhenyun@gmail.com
Manuscript submitted June 5, 2016; accepted June 30, 2016.
doi: 10.17706/jsw.11.10.1008-1025

Abstract: For customer-facing Java applications (e.g., online gaming and online chatting), ensuring low latencies is not just a preferred feature, but a must-have feature. Given the popularity and powerfulness of Java, a significant portion of today's backend services are implemented in Java. JVM (Java Virtual Machine) manages a heap space to hold application objects. The heap space can be frequently GC-ed (Garbage Collected), and applications can be occasionally stopped by JVM GC pauses. As a result, ensuring low JVM GC pauses is critical for business applications that have latency SLA (Service Level Agreements).

In our production system, we found out that there are some (and large) JVM pauses cannot be explained by application-level activities and JVM activities during GC; instead, they are caused by OS mechanisms. In this work, we investigate the large JVM pauses caused by OS. We characterize various such problems into two scenarios. For both scenarios, we successfully root-cause the reasons and propose solutions. The findings can be used to mitigate the long JVM pauses and enhance JVM implementations. We share the knowledge and experiences in this writing.

Key words: Java, JVM pause, OS, garbage collection, performance.

1. Introduction

Online applications such as social networks (e.g., linkedin.com) are customer-facing (e.g., online gaming and online chatting), thus ensuring low latencies is not just a preferred feature, but a must-have feature for these applications. Various studies have suggested that 200ms latency is the maximum latency an online user can tolerate before going away. Because of this, ensuring lower-than-200ms (or even smaller) latency should be part of the defined SLA (Service Level Agreements) for applications serving online users.

Given the popularity and powerfulness of Java platforms, a significant portion of today's online platforms run Java. One example is Oracle Java Cloud [1], which provides cloud-deployed Java applications using WebLogic Server [2]. Despite tremendous efforts put at various layers (e.g., application layer, JVM layer) to improve the performance of Java applications, based on our production experiences, Java applications can occasionally experience unexplainable large STW (Stop-The-World) JVM pauses that cannot be explained by typical known reasons at application layer.

Java-based applications run in JVM (Java Virtual Machine), which manages a heap space to hold application objects. The heap space can be frequently GC-ed (Garbage Collected) [3], and JVM could be stopped during GC and JVM activities (e.g., Young or Full GC), which introduce STW (Stop-The-World) pauses to the applications. Depending on JVM options supplied when starting the JVM instance, various types of GC and JVM activities are logged into GC log files. Though GC-induced STW pauses that scan/mark/compact heap objects are well-known and paid much attention to, as we find out, there are some (and large) STW pauses could be caused by OS (Operating System) mechanisms. In our production

environments, we have been seeing OS-caused large STW pauses (>11 seconds) happened to our mission-critical Java applications. Such pauses cannot be explained by application-level activities and the garbage collection activities during GC.

For latency-sensitive and mission-critical Java applications, the larger-than-SLA STW pauses are intolerable. Hence we spent efforts investigating the problems. We successfully reproduced the problem in lab environments and root caused the reasons. We find out that, JVM mechanisms, coupled with OS-level features, give rise to unique problems that are not present in other deployment scenarios. Though the investigations and findings are Linux-specific, given the wide deployment of Linux OS, particularly in server markets, the findings apply to a significant portion of backend platforms. In addition , other platforms are expected to expose similar problems of varying degree of severity. On one hand, JVM mechanisms are largely universal across OS platforms. On the other hand, most OS platforms have mechanisms of swapping and reclaiming. These similarities make us believe our findings can help on similar problems and solutions in other cloud platforms.

There are multiple scenarios where such problems can happen, and for ease of grasping, we characterize into two scenarios: (1) OS has memory pressure; (2) OS has heavy IO. For instance, in the second scenario, the large STW pauses are caused by GC logging write() calls being blocked. These write() calls, though are issued in buffered write mode (i.e., non-blocking IO), can still be blocked due to certain OS internal mechanisms related to “writeback” [4] IO activities. Specifically, w hen buffered write() needs to write to a file, it firstly writes to memory pages in OS cache. These memory pages can be locked by OS cache-flushing mechanism of "writeback”, which could last for substantially long time when IO traffic is heavy. Furthermore, for typical production applications, the application-level logging (e.g., access logs) and log rotations also prove to be sources of background IO traffic.

In a nutshell, this work considers the performance of Java applications, and identified a set of Linux OS features, as well as the interactions between JVM and OS, can adversely affect Java application performance in certain scenarios. After root-causing the problems, we provide solutions to mitigate the challenges. Though the solutions have been verified working on Linux OS, on which most of LinkedIn products run, they can easily be applied to similar setup of other platforms to achieve better performance. More importantly, the studies gained could be incorporated into future designs of JVM or OS features.

In this work, we share our findings. For the remainder of the paper, after providing necessary technical background in section II, we present the scenarios in Section III and Section IV, respectively. For both scenarios, we root cause the problem, propose solutions and performance evaluations. Section V gives related works, and finally Section VI concludes the work.

2. Background

We begin by providing background information regarding JVM heap management and GC (Garbage Collection), File System, OS page cache, Linux memory management, and typical application logging and log rotation.

2.1. JVM, Heap Management and GC

A JVM (Java Virtual Machine) is a virtual environment to run Java applications. JVM has well defined specification that can be implemented by different vendors, with the most widely adopted JVM being Oracle's HotSpot. JVM implementations typically feature a set of optimizations aimed at running the program faster.

Java programs run in JVM, and the area of memory used by the JVM is referred to as heap. JVM heap is used for dynamic Java objects allocations and is divided into generations (e.g., Young and Old). Java objects are firstly allocated on the Young generation; when unused, they are collected by a mechanism called GC.

When GC occurs, objects are checked for reference counters starting from a root object. If the reference counter of an object drops to zero, the object is deleted and the corresponding memory space is reused. Some phases of GC process require applications to stop responding to other requests, a behavior commonly referred to as STW (Stop the world) Pause. One of the important objectives of Java performance is to minimize the durations of GC pauses.

JVM enables automatic memory management for Java applications by providing garbage collection on heap space. Depending on specific garbage collectors, JVM may scan/mark/remark/compact objects to free heap space for later usage during garbage collection. These actions may incur STW pauses to the applications, and during the pauses the application has to stop processing. Based on JVM options provided during command line instance starting, various GC statistics can be recorded into a gc log file.

2.2. File System and Asynchronous IO

Many types of file systems have been designed to support various types of media and different scenarios. A special file system is *tmpfs*, which uses RAM to create temporary file systems for short-term use. To deal with the potential inconsistency during system crush, journal file systems introduce transaction-level consistency to file system structures. Many popular file systems such as EXT4 [5] fall into this category.

To write to file systems, system calls such as write() are provided. Writing can operate in asynchronous mode (i.e., non-blocking, or buffered) or synchronous mode (i.e., blocking). In typical scenarios, asynchronous writing will modify the data in memory first and the write() call returns before the data are persisted into disk.

2.3. OS Page Cache and Cache Writeback

Page cache is provided by OS to expedite the data access for slower (than RAM) disk drives. OS keeps a portion of the RAM for caching the file data. Modified page cache will be periodically written to disk files for persistence. There are several mechanisms controlling how soon a particular “dirty” (i.e., containing newer-than-disk data) page needs to be written back. When writing happens, it appears to the users that there is bursty IO happening to disks.

2.4. Linux Memory Management

Memory management is one of the critical components of Linux OS. The virtual memory space is divided into pages of fixed sizes (e.g. 4KB). Over the years, many features of memory management have been designed to increase the density and improve the performance of running processes.

Page reclaiming Linux OS maintains free page lists to serve applications' memory request. When the free pages drop to a certain level, OS will begin to reclaim pages and add them to the free lists. When performing page reclaiming, page scanning is needed to check the liveness of allocated pages. Both background scanning (performed by *kswapd* daemon) and foreground scanning (performed by processes) are used. Oftentimes the foreground page reclaiming is referred to as direct reclaiming or synchronous reclaiming, which represents a more severe scenario where the system is under heavy memory pressure, and the application stops during the process.

Swapping Swapping is designed to increase process density. When free memory is low, Linux will swap memory pages out to swap spaces to allow for new processes being invoked. When the memory space corresponding to swapped-out pages is active again, these pages will be swapped in.

THP Transparent huge pages [6] is a relatively new feature that aims to improve the processes' performance. With larger page sizes (e.g., 2MB) , the number of page table entries is reduced for a particular process. More virtual address translations can be covered by TLB (Translation Lookaside Buffer) lookup, hence higher TLB hit ratio [7]. Though the benefit of using huge pages has long been understood, the use of

huge pages was not easy. For instance, the huge pages need to be reserved when OS starts, and processes have to make explicit calls to allocate huge pages. THP, on the other hand, promises to avoid both problems and thus is enabled by default. THP allows regular pages to be collapsed into transparent huge pages (THPs) through two types of collapsing: background collapsing by *khugepaged* daemon and direct collapsing by applications requesting memory. Note that we use THP to denote the feature, while use THPs to denote the transparent huge pages.

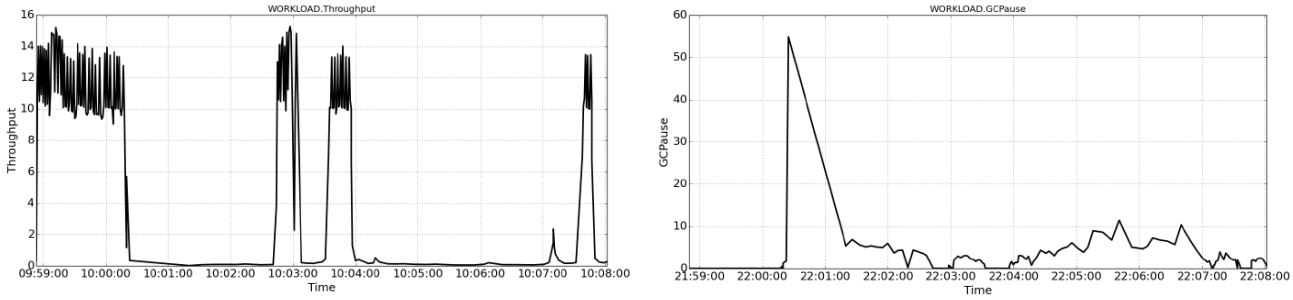
2.5. Application Logging and Log Rotation

Server applications oftentimes need to log its activities (e.g., received queries, sent responses). A typical example is a web server which maintains a history of page requests. Logging frameworks such as Apache log4j [8] are adopted to facilitate and format the application logging.

In many non-trivial deployment scenarios such as business environments, log rotation is required to automate the process of archiving application logs. Many logging practices are used, and the typical process is to periodically read some dated log files, process them, compress the raw files or relocate them. Whenever the rotation kicks off, a set of log files accumulated during last rotation period are read, processed and compressed.

3. Scenario I: Application Steady State with Memory Pressure

In this scenario, the Java application has entered steady state since being started, but OS has memory pressure (i.e., the free memory becomes scarce).



(a) JavaApp throughput (K allocations/sec) (b) JavaApp GC pauses (second)

Fig. 1. Scenario I: Application performance during steady state with memory pressure.

3.1. Motivation

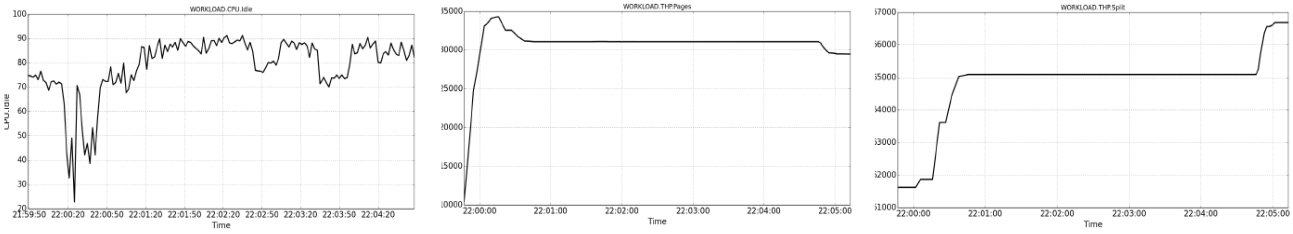
Two applications are used, one is written in Java, the other in C++. For easy presentation, we refer to the Java application as JavaApp, and the C++ one BackgroundApp. JavaApp is used to represent real production applications, while BackgroundApp simply consumes computer resources of memory to mimic production environments. JavaApp keeps allocating Java objects, and also periodically discards objects such that they can be reclaimed during JVM GC.

As a motivation example, we start a JavaApp is started with 20 GB heap and runs for 1 hour. Then a BackgroundApp starts and begins to allocate 50 GB of memory. We observe similar behaviors using a JavaApp as the second application, but for easy to present, we choose a BackgroundApp. We plot the performance numbers in Fig. 1.

In Figure 1(a), we saw that the JavaApp achieves a steady 12K/sec throughput in the beginning. Then the throughput sharply drops to zero which lasts for about 2 minutes. From then on, the throughput varies wildly. Sometimes it comes back at 12K/sec, other times drops to zero again.

In Figure 1(b), the GC pauses are almost zero in steady state, then it rises to 55 seconds! From then on, the GC pauses varies a lot, but rarely come back to zero. Most of the pauses are of several seconds.

3.2. Investigations



(a) Cpu idle usage (b) Number of anonymous THPs (c) Number of split THPs

Fig. 2. Cpu idle usage (a), number of anonymous THPs (b), and number of split THPs (c).

We observe that the actions of other applications can severely impact the performance of a Java application. Since the total physical memory is only about 70 GB, while the two applications together request that amount of memory, our first observation is that the system is under memory pressure. We found that there are quite a lot of swapping activities going on.

Though swapping activities affect GC pauses, we suspect that they alone are unable to explain the excessive pauses we see (i.e. 55 seconds). Our suspicion is justified by our examination of the GC pauses, many of which show high sys time. In Fig. 2(a) We observed that the system is also under severe cpu pressure. The high cpu usage cannot be entirely attributed to swapping activities as swapping typically is not cpu intensive. There must be other activities contributing to the cpu usage. We examined various system performance statistics, and identified a particular Linux mechanism of THP that significantly exacerbates the performance degradation.

With THP enabled, when Linux applications allocate memory, preference is given to allocations of transparent of huge pages of 2 MB size rather than the regular pages of 4 KB. We can verify the allocation of transparent huge pages in Figure 2(b), which shows the instantaneous number of anonymous transparent huge pages. Since THPs are only allocated to anonymous memory, the figure practically shows the total T HPs of the system. At the peak we see about 34K THPs, or about 68 GB.

We also observed that the number of THPs begins to drop after a while. This is because some of the THPs are split in response to low available memory. When system is under memory pressure, it will split the THPs into regular pages to be prepared for swapping. The reason for splitting is because currently Linux only supports swapping regular pages. The number of splitting activities are seen in Figure 2(c). We see that during the five minutes, about 5K THPs are split, which corresponds to 10 GB of memory.

At the same time, Linux attempts to collapse regular pages into THPs, which requires page scanning and consumes cpu. There are two ways to collapse: background collapsing and direct collapsing. Background collapsing is performed by khugepaged daemon. We occasionally observed that the khugepaged daemon tops cpu usage. Direct collapsing by applications trapping into kernel and allocating huge pages has even more severe consequences in terms of performance penalty, which can be seen by direct page scanning count.

An even more troublesome scenario THP can ran into is that the two contradicting activities of compacting and splitting are performed back and forth. When system is under memory pressure, THPs are split into regular pages, while a short while later regular pages are compacted into THPs, and so on and forth. We have observed such behaviors severely hurting application performance in our production systems.

3.3. Solution

```
Variables:
1  Th_freemem: Threshold of free memory;
2  Th_dd: Threshold of direct page scanning;
3  Th_cpu: Threshold of khugepaged cpu usage percentage;

Algorithm
1  Every time of T (Adjustment period):
2      Perform system resource monitoring (e.g. free mem);
3      If free memory > Th_freemem:
4          Enable THP;
5      Else:
6          Disable THP;
7          Continue;
8      If average direct page scanning rate > Th_dd:
9          Disable direct THP collapsing;
10     Else:
11         Enable direct THP collapsing;
12     If khugepaged average cpu usage > Th_cpu:
13         Disable background THP collapsing;
14     Else:
15         Enable background THP collapsing;
```

Fig. 3. Main algorithm of scenario I.

We now present solutions to prevent Java applications from performance degradation in this scenario. The key idea of our solution is to dynamically tune THP. Though we have seen that enabling THP feature could cause critical performance penalty, THP provides performance gains in other scenarios. The bottom line is to enable THP when it can bring benefits, while disable it when it could cause troubles. The first observation we make is that THP exposes performance penalty mostly when the system's available memory is low. When that happens, existing THPs need to be split into regular pages for swapping out. Thus, it is better to disable THP entirely when the system is under memory pressure. Since Java applications allocate heap when started (particularly when started with " -XX:+AlwaysPreTouch"), it is important to decide on whether to allocate THPs to a Java application when started. Thus, we choose to use the memory footprint size of the Java application as the memory threshold to decide whether to turn on or off THP. When the available memory is significantly larger than the application's memory footprint size, then THP is enabled, as the system is unlikely under memory pressure after launching the particular application. Otherwise, THP is disabled. Since many dedicated backend platforms like LinkedIn's are hosting homogenous applications, assigning applications' footprint size is a simple while effective decision.

Moreover, regular pages need to be collapsed into THPs before the huge pages can be allocated to applications. Thus, part of the element is to decide when to allow THP collapsing. We propose to base the decision on the direct page scanning rate and the cpu usage of khugepaged.

In summary, the design element consists of 3 components which control different collapsing types separately: (1) When available memory drops below the threshold, disable THP; (2) When direct page scanning is high, disable direct THP collapsing; and (3) when khugepaged daemon appears to be a cpu hogger, disable background THP collapsing.

3.4. Algorithm

After Java applications are started, every T time (adjustment period), the algorithm finely tunes THP.

Firstly, the algorithm obtains current system performance statistics including free memory size. It then decides to enable or disable THP, as shown in Fig. 3 Algorithm. Note that once THP is turned off, the algorithm simply skips the following steps as finer knobs are disabled inside THP.

The algorithm then checks whether it should turn on or off the background and direct THP collapsing independently. For direct THP collapsing, it relies on the past period's statistics of direct page scanning. If it appears to be having heavy direct page scanning activities, it will turn off the knob. Otherwise, direct THP collapsing is enabled. Similarly, for background THP collapsing, it relies on the activities of khugepaged as shown in cpu monitoring utilities such as top. If khugepaged appears to be hogging cpu, that knob is turned off. Otherwise, the knob is turned on. The tuning of THP is shown in Fig. 3 Algorithm.

3.5. Evaluation

Table 1. Javaapp Throughput (K Allocations/SEC)

Mechanisms	THP Off	THP On	Dynamic THP
JavaApp I	12	15	15
JavaApp II	13	11	12

We now present the evaluation results. We use the same JavaApp as described before, and also use the BackgroundApp for creating deployment scenarios with different available memory sizes.

We consider the scenario where Java Apps are started with abundant available memory and less-abundant available memory. Specifically, the first JavaApp is started without other applications running. It then runs for 3 minutes and stops. After that, a BackgroundApp takes 45GB of memory, then another JavaApp is started and runs for 3 minutes. Note that for these runs, the first two design elements are both enabled.

For the above scenario, 3 mechanisms are considered: THP is turned off, THP is turned on, and THP is dynamically tuned. The adjustment period is set to be 2 seconds. The results are shown in Table I. We see that when THP is off, JavaApp-I achieves the lowest throughput of 12 K/s, while the other two mechanisms have THP enabled and hence see higher throughput of 15 K/s.

For JavaApp-II, since it is running under memory pressure, turning THP off gives the highest throughput. Dynamically tuning THP results in less throughput (12K/s) than turning-off THP, but outperforms THP-on since it turns off THP when necessary. Note that dynamically tuning THP brings the benefit of accommodating more scenarios, particularly in scenarios where the system usage is unpredictable thus manual configuration of THP is not desirable.

We also notice that for JavaApp II, the performance benefit brought by Dynamic THP when compared to THP-ON is not significant (i.e. 1K/s), that is because of the relatively simple scenario we considered and hence less performance improvement. However, in other scenarios as well as in real productive environments, we have observed much more significant improvement by turning off THP appropriately. Thus, we believe the algorithm used by Dynamic THP needs to be finely tuned, which will be our future work.

4. Scenario II: Application Steady State with Heavy IO

In this scenario, the Java application has entered steady state since being started, but OS has heavy IO (i.e., disk read/write). Let's start with a real production issue we experienced at LinkedIn. The production issue is about very large JVM STW pauses, which we can easily reproduce in lab environments using custom-built Java workload and background IO traffic.

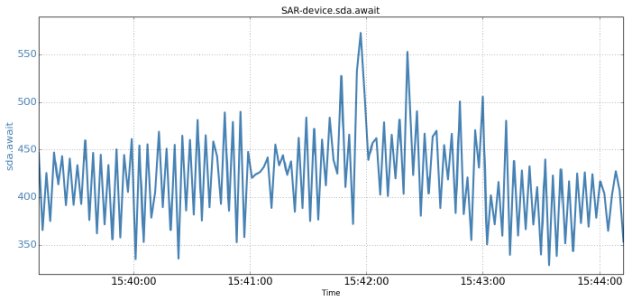
```
2016-01-14T22:08:28.028+0000: 312052.604: [GC (Allocation Failure) 312064.042: [ParNew
Desired survivor size 1998848 bytes, new threshold 15 (max 15)
- age 1: 1678056 bytes, 1678056 total
: 508096K->3782K(508096K), 0.0142796 secs] 1336653K->835675K(4190400K), 11.4521443 secs]
 [Times: user=0.18 sys=0.01, real=11.45 secs]
2016-01-14T22:08:39.481+0000: 312064.058: Total time for which application threads were stopped: 11.4566012 secon
```

Fig. 4. A large STW pause (11.45 seconds) that is not caused by the application itself.

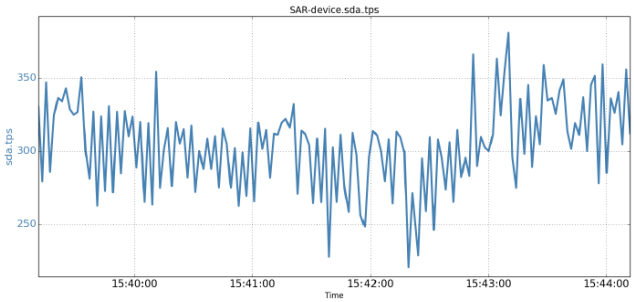
4.1. Production Issue

The Java heap space which JVM manages is GC-ed when the heap is almost full. JVM could be stopped during certain GC and JVM activities (e.g., Young or Full GC), which introduce STW pauses to the applications. Depending on JVM options supplied when starting the Java applications, various types of GC and JVM activities are logged into GC log files.

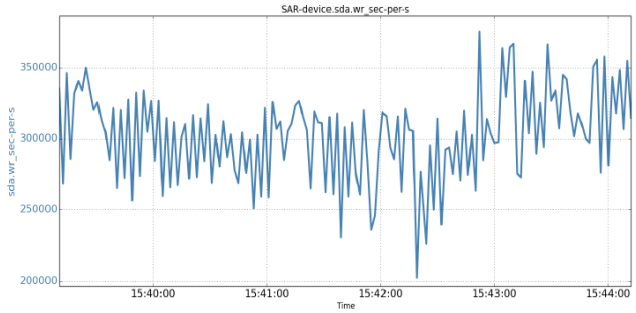
In our production environments, we have been seeing unexplainable large STW pauses (>5 seconds) happened to our mission-critical Java applications. Such STW pauses cannot be explained by application-level activities and the garbage collection activities during GC. In Figure 4 we show a large STW pause of more than 11 seconds and some GC information. With a mere 4GB heap size, the garbage collection typically takes sub-second to complete, and the simple GC log options incurs little overhead, however the application threads stopped for more than 11 seconds. The user and sys time are both negligible, hence the amount of work done by GC (e.g., collected heap size) is not able to explain the large pause value.



(a) “await” (Aver. time of I/O requests issued to the device to be served, in ms)



(b) “tps” (Number of IO transfers issued to the device per second).



(c) “wr sec-per-s” (Number of sectors written per sec, each sector is 512 bytes)

Fig. 5. Disk IO statistics reported by "sar -d -p 2" Linux utility (IO is caused by Background IO workload)

4.2. Reproducing the Problem in Lab Environments

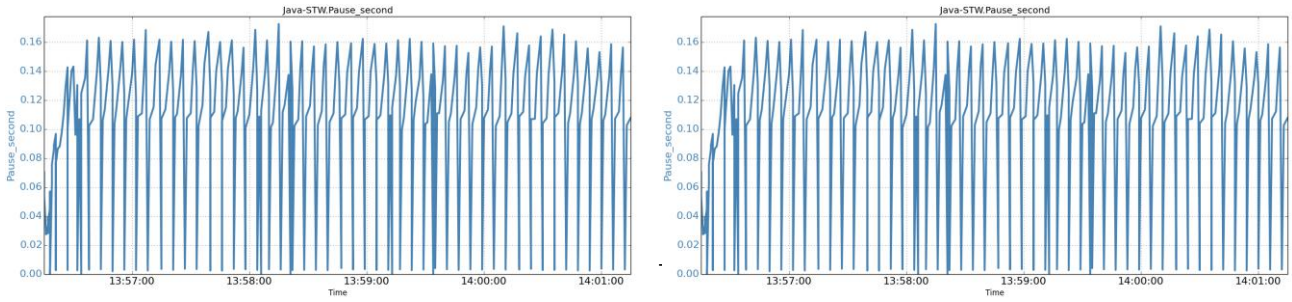
Our investigation efforts start with reproducing the problem of unexplainable large JVM pauses in lab environments. For reasons of controllability and repeatability, in this writing we use a simple workload which removes the complexity of the production applications. We run the workload in two scenarios: with

and without background IO activities. The scenario where no background IO exists is treated as the baseline, whereas the other scenario with introduced background IO is to reproduce the problem.

The Java workload we use has the following simple logic. It keeps allocating objects of certain size (denoted by object_size) to a queue. Whenever the number of objects reaches a threshold (denoted by count_down_size), a portion of the objects (denoted by removal_size) are removed from the queue. In all our following experiments, object size is 10KB, count_down_size is 100K, and removal size is 50K. So the maximum number of objects in the heap is 100K objects, which is about 1GB raw size. This process continues for fixed amount of time (e.g., 5 minutes). The source codes of the Java workload and the below background IO script are open-sourced on github [9], where the Java workload is Test.java, and the background IO code is background.load.sh.

Since this work focuses on the JVM STW pauses, the main performance metrics we consider are the application STW pauses; we specifically consider: total STW pause counts, total pause duration and numbers of big pauses.

Background IO Workload: The background IO [9] is produced by a bash script which repeatedly copies big files. In our lab environment, the background workload is able to generate 150MB/s writing load, sufficient to saturate the mirror ed hard drives equipped to the machine. To gain an idea of how heavy the generated IO load is, in Fig. 5 we show a 5-minute outputs of "sar -d -p 2", which includes await, tps, and wr sec-per-s values. The average values of them are: await=421 ms, tps=305, wr sec-per-s=302K.



Scenario A (W/o background IO load) Scenario B (With background IO load)

Fig. 6. JVM STW pauses in two scenarios.

System setup: Below we show the detailed hardware/software setup for the experiments we conducted in this writing. The platform is a HP Z620 Workstation with 1 socket of 12 Intel(R) Xeon(R) CPU E5-2620 2GHz hardware cpus. OS is RHEL Linux 2.6.32-504.el6.x86 64. The storage is a mirrored setup consisting of two SEAGATE ST3450857SS disks, SAS-connected. File system is EXT4, with default mounting options. We use Oracle HotSpot JDK-1 8 0 5. The JVM options are: -Xmx10g -Xms10g -XX:+UseG1GC -Xloggc:gc.log -XX:+PrintGCDateStamps -XX:+PrintGCTimeStamps - XX:+PrintGCApplicationStoppedTime.

Scenario-A (Without background IO load): The first scenario we present is treated as a baseline scenari o, where the Java workload runs without background IO load. We perform many runs in lab environments and the results are consistent. In the following we show a run which lasts for 5 minutes. In Figure 6(a) the time series data of all JVM STW pauses are shown along the 5-minute time line.

We also consider the following statistics values about STW pauses for the 5-minute run: (1) Total pause duration, i.e., the aggregated pause time for all STW pauses; and (2) Counts of large STW pauses (i.e., more than 2/1/0.5/0.25 seconds). We observe that all pauses are very minor, and no STW pause exceeds 0.25 second. The total STW pauses is about 32.8 seconds.

Scenario-B (With background IO load): The second scenario is running the same Java workload with the background IO load, which represents a production environment where there are significant IO load

generated. The IO load can come from the OS, other applications on the same node, or the IO activities by the same Java application that experiences the large JVM STW pauses. In Figure 6(b) the time series data of all JVM STW pauses are shown along the 5-minute time line.

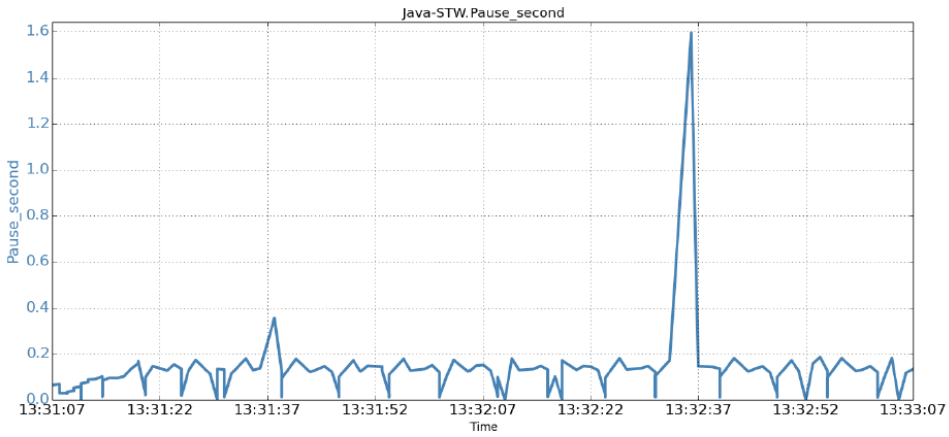
When the background IO runs, the same Java workload has seen 1 STW pause exceeding 3.6 second, and 3 pauses exceeding 0.5 seconds during a mere 5-minute run! As a result, the total STW pause time is 36.8 seconds, 12% more than Scenario-I. These results are shown in Table II.

Table 2. Statistics of JVM STW Pauses for 5-Minute Run in Scenario II

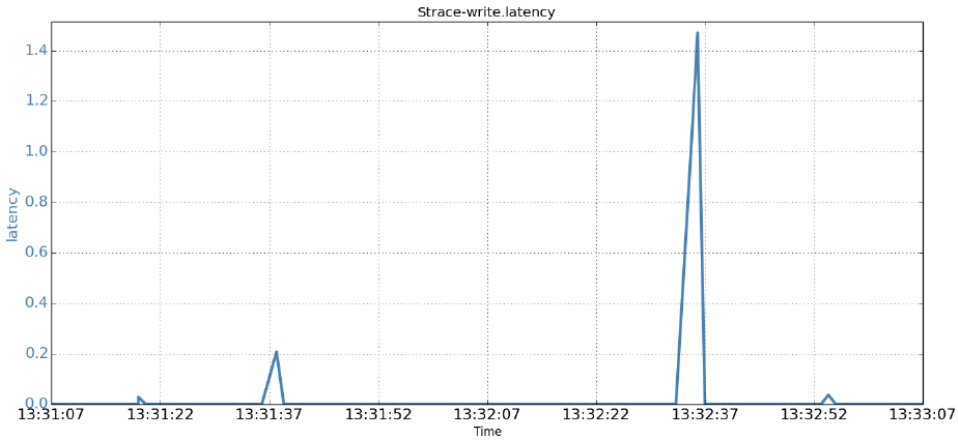
Metric	Total pause	>2s	>1s	>0.5s	>0.25s
Value	36.8 (second)	1	1	3	4

4.3. Investigations

After ruling out all other possible reasons, our investigations turned to JVM internals. Since JVM works by invoking system calls [10] just like other applications, we tried to understand what system calls that cause the STW pauses. We used strace [11], a tool for tracing system calls, to profile the Java work load. We found that several write() system calls have unusually large execution time.



(a) JVM STW pauses



(b) Strace-reported write() system call latencies

Fig. 7. Time series correlation between JVM pauses and write() system call latencies reported by strace utility.

Examining the time stamps of the system calls and the JVM pauses, we found that they correlate well. In

Fig. 7, we plot the time series of the two latencies. Such apparent correlations suggest that the large STW pauses are caused by the large write() call latencies.

gc.log:

1 2015-12-21T13:32:33.736-0800: 86.268: Total time for which application threads were stopped: 0.1714770 seconds
2 2015-12-21T13:32:35.047-0800: 87.578: [GC pause (G1 Evacuation Pause) (young) 7814M->4114M(10G), 0.1227713 se
3 2015-12-21T13:32:36.641-0800: 89.172: Total time for which application threads were stopped: 1.5946271 seconds

strace output:

4 [pid 11797] 13:32:35.169922 write(3, "2015-12-21T13:32:33.736-0800: 86"..., 224 <unfinished ...>
5 [pid 11797] 13:32:36.640856 <... write resumed>) = 224 <1.470906>

Fig. 8. Correlating gc log and strace output for a large JVM STW pause.

Though JVM GC logging uses buffered writes, the correlations between GC pauses and write() latencies suggest that the buffered writes of GC logging are still blocked due to some reasons. We examined the two time series data carefully, in Figure 8 we show one snapshot of a large JVM STW pause of 1.59 seconds. Let's explain the data. At time 35.04 (line 1), (For consistency, we use Linux system time (as opposed to JVM time); and for simplicity, we limit the time stamp to 2 decimal digits) a young GC starts and takes 0.12 seconds to complete. The young GC finishes at time 35.17 and JVM tries to output the young GC statistics to gc log file by issuing a write() system c all (line 4). The write() call is blocked for 1.47 seconds and finally finishes at time 36.64 (line 5), taking 1.47 seconds. W hen write() call returns at 36.64 to JVM, JVM records this STW pause of 1.59 seconds (i.e., 0.12 + 1.47) (line 3). In other words, the actual STW pause time consists of two parts: (1) GC time (e.g., young GC) and (2) GC logging time (e.g., write() time).

These data suggest that GC logging process is on the JVM's STW pause path, and the time taken for logging is part of STW pause. If the logging (i.e., write() calls) is blocked, the blocking time contributes to the STW pause. The new question is why buffered writes are blocked? Digging into various resources including the kernel source code, we realize that buffered writes could be stuck in kernel code. There are multiple reasons including: (1) stable page write; and (2) journal committing.

Stable page write: JVM writing to GC log files firstly dirties the corresponding fi le cache pages. Even though the cache pages are later persisted to disk files via OS's writeback mechanism, dirtying the cache pages in memory is still subject to page contention caused by stable page write. With stable page write, if a page is under OS writeback, write() to this page has to wait for the writeback completion. This is to ensure data consistency by avoiding partially fresh page being persisted to disk.

Journal committing: For journaling file system, appropriate journals are genera ted during file writing. When appending to the gc log file results in new blocks being allocated, file system needs to commit the journal data to disk first. During journal committing, if the OS has other IO activities, the commitment might need to wait. If the background IO activities are heavy, the waiting time can be noticeably long. Note that EXT4 file system has a feature of delayed allocation which postpones certain journal data to OS writeback time, which alleviates this problem. Note also that changing EXT4's data mode from the default “ordered” mode to “writeback” does not really address this cause, as the journal needs to be persisted before write-to-extend write() call returns.

Background IO activities: From the standpoint of a particular JVM garbage collection, background IO activities are inevitable in typical production environments. There are several sources of such IO activities: (1) OS activity; (2) administration and housekeeping software; (3) other co-located applications; (4) IO of the same JVM instance. First, OS contains many mechanisms (e.g., “/proc” file system [12]) that incur data writing to underlying disks.

Enterprise machines oftentimes feature softwares that facilitate administrations. For instance, machines

with CFEngine [13] installed needs to periodically download configuration files from other nodes, and the configuration files are persisted to local disks. Depending on software type and the scale, these administration software may incur significant (and burst y) IO loads.

Oftentimes an enterprise node may have other co-located applications that also incur disk IO. These co-located applications may or may not be identical JVM instances. Co-locating applications can help achieve better business cost efficiency by sharing the same computing resources (e.g., hardware).

Finally, the particular Java application may have other types of logging (e.g., access logs as commonly seen in web servers) and log rotations (e.g., compressing log files and relocate them). Depending on application type and traffic volume, the IO traffic incurred can be quite heavy, as we observed with our products.

4.4. Solution

We have seen that due to OS mechanisms (i.e., page cache writeback, journaling file system), JVM could be blocked for large periods during GC logging due to the internal page contention with other IO activities. Despite the dominance of HotSpot JVM adoption, we expect this problem should also occur to other JVM implementations that log information to files. Based on incomplete information, there are 80+ JVM implementations already. For instance, we observe the same issue happened to OpenJDK [14].

There are various approaches that can help mitigate this problem. In the following we list 4 types of possible approaches: (1) Enhancing JVM; (2) Reducing background IO; (3) Improving application IO; (4) Separating GC logging from other IO; These approaches differ in the degree of effectiveness and the amount of adoption effort. The three approaches of enhancing JVM, separating GC logging from other IO, and reducing background IO are application-transparent, meaning no application change is needed. The only approach that requires application change is the approach of improving application IO.

Enhancing JVM: Firstly, the JVM implementation could be enhanced to completely address this issue. Particularly, if the GC logging activities are separated from the critical JVM GC processes that cause STW pauses, then the corresponding problem which is caused by GC logging blocking will go away. For instance, JVM can put the GC logging into a different thread which handles the log file writing independently, hen ce not contributing to the STW pauses. Taking the separate-thread approach however risks losing last GC log information during JVM crash. It might make sense to expose a JVM flag allowing users to specify their preference.

Though a JVM-side enhancement is a clean and preferred solution, it is in the hands of the JDK provider (e.g., Oracle) and out of most users' (e.g., LinkedIn) control. Hence we present this only as a reference point and will propose this to JDK provider as an offline effort, which is beyond the scope of this work.

Reducing background IO: Since the extent of STW pauses caused by background IO depends on how heavy the background IO is, various ways to reduce the background IO intensity can be applied. Specifically, if the IO coming from administration housekeeping software is heavy, we can move to less IO-intensive software. Also, de-allocating other IO-intensive applications on the same node will certainly reduce the background IO.

Improving application IO: For the particular JVM instance that suffers from large STW pauses, we should try to reduce the intensity of other types of logging. One particular IO reduction about this is improving on the log rotation. Enterprise applications that serve external requests (e.g., frontend users or backend service requests) normally have a log rotation mechanism to persist the service access logs. The straightforward and popular logging rotation mechanism simply keeps outputing raw access logs, and another process periodically compresses the raw logs files. This mechanism could incur heavy and bursty IO writing (i.e., one for continuous raw log writing, the other for bursty compressed log writing).

To reduce the amount IO incurred in the straightforward log rotation process, we propose a “compressed-logging” approach, which only logs “compressed” access logs. With this approac h, in stead of

writing both raw and compressed files to disk as in the previous approach, it only writes the compressed logs. Since the raw access logs typically can result in very high compression ratio (e.g., 20X), the write IO saving is huge. In addition, because the output log files are already compressed, the compression part of log rotation is no longer needed. A less obvious benefit is the saving on the reading of raw logs during log rotation.

In some production environments, the raw logs are also used by other processes (e.g., log mining), compressed-logging may violate the existing workflow. Addressing this problem i s rather simple - it is not hard to modify the other workflows to adapt to compressed-logging (e.g., by reading/de-coding the compressed logs).

Separating GC logging from other IO: For latency-sensitive applications such as online ones servicing interactive users, large STW pauses (e.g., >250ms) typically are intolerable. Based on our experiences, ensuring < 200ms pauses is a mandatory for these applications.

To avoid large latency STW pauses induced by OS, various methods can be applied, and the bottom line is to avoid gc logging being blocked by OS IO activities. A straightforward approach is to put gc log file on another file system which is different from other OS IO activities. One example file system is tmpfs [15], a memory-based file system. Tmpfs has the advantage of very low writing latency since it does not incur actual disk writing. By putting the JVM GC log file on tmpfs (i.e., -Xloggc:/tmpfs/gc.log), JVM logging will not blocked by other IO that write to disk files.

There is a persistency problem with tmpfs-based approach, however. Since tmpfs does not have backup disk, during system crush the GC log file will be lost. A remedy to this is to periodically backup the log file to persistent storage to reduce the amount of the loss.

Another approach is to put gc log file on faster disks such as SS D (Solid State Disks) [16], which typically has much better IO performance in terms of writing latency and tps (transactions per second). Depending on the IO load, SSD can be adopted as a dedicated drive for gc logging, or shared with other IO loads.

The disadvantage of SSD-approach is the higher cost associated with SSD when compared to its HDD counterpart. Cost-wise, rather than using SSD, a more cost-effective approach is to put gc log file on a dedicated HDD. With only the IO activity being the gc logging, the dedicated HDD should be able to meet the low-pause JVM performance goal. In fact, the Scenario-I we show in Section ?? can mimic such a setup, since in that setup no other IO activities exist on the gc-logging drive.

4.5. Evaluation

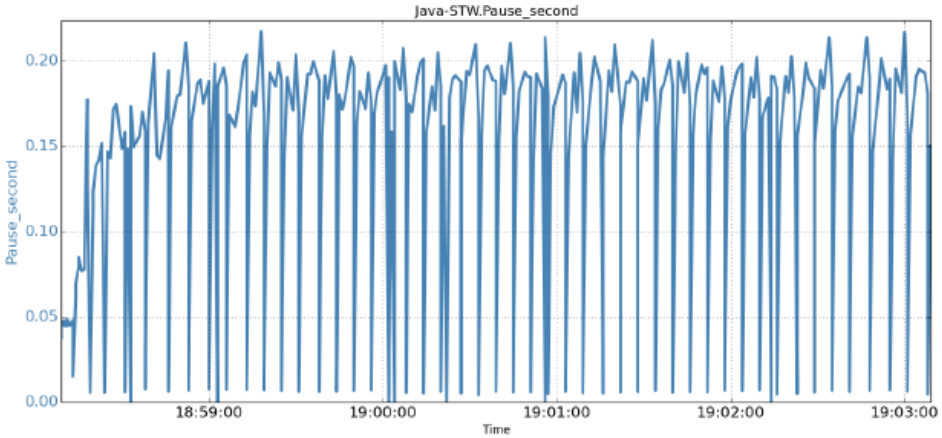


Fig. 9. JVM pauses when GC logging to SSD (With b/g IO load).

For the four types of approaches we proposed, the first approach of enhancing JVM cannot be evaluated

at this time. The second approach is self-explanatory, that is, if there is no (or less) background IO, then the type of STW pauses we consider in this work will disappear (or become less severe). Hence, in this work, we focus on evaluating the other two approaches of: (1) Separating GC logging from other IO; and (2) Improving application IO.

All the performance runs we conducted in this section use the same lab environment as we described before unless explicitly mentioned otherwise.

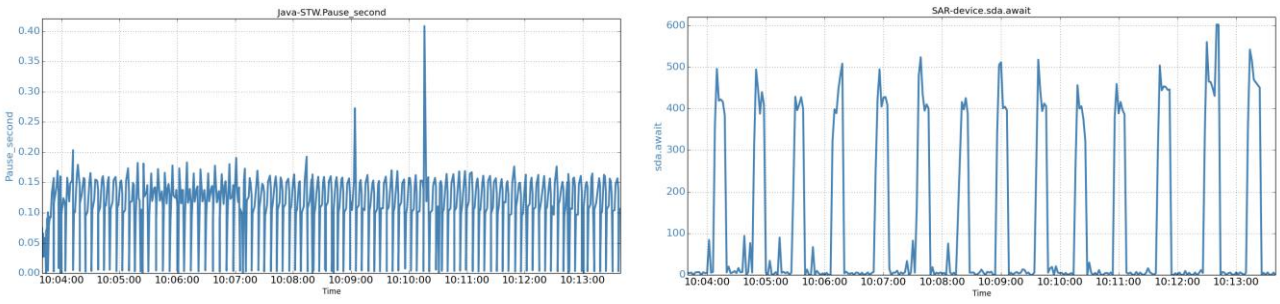
1) Separating GC logging from other IO: We evaluate this approach by putting the gc log file on a SSD file system. We run the same Java workload and the background IO load as in Scenario-B of Section IV-C. All STW pauses and the time stamps are shown in Figure 9 for the 5-minute run. No single big STW pause is observed.

For the statistics of the pauses, we notice that the JVM pausing performance are on-par with the Scenario-A, and all pauses are under 0.25 seconds (as opposed to the large pauses of 3.6 second seen in Scenario-B). The performance numbers are huge improvements, which indicate the background IO load does not cause large STW pauses to the JVM instance.

2) Improving application IO: To evaluate the approach of improving application IO, we introduce application logging to the Java workload. The raw logging rate is 40MB per second, and no other background IO is introduced.

The raw logging uses asynchronous writing, and the actual writing to disk is governed by OS's writeback mechanism. The OS we use sets the writeback period to 5 seconds (i.e, vm.dirty writeback centisecs=500), so every 5 seconds, the dirty pages corresponding to the log files will be flushed to disk. The log rotation process uses gzip utility to compress the raw logs, and the process is kicked off every 30 seconds (i.e., each kickoff compresses about 1.2GB of raw log data).

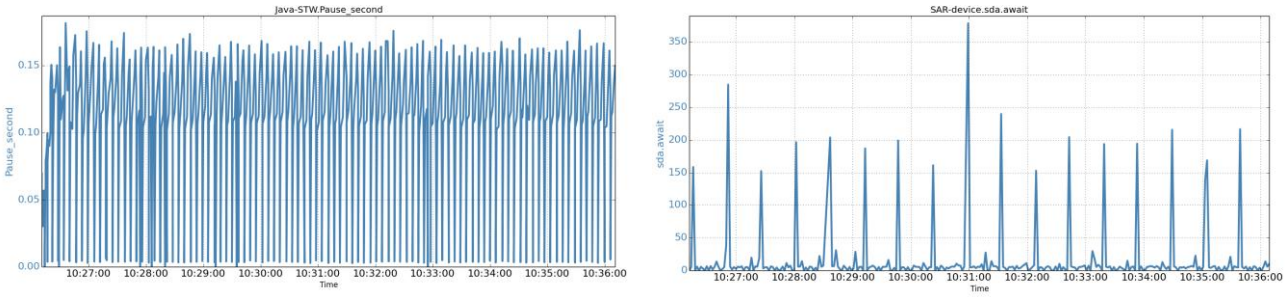
We now evaluate the proposed compressed-logging approach. We start with the default scenario where raw log files are written and traditional log rotation is invoked. The time series of JVM STW pauses and sar-reported disk await metric are shown in Figure 10, and the statistics of STW pauses are also aggregated in Table III. For this 10-minute run, we see that 2 larger-than-250ms pauses are observed. The performance is not as severe as when we invoke heavier background IO workload, but it still violates a 250ms-SLA (Service Level Agreements) twice in only 10 minutes.



(a) JVM STW pauses (b) Sar-reported disk IO "await"

Fig. 10. Time series correlation between JVM STW pauses and disk IO latency reported by sar utility (raw logging and log rotation).

We then adopt the compressed-logging approach by only logging compressed application logs. The time series of JVM STW pauses and sar-reported disk await metric are shown in Fig. 11. We observe no single STW pause is more than 250ms. In fact, the largest pause we see is 181ms.



(b) JVM STW pauses (b) Sar-reported disk IO "await"

Fig. 11. Time series correlation between JVM STW pauses and disk IO request latency reported by sar utility (Compressed-logging).

3) Summary: In this section, we evaluated the two approaches of Improving application IO and Separating GC logging from other IO. Based on the nature of the approaches and the performance results we obtained, we feel the latter approach of separating GC logging from other IO is preferred, both because of its easy adoption (i.e., no application code change, only needs to change -Xloggc option during JVM startup) and better performance.

Table 3. Statistics of JVM STW Pauses for 10-Minute Run (Raw Logging and Log Rotation)

Metric	Total pause	>2s	>1s	>0.5s	>0.25s
Value	64.8 (second)	0	0	0	2

5. Related Work

5.1. JVM Implementation and Tuning

JVM, the running environment for Java-based applications, has seen different embodiments each with certain design tradeoffs. Besides the widely deployed Oracle HotSpot [17], OpenJDK [14] and IBM J9, some companies design/implement their own JVM [18], [19].

With the increasing popularity of Java-based platforms, the performance tuning options also grow. There are all sorts of scenarios where Java applications can potentially perform better by various types of tunings. In addition, special tunings can be applied for specific scenarios where Java applications run [20], [21].

5.2. Linux and File Systems

Linux, being one of the mostly deployed Unix system, has complicated mechanisms which are explained in many books [22] and research articles. For applications dealing with data, file system component of the OS is critical. Many file systems [23], [24] are proposed, implemented and merged into Linux distributions.

Linux has long been the mostly deployed OS in server market [25]. Its key component of memory management has seen a plethora of advanced features being designed over the past years [26]. To accommodate the fact of limited RAM and requirement of supporting multiple processes, paging and swapping are continuously optimized to improve the system/application performance [6], [27], [28]. Meanwhile, to better fit a particular setup, various system and configuration optimization s are researched extensively [29].

5.3. Application Performance on Linux System

Many hands-on books [30], [31] present various performance tuning options for Linux system, and these tunings can be used as the reference points when observing application performance issues on Linux

system.

Performance study of various components of Linux are studied [32]. Application performance, particularly Java performance, can be affected by the interactions between different layers across the software stack. For instance, with the increasing adoption of multi-core, various studies have been done analyzing the performance scalability of applications running on multi-core platforms [33]–[35].

Bearing many advantages including platform-independence, Java is used as one of the top languages to build and run server applications. Ever since its birth, extensive works have been done to finely improve its performance in various deployment scenarios [36]–[38]. However, most of the performance studies focused on the Java/JVM itself. Our work, on the other hand, deals with an undesirable deployment scenario caused by system level resource shortages and multi-tenant interactions. Focusing on Java-based applications, our recent work [39] identified performance issues caused by JVM heap being swapped out by OS and THP, and we also proposed solutions to address the issues.

6. Conclusion

In this work, we studied the large JVM STW pauses faced by Java-based applications. We identified that the JVM and GC mechanisms, coupled with OS-level features, causes this issue. Based on our experiences with LinkedIn's platforms, we identify and solve a set of issues that causing large JVM pauses in two scenarios. We propose solutions that address this problem and share the knowledge we learned.

References

[1] Oracle java cloud service. Retrieved from: https://cloud.oracle.com/java

[2] Weblogic server. Retrieved from: https://www.oracle.com/middleware/weblogic/index.html.

[3] Java garbage collection basics. Retrieved from: http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

[4] Love, R. (2010). *Linux Kernel Development*. 3rd ed. Addison-Wesley Professional, 2010.

[5] Ext4 filesystem. Retrieved from: https://www.kernel.org/doc/Documentation/filesystems/ext4.txt

[6] Huge pages and transparent huge pages. Retrieved from: https://access.redhat.com/site/documentation/en-US/Red Hat Enterprise Linux/6/html/Performance Tuning Guide/s-memory-transhuge.html

[7] Talluri, M., Kong, S., Hill, M. D., & Patterson, D. A. (1992). Tradeoffs in supporting two page sizes. *Proceedings of the 19th Annual International Symposium on Computer Architecture.*

[8] Apache Log4j. Retrieved from: http://logging.apache.org/log4j/

[9] Workloads of java and background io. Retrieved from: https://github.com/zhenyun/JavaGCworkload

[10] Love, R. (2007). *Linux System Programming: Talking Directly to the Kernel and C Library*. O'Reilly Media, Inc.

[11] Strace - trace system calls and signals. Retrieved from: http://linux.die.net/man/1/strace

[12] Bishop, A. M. (1997). The /proc file system and procmeter. *Linux J*.

[13] Zamboni, D. (2012). *Learning CFEngine 3: Automated System Administration for Sites of Any Size*. O'Reilly Media, Inc.

[14] Openjdk. Retrieved from: http://openjdk.java.net/.

[15] Tmpfs file system. Retrieved from: https://www.kernel.org/doc/Documentation/filesystems/tmpfs.txt

[16] Enterprise ssds. Retrieved from: https://queue.acm.org/detail.cfm?id=1413263

[17] Java hotspot virtual machine. Retrieved from: http://www.oracle.com/technetwork/articles/javase/index-jsp-136373.html

[18] Tene, G., Iyengar, B., & Wolf, M. (2011). C4: The continuously concurrent compacting collector. *SIGPLAN Not.*, *46(11)*.

[19] Printezis, T. (20140. Use of the jvm at twitter: A bird's eye view. *SIGPLAN Not.*, *49(11)*.

[20] Chen, G., Shetty, R., Kandemir, M., Vijaykrishnan, N., Irwin, M. J., & Wolczko, M. (2002). Tuning garbage collection for reducing memory system energy in an embedded java environment. *ACM Trans. Embed. Comput. Syst.*, *1(1)*, 27–55.

[21] Hork´y, V., Libi˘c, P., Steinhauser, A., & T°uma, P. (2015). Dos and don'ts of conducting performance measurements in java. *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*.

[22] Corbet, J., Rubini, A., & Kroah-Hartman, G. (2005). *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc.

[23] Lu, L., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., & Lu, S. (2014). A study of linux file system evolution. *Trans. Storage*, *10(1)*.

[24] Jambor, M., Hruby, T., Taus, J., Krchak, K., & Holub, V. (2007). Implementation of a linux log-structured file system with a garbage collector. *SIGOPS Oper. Syst. Rev.*, *41(1)*.

[25] Usage share of operating systems. Retrieved from: http://en.wikipedia.org/wiki/Usage share of operating systems

[26] Bovet, D., & Cesati, M. (2005). *Understanding The Linux Kernel*. Oreilly & Associates Inc.

[27] Cervera, R., Cortes, T., & Becerra, Y. (1999). Improving application performance through swap compression. *Proceedings of the Annual Conference on USENIX Annual Technical Conference*.

[28] Oikawa, S. (2013). Integrating memory management with a file system on a non-volatile main memory system. *Proceedings of the 28th Annual ACM Symposium on Applied Computing*.

[29] Johnson, S. K., Hartner, B., Pulavarty, B., & Vianney, D. J. (2005). *Linux Server Performance Tuning*.

[30] Johnson, S. K., Hartner, B., Pulavarty, B., & Vianney, D. J. (2006). *Linux Server Performance Tuning*. Upper Saddle River, NJ, USA: Prentice Hall PTR.

[31] Ezolt, P. G. (2005). *Optimizing Linux(R) Performance: A Hands-On Guide to Linux(R) Performance Tools*.

[32] Bryant, R., Forester, R., & Hawkes, J. (2002). Filesystem performance and scalability in linux 2.4.17. *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*.

[33] Boyd-Wickizer, S., Clements, A. T., Mao, Y., Pesterev, A., Kaashoek, M. F., Morris, R., & Zeldovich, N. (2010). An analysis of linux scalability to many cores. *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*.

[34] Song, X., Chen, H., Chen, R., Wang, Y., & Zang, B. (2011). A case for scaling applications to many-core with os clustering. *Proceedings of the Sixth Conference on Computer Systems*.

[35] Chen, L., & Gao, G. R. (2010). Performance analysis of cooley-tukey fft algorithms for a many-core architecture. *Proceedings of the 2010 Spring Simulation Multiconference*.

[36] Hunt, C., & John, B. (2011). *Java Performance*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall Press.

[37] Taboada, G. L., Ramos, S., Exp´osito, R. R., Touri˜no, J., & Doallo, R. (2013). Java in the high performance computing arena: Research, practice and experience. *Sci. Comput. Program.*, *78(5)*.

[38] Kazi, I. H., Chen, H. H., Stanley, B. & Lilja, D. J. (2000). Techniques for obtaining high performance in java programs. *ACM Comput. Surv.*, *32(3)*, 213–240.

[39] Z. Zhuang, C. Tran, Ramachandra, H., & Sridharan, B. (2014). Ensuring high-performance of mission-critical java applications in multitenant cloud platforms. *Proceedings of the 2014 IEEE 7th International Conference on Cloud Computing (CLOUD)*.

Zhenyun Zhuang is received his Ph.D. and M.S. degrees in computer science from Georgia Institute of Technology. He also received his M.S. degree in computer science from the Tsinghua University and B.E. degree from in information engineering from Beijing University of Posts and Telecommunications. He has been actively conducting research in various areas including system and Java performance, wireless communications mobile networks, distributed systems, cloud computing and middleware platforms.

Cuong Tran currently is a staff engineer at LinkedIn. He worked at Yahoo and Symantec as a system architect. He also worked as an architect at Veritas Software and the director of technology at Adforce. With 10+ patents/papers/blogs, he is particular interested in the areas of system/application performance and Linux memory management and file systems.

Haricharan Ramachandra currently is an engineering manager at LinkedIn in LinkedIn. He received his MS in Computer Science from California State University-Chico and BS in Engineering from University of Mysore. He serves in the board of directors at Santa Clara Tech Academy. He also worked at Yahoo as an technical lead. His interest includes web/mobile performance, system performance and performance monitoring/analysis tools.

Badri Sridharan currently is a director of engineering at LinkedIn in LinkedIn. He received his MS in computer engineering from University of Wisconsin-Madison, and BE in electronics and communication from B.M.S College of Engineering. Leading the performance division, his interest covers all areas of system/application performance.