Performance Characteristics of Linux for Java Workloads Oversubscribing Memory

Takuya Nakaike Yohei Ueda Takanori Ueda Moriyoshi Ohara

Cloud computing can reduce the cost of IT services by maximizing resource sharing. Memory oversubscription is a technology to increase the degree of resource sharing by allowing applications to request a memory space beyond a given physical memory size. OS-based virtualization technologies such as Docker can increase it further compared to hypervisor-based virtualization technologies such as KVM, because the former runs each application in a container to share the whole operating system with other applications while the latter runs each application in a separate operating system. While the memory oversubscription of virtual machines on a hypervisor has been studied extensively, that on an operating system hasn't in a context of modern cloud services. This paper describes an analysis on how the current Linux virtual memory system affects the performance of Java applications that oversubscribe memory. Our major finding is that a major cause of performance degradation is page-cache reclamation operations which evict frequently-accessed memory pages mapped to a file rather than swapping out rarely-accessed anonymous pages. This paper also confirms that a Linux virtual memory system can be enhanced to improve the performance of Java workloads that oversubscribe memory through an experiment which forces several file pages to persist in a physical memory.

1 Introduction

Cloud computing can reduce the cost of IT services by letting a massive number of applications share a given hardware resources such as CPU and memory in a data center. Cloud service providers want to maximize the resource sharing to minimize the operational and capital cost for their data centers without degrading the performance of hosted applications significantly.

Oversubscription is a concept to increase the degree of resource sharing. It allows an application to request more resources than those available from the actual hardware. Memory oversubscription is attractive for cloud service providers because memory is one of the most expensive resources. It could, however, degrade the application performance drastically due to slow disk accesses to swap memory pages.

While there have been many studies on the memory oversubscription for a hypervisor [23] [17] [24] [13] [15] [5] [21] [1] [16] [20] [27], there are few studies on it for a modern operating system (OS), especially in a context of interactions between an OS and a language runtime. As container technologies such as Docker [11] are becoming popular as a lightweight virtualization mechanism, the OS-level memory oversubscription is becoming important since they rely on OS-level memory management.

Our work is to identify opportunities to improve the performance of a Linux virtual memory system for Java workloads that oversubscribe mem-

Linux 上でメモリ超過した Java ワークロードの性能特性

Takuya Nakaike, Yohei Ueda, Takanori Ueda, Moriyoshi Ohara, IBM 東京基礎研究所, IBM Research - Tokyo.

ory. Many researchers have already shown that swapping out memory pages belonging to a heap area of a Java virtual machine (JVM) degrades the performance significantly [2] [26] [4] [3] [25] [22]. Their common knowledge is that activities to touch objects placed on swapped-out memory pages, especially garbage collections (GCs), cause the performance degradation, and thus they focus on resizing the Java heap area to fit the total working set in a physical memory size. On the other hand, we focus on figuring out the detailed behavior of OS-level memory management to identify optimization opportunities for Linux.

We choose a Linux because it is one of the most widely used OS for cloud services. We ran two representative Java workloads directly on Linux without using containers to study the intrinsic characteristics of the Java workloads and the Linux virtual memory system.

We evaluated the performance characteristics of two benchmarks when multiple Java virtual machines (JVMs) running on the same node oversubscribing the memory. While the performance of both benchmarks degrades when memory pages are swapped out as expected, the root cause was not obvious. In fact, we identified that the performance degradation was due to page-cache (aka file cache) reclamation to read files from a disk because the OS evicts those pages when the memory pressure is high. While disk accesses to a swap area occurred, they were not the primary cause for the degradation.

We conducted an experiment to identify whether there is a room to enhance the Linux memory management system. We ran the benchmarks with pinning several file pages to a physical memory. In this experiment, the file-page pinning improved the throughputs of two benchmarks significantly. This result indicates that we can enhance the Linux virtual memory system to select victim pages that cause less performance impacts.

Here are our contributions:

- **Detailed behavior of a Linux virtual memory system that runs Java workloads oversubscribing memory**: We identified how a Linux virtual memory system manages memory pages when Java workloads oversubscribe memory.
- **Root cause for performance degradation in a memory-oversubscribed Linux OS**: We identified that page-cache reclamation is too aggressive compared to anonymous-page reclamation, resulting in the eviction of frequently-accessed file pages instead of swapping out rarely-accessed anonymous pages.
- **Experiment to identify an optimization opportunity for a Linux memory management system**: We identified how a Linux virtual memory system can be enhanced to improve the performance of applications when memory is oversubscribed.

Sections 2 and 3 overview the memory management systems of a Linux OS and a JVM, respectively. Section 4 describes our experimental environment. Section 5 shows our experimental results. Section 6 reviews related work. Section 7 concludes the paper.

2 Linux Memory Management

In this section, we describe the memory management system of a Linux OS [7] for the completeness of the discussion on our analysis results. Similar to other modern operating systems, the Linux OS integrates the management of memory pages for the page cache (aka file cache) and those for the user process memory in the virtual memory system. This integration has multiple benefits for the overall system performance. For example, under certain circumstances when a user process extensively accesses a large file but does not require a large amount of user process memory, the Linux OS utilizes the free physical memory for page caches, which substantially reduces file-access latencies. In contrast, when a user process starts accessing a large amount of memory and stops accessing the file, the OS reclaims the page caches for the file to increase its process memory. In this way, the OS can avoid time-consuming page-swap operations.

A Linux OS manages the physical and virtual memory through pages of two types: file and anonymous pages. The file page is a memory page mapped to a file and is kept on memory as a page cache for subsequent accesses to the file. A file can be mapped to file pages implicitly when it is read from a disk by using a read system call. In this case, the content of the file is first copied into file pages and then copied to a buffer specified by the read system call. A file can be also mapped to file pages explicitly by using a mmap system call. Such a file is called memory-mapped file. In this case, when a file is mapped to file pages, they are not present a page cache. When the content of the file is accessed, a page fault occurs to let a page-fault handler allocate one or more physical pages to fill them with the content of the file, which is read from a disk.

The anonymous page is, on the other hand, a memory page, not mapped to a specific file, which is typically allocated by a malloc function or another memory allocator in a language runtime. A malloc function internally uses a brk system call or a mmap system call with a MAP_ANONYMOUS option. Both calls allocate one or more zero-filled memory pages that are not associated to a specific file and can be used for any purposes.

A Linux OS keeps track of free physical pages to allocate one when a new page (file or anonymous) is requested. To maintain a certain number of free pages, the OS periodically checks the allocated pages and reclaims those that are not frequently accessed. When the OS needs to reclaim a file page, it first checks whether or not the page content has been changed since the content was copied from the file, and it writes back the content to the file if necessary. When the OS reclaims an anonymous page, it evicts the page content to a swap space on disk.

Since the algorithm to select a page to be reclaimed can affect the system performance, the OS needs to choose victim pages carefully. An important factor is the balance of victim selection between file and anonymous pages. Swappiness is a Linux kernel parameter that controls the balance. A low swappiness value discourages swapping of anonymous pages, while a high value encourages it. When the swappiness value is 0, its minimum value, the OS never reclaims anonymous pages unless the system almost runs out of memory. When the swappiness value is 100, its maximum value, on the other hand, the OS equally selects a victim from files and anonymous pages.

The victim selection algorithm in Linux is a variant of an approximate least recently used (LRU) algorithm, called split LRU. The OS manages four LRU lists of physical pages: active-file, inactive-file, active-anonymous, and inactive-anonymous pages. When it allocates a new file or anonymous page, it adds it at the head of the file-active or anonymous-active list, respectively. It also moves pages between active and inactive lists based on their access frequency, and selects a victim page from the tail of either inactive list.

The OS identifies inactive pages in an active list by checking the reference bit of each page. Modern processors have a hardware support to set the reference bit of a page automatically when the page is accessed. The kernel also sets the reference bit when it handles the read and write system calls.

The OS periodically scans the two active lists to find the pages to be moved to an inactive list. If the reference bit of a page is set, the OS considers the page is recently accessed and moves it to the head of the list. If the reference bit is unset at the tail of an active list, the OS moves the page to an inactive list.

The OS also scans the two inactive lists. When the OS finds a page with the reference bit set in an inactive list, it moves the page back to an active list. When the OS finds a page with a reference bit unset, it finally decides to evict the page. Each time after scanning a page, the OS clears its reference bit.

The swappiness value controls the scan frequencies of the file and anonymous lists. When the swappiness value is 0, the OS never scans the anonymous inactive list unless the system almost runs out of memory. When the swappiness value is 100, the OS scans the same number of pages in file and anonymous lists.

3 Java Memory Management

In this section, we present an overview of the memory management system of the IBM J9 JVM, which we used for our experiments. The memory space of a JVM consists of four subspaces [19] as described below.

Heap Area: This area stores Java objects. When a JVM starts up, the number of physical pages allocated for this area is minimal. As a JVM creates Java objects, it requests physical pages to the OS, hence the physical memory size increases. The size of this area can be specified explicitly by a command-line option (e.g. `-Xmx<heap_size>`).

When this area becomes full, a garbage collector discards dead (i.e., unreferenced) objects. After a GC, physical memory pages in this area can be freed and returned to an OS when there is no object in them. Two major GC algorithms are used in the IBM J9 JVM: mark-and-sweep and generational. One of the major differences between them is whether or not the heap area is divided. In the generational algorithm, the heap area is divided into nursery and tenure spaces. The nursery space is used to allocate new objects, while the tenure space is used to store long-lived objects, which are copied from the nursery space when their lifetime reaches a threshold. The separation of the two types of objects reduces the cost to detect live objects, while it increases the frequency of GCs because it cannot use the entire heap area for object allocation.

The generational algorithm involves two kinds of GC operations: local and global. A local GC is triggered to collect objects in a nursery space when it becomes full, while a global GC is triggered to collect objects both in nursery and tenure spaces when the tenure space becomes full. The IBM J9 JVM assigns one fourth of a given heap area to the nursery space unless a command-line option specifies its size explicitly.

The mark-and-sweep algorithm marks live objects by traversing objects from root references such as stack and static variables, and then discards (i.e. sweeps) unmarked objects. It traverses the entire heap area per GC. It is also used for global GCs in the generational algorithm.

Internal Area: This area is used mainly as a work area for the JIT compiler. This area expands when the compiler is active, and it does not disappear even when the compiler becomes inactive because it stores some persistent data such as profiling data.

Class Area: This area stores the metadata of each Java class such as the bytecode, the constant pool, and the method table, which is loaded from a Java class file. This area expands as the number of loaded classes increases.

JIT (Code Cache) Area: This area stores the binary code generated by the JIT compiler. It expands as the number of compiled methods increases. Because the IBM Testarossa JIT compiler uses an adaptive compilation strategy, where the compiler can recompile frequently-executed methods to apply high-level optimizations, the code that has been optimized at low-optimization levels can become useless. The space for the useless code is reclaimed for newly-generated code.

4 Experimental Environment

Figure 1 shows our experimental environment. Our benchmarks are DayTrader [6] and AcmeAir [9], which are Java-based Web applications to emulate an online stock trading system and an online air ticket booking system, respectively. We chose Web applications as benchmarks because they are one of the most popular applications in cloud. Each benchmark processes a sequence of requests from multiple clients by accessing a backend database system and then returns a response to each client. The two applications have different object lifetime characteristics, which allow us to investigate the interplay between the GC behavior and the performance. DayTrader has more long-lived objects than AcmeAir; this is because DayTrader stores transaction states, such as a client state, on the Java heap through an EJB framework, while AcmeAir stores them on the backend database through a REST API that supports stateless transactions.

We ran the two benchmarks on an IBM WebSphere Liberty application server [8] that runs on an IBM Java runtime environment. Our operating system is Red Hat Enterprise Linux, and its kernel version is 3.10.0. Our software stack is built on a 4-core 5.5 GHz zEC12 processor [14]. We limited the size of the physical memory to 2 GB intentionally to cause memory swapping. We used two hard disks, one exclusively for a swap space, to separate the disk-access activities for swap from the rest.

Total	**Nursery**	**Tenure**
1024 MB	256 MB	768 MB
512 MB	128 MB	384 MB
256 MB	64 MB	192 MB

Table 1: Java heap sizes for performance evaluation

We used DB2 [12] and MongoDB [11] as the backend database for DayTrader and AcmeAir, respectively. DayTrader sends an SQL query to the DB2 server to access a database record, while AcmeAir sends a JSON message to the MongoDB server. The application server and the rest of the system run on a separate system - a separate OS on a separate logical partition (LPAR) running on a separate set of physical processors.

We measured the performance by increasing the number of JVMs (one JVM per application server) with a fixed heap size for each JVM. We tuned the number of clients to maximize the throughput when one JVM is running an application with a given heap size. When we increased the number of JVMs, we divided the tuned number of clients to each JVM equally. Each measurement consists of the following three steps.

1. Warm up a given number of JVMs for five minutes to complete most of the JIT-compilation activities
2. Collect dead Java objects created in the warming-up phase by executing a System.gc() method
3. Continue to run the JVMs concurrently for one hour

Table 1 shows a set of heap size parameters that we used. We used the default nursery space, which is one fourth of the heap size as discussed in Section 3.

We used 100 as our default swappiness value, the maximum value, because it maximized the

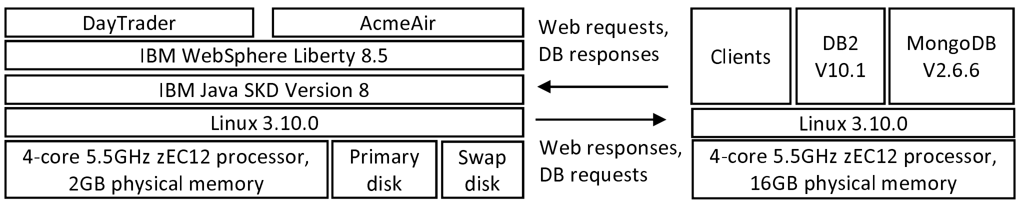

DayTrader
AcmeAir
IBM WebSphere Liberty 8.5
IBM Java SKD Version 8
Linux 3.10.0
4-core 5.5GHz zEC12 processor, 2GB physical memory
Primary disk
Swap disk
Web requests, DB responses
Web responses, DB requests
Clients
DB2 V10.1
MongoDB V2.6.6
Linux 3.10.0
4-core 5.5GHz zEC12 processor, 16GB physical memory

Figure 1: Experimental environment

throughput performance when memory swapping occurred. We used a generational GC algorithm as default except for the experiment in Section 5.3. Section 5.4 shows the performance effect for a smaller swappiness value. We obtained the following statistics by using vmstat and iostat commands.

- I/O-wait ratio: We obtained this statistics from the vmstat command directly. This statistics indicates the ratio of the CPU idle time when there is at least one outstanding I/O operation.
- Footprint: We calculated this statistics by using the following formula.

$$\begin{aligned}\text{Footprint} &= \text{Physical memory size}\\ &- \text{Free memory size}\\ &+ \text{Swapped-out memory size}\end{aligned}$$

  A free memory size and a swapped-out memory size are obtained from the vmstat command.
- Amount of disk accesses: We obtained this statistics from the iostat command.

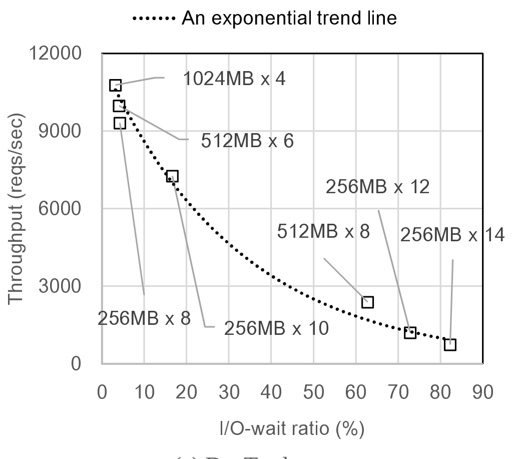
5 Experimental Results

5.1 Overall Performance of DayTrader and AcmeAir

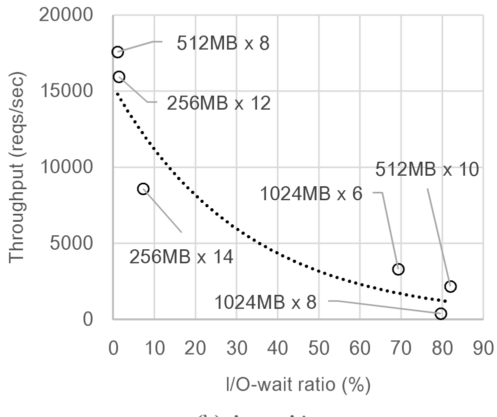
In this section, we look at the basic performance characteristics of DayTrader and AcmeAir. Figure 2 shows the relationship between the I/O-wait ratio and the aggregated throughput. Each dot corresponds to an experimental result with a configuration defined by the number of JVMs and the heap size for each JVM. We iterated the measurement for each configuration four times to compute their average. We excluded configurations with low memory pressure (i.e. the I/O-wait ratio is not higher than 1.0) because we want to focus on the workload characteristics when the memory pressure is high. We also drew a dotted line as an exponential trend for each benchmark.

A common characteristics of DayTrader and AcmeAir is that the throughput degrades as the I/O-wait ratio increases. For example, when we increased the number of JVMs (256 MB heap) from 10 to 12 for DayTrader, the I/O-wait ratio increased from 16% to 72% and the throughput degraded by 84%. Similarly, when we increased the number of JVMs (512 MB heap) from 8 to 10 for AcmeAir, the I/O-wait ratio increased from 1% to 81% and the throughput degraded by 88%. Figure 3 shows the relationship between the I/O-wait ratio and the average response time. The response time increases similarly as the I/O-wait ratio increases. We analyze the root cause of the increase in the I/O-wait ratio in the next section.

Figure 4 shows the relationship between the footprint and the response time. Note that memory swapping occurs when the footprint exceeds the 2 GB physical memory size. It is interesting to see that the two benchmarks exhibit very different characteristics when the footprint exceeds the physical size. As shown in the square dots, the response time of DayTrader is lower than 10 milliseconds even when the footprint is around 4000 MB. For example, the response time of DayTrader

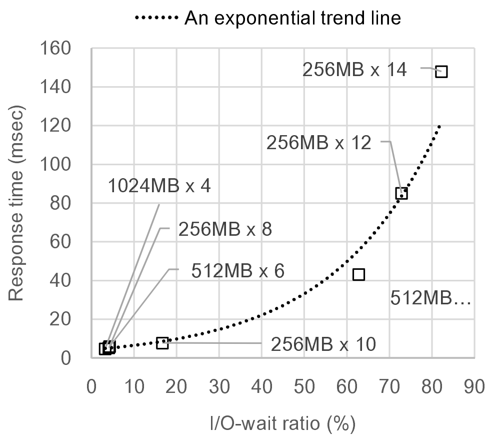


(a) DayTrader

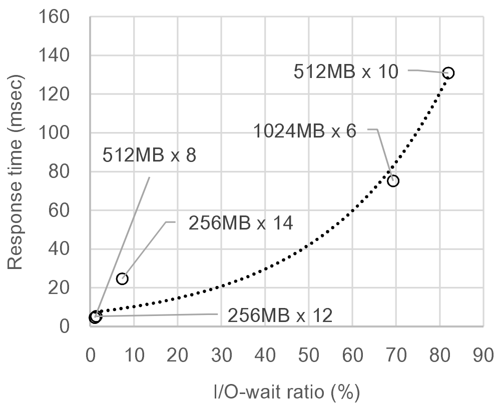


(b) AcmeAir

Figure 2: Relationship between the I/O-wait ratio and the aggregated throughput. Each dot corresponds to an experimental configuration defined by the number of JVMs and the heap size. We excluded configurations with low memory pressure for ease of viewing. We drew a dotted line as an exponential trend.



(a) DayTrader



(b) AcmeAir

Figure 3: Relationship between the I/O-wait ratio and the response time. Each dot corresponds to an experimental configuration defined by the number of JVMs and the heap size. We excluded configurations with low memory pressure for ease of viewing. We drew a dotted line as an exponential trend.

running on four JVMs (1024MB heap) is 4.9 milliseconds when the footprint is 3992 MB. As shown in the circle dots, on the other hand, the response time of AcmeAir is much longer than that of DayTrader even when the footprint is around 3000 MB. For example, the response time of AcmeAir running on 10 JVMs (512 MB heap) is 130.9 milliseconds when the footprint is 3170 MB. These results indicate that DayTrader can have more swapped-out memory pages than AcmeAir without causing destructive performance degradation.

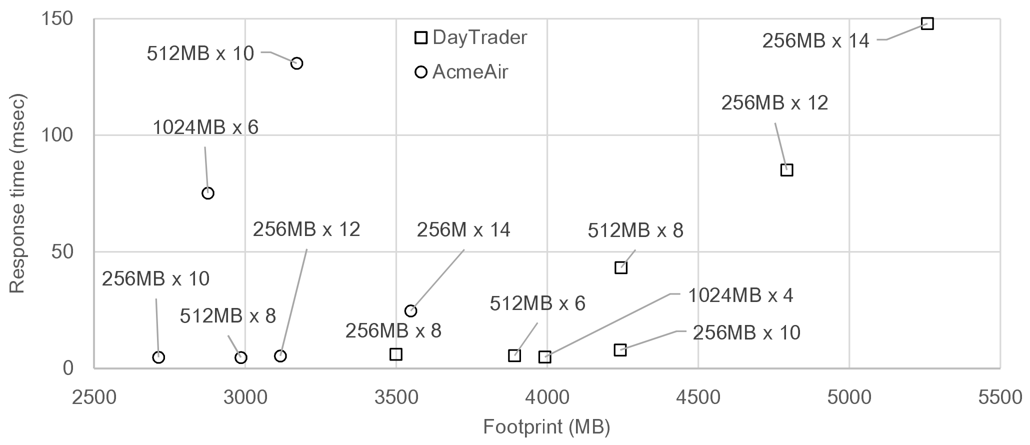


Figure 4: Relationship between the footprint and the response time. Each dot corresponds to an experimental configuration defined by the number of JVMs and the heap size. All of the configuration in this figure causes memory swapping because the footprints exceed the 2 GB physical memory size.

5.2 Deep Analysis of DayTrader

In this section, we discuss the Linux virtual-memory system algorithm evicts inappropriate pages to degrade the throughput performance of DayTrader when the memory is oversubscribed. More specifically, we identified that the algorithm tends to evict memory pages for the page cache more aggressively than anonymous memory pages, and that such evictions lead to a significant increase in the amount of the disk traffic to read files. In fact, by pinning all pages for the page cache to disable such evictions, we were able to improve the throughput performance significantly.

To simplify our analysis on the disk traffic, we configured two disks: one for a primary disk, and the other for a swap disk. The traffic for the primary disk consists only of regular file accesses, which fill a page cache with the file content, while that for the swap disk consists only of swap operations for the process memory. Figure 5 (a) shows the amount of disk accesses per request for the primary and swap disks when DayTrader runs on 10, 12, 14 JVMs. For all of the three experiments, the amount of the primary-disk traffic is larger than that of the swap-disk traffic. For example, when DayTrader runs on 12 JVMs (256MB heap each), while the size of the swapped-out memory is 2.8GB - much larger than the size of the whole physical memory, the amount of the primary-disk traffic (83KB per request) is much larger than that of the swap-disk (17KB per request). Most of the primary-disk accesses (more than 90%) were read.

To reduce the traffic of the primary disk, we pinned file pages accessed during DayTrader runs. A smaps file under the Linux proc file system indicated the following four types of memory-mapped files are accessed during DayTrader runs.

- Shared libraries in /usr/lib64
- Shared libraries in a Java runtime system
- Class-cache files of a Java runtime system
- Files that WebSphere Liberty uses to manage messages

As described in Section 2, a memory-mapped file is accessed through file pages which are kept in a page cache. By pinning such pages, we can prevent the virtual memory system from evicting them, and hence can avoid primary-disk accesses to read the file when the file is accessed later. To this end, we created a custom version of JVM, which parses a smaps file to identify the address of those file pages and pins them on the physical memory by using a mlock system call.

Figure 5 (b) shows the file-page pinning can make the traffic of the primary disk almost negligible without significantly affecting that of the swap disk. When DayTrader runs on 12 JVMs, as a result, the total amount of disk accesses were reduced by more than a factor of five, from 100KB per request to 18KB per request. This result indicates that the traffic for the primary disk mostly occurred to reclaim file pages, which had been evicted when the memory pressure is high. This means that there is a significant room to enhance the page eviction algorithm in the Linux virtual-memory system to select victim pages that impact the performance less, such as anonymous pages rather than file pages for memory-mapped files in our experiments. Figure 5 (c) further shows the file-page pinning improved the throughput significantly for all of the three configurations we examined.

Finally, we analyze which anonymous memory pages are swapped out by the Linux memory management system to keep files pages in a page cache when the file-page pinning is used. Figure 6 shows the breakdown of memory pages for DayTrader. We obtained the sizes of file-pages in a page cache, free pages, and swapped-out pages from a vmstat command. We further broke down the swapped-out and physical memory pages into five parts described in Section 3 respectively. To do it, we let each JVM generate a javacore file by sending a signal (`kill -3`) to the JVM process. The javacore file includes the address range of each memory area we discussed in Section 3. We implemented a tool to correlate this information with page-map entries that are provided through the Linux proc file system. Each page-map entry corresponds to a virtual address and has a flag to indicate whether the virtual page is present on the physical memory or is swapped out. We invoked the correlation tool after stopping all of the clients for DayTrader and before terminating JVM processes which are needed to keep their page-map entries.

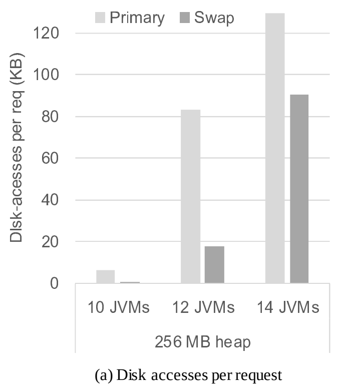
As shown in Figure 6, the Linux memory management system swaps out memory pages in the other category (rightmost bars) to keep the pinned files pages in a physical memory when the number of JVMs is twelve. This other category includes various types of memory pages such as the memory pages that are allocated outside of a Java heap area through Java native interface (JNI) calls. This indicates that the file-page pinning helped the Linux memory management system swap out such pages as rarely-accessed pages because the destructive performance degradation did not occur as shown in Figure 5.

When the number of JVMs is fourteen, the Linux memory management systems swaps out memory pages in the tenure and nursery spaces to keep the pinned file pages in a physical memory. Swapping out the memory pages for the tenure space is a good decision because they are rarely-accessed. In fact, the destructive performance degradation did not occur even when a large portion of the tenure space is swapped from 12 JVMs. However, swapping out the memory pages in the nursery space, which is accessed more frequently than the tenure space, degraded the performance significantly because of the increase in the amount of swap-disk accesses.

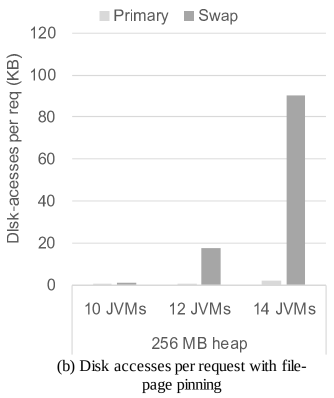
5.3 Deep Analysis of AcmeAir

In this section, we discuss whether the enhancement for the page-cache reclamation algorithm, which we mentioned in the previous section, is effective for AcmeAir.

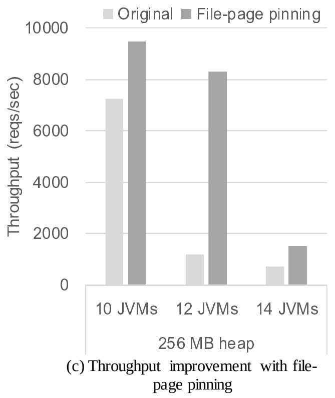
Figure 7 (a) shows the amount of disk accesses per request for a primary disk and a swap disk for



(a) Disk accesses per request



(b) Disk accesses per request with file-page pinning



(c) Throughput improvement with file-page pinning

Figure 5: Effect of file-page pinning for DayTrader. (a) the amount of disk traffic per request without file-page pinning. (b) the amount of disk traffic per request with file-page pinning. (c) the throughput with and without file-page pinning.

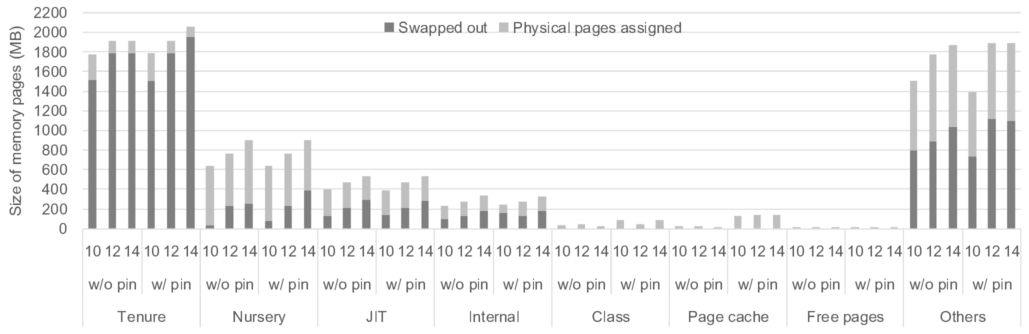


Figure 6: Breakdown of memory pages for DayTrader. We changed the number of JVMs with fixing the heap size to 256 MB.

AcmeAir. We choose the 512 MB heap for AcmeAir because using the 256 MB heap kept the amount of disk accesses small even when we ran fourteen JVMs.

When we ran 10 JVMs, the traffic of the primary disk was much larger than that of the swap disk. When we pinned file pages for the memory-mapped files mentioned in the previous section, most of the primary-disk accesses disappeared and the performance improved significantly as shown in Figure 7 (b) and (c). This indicates that the enhancement for the page-cache reclamation algorithm can be effective for AcmeAir.

A difference between DayTrader and AcmeAir is that the file-page pinning reduces the traffic of the swap disk in addition to the traffic of the primary disk. We suppose that the reduction in swapped-out memory pages for the nursery space brings this effect. Figure 8 shows the breakdown of memory pages for AcmeAir. As shown in this figure, the file-page pinning reduces the swapped-out memory pages for the nursery space when the number of JVMs is ten. Since the nursery space is accessed more frequently than the tenure space, the reduction in the swapped-out memory pages for the nursery space can reduce the traffic of the swap disk.

As shown in Figure 8, the Linux memory management system swaps out memory pages for the other category (rightmost bars) to keep the memory pages for the nursery space and file pages in a physical memory. This result is similar with the result in DayTrader.

A counter intuitive result is that the file-page pinning reduces the traffic of the swap disk without reducing the swapped-out memory pages for the nursery space when 12 JVMs run. We guess that this is a side effect of our tool which correlates the information about the JVM memory areas with page-map entries given from the Linux proc file system. As mentioned in the previous section, our correlation tool works with keeping JVM processes alive because the page-map entries are removed if we terminate the JVM processes. Therefore, our tool can swap out the memory pages that are resident in a physical memory during a benchmark is running. We consider that this side effect causes swapping out the memory pages for the nursery space.

5. 4 Effects of Swappiness

In this section, we show the impact of the swappiness value on the performance. Figure 9 shows the throughput of the two benchmarks when we changed the number of JVMs with a fixed 256MB heap size. We used two swappiness values: 30 (default) and 100 (maximum). Thirty is the default value on our Linux OS.

As shown in Figure 9, the default swappiness value causes a lower performance than the maximum one. As described in Section 2, the lower the swappiness is, the more likely the OS selects a victim from file pages than anonymous pages. As a result, when we reduce the swappiness from 100 to 30, the OS increases the amount of primary-disk accesses by reclaiming memory from page caches even though the JVM has many rarely accessed pages in the tenure space, which can be swapped out with a limited impact on the performance. For our benchmarks, the Linux OS did not cause a negative side

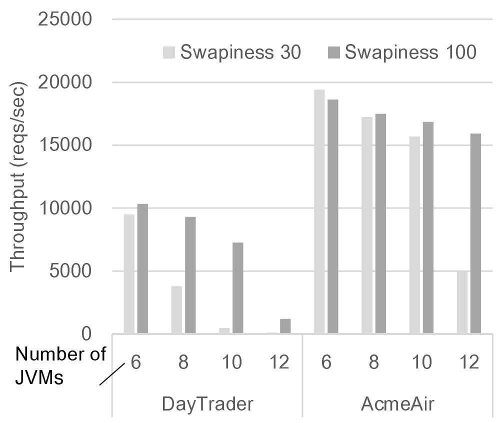


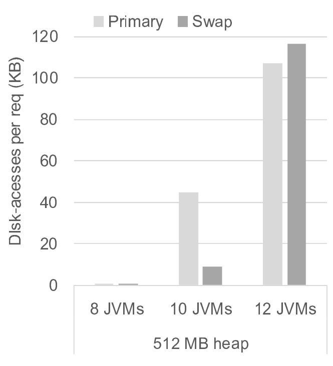
Figure 9: Throughputs of two benchmarks when we changed the number of JVMs with fixing the heap size to 256 MB. Two swappiness values are used. Reducing the swappiness value suppresses memory swapping.

effect, such as excessive memory swapping, with the maximum swappiness value, while it lost opportunities to swap out rarely accessed memory pages with the default swappiness value.

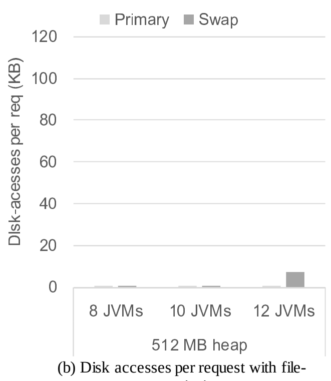
6 Related Work

Memory ballooning is a technique that a hypervisor uses to reduce the amount of accesses to swapped-out memory pages for its managing virtual machines (VMs) [23] [15] [27]. When a hypervisor lacks physical memory pages, it forces a guest OS to pin some physical memory pages in its VM and to pass the pages to other VMs. A benefit of this approach is to leave the memory reclamation to guest OSes by assuming that each guest OS reclaims memory more efficiently than the hypervisor by using its own page access information. Because our work focused on improving the OS-level memory management, it is orthogonal to the ballooning approach.

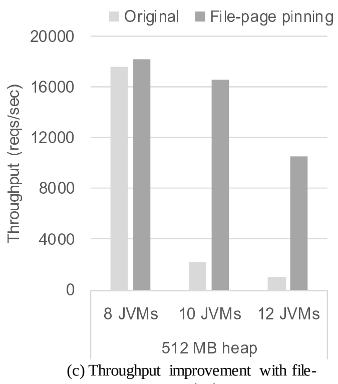
T. Salomie et al. and M. R. Hines et al. have applied a memory ballooning technique for a JVM [5] [20]. This approach avoids swapping out



(a) Disk accesses per request



(b) Disk accesses per request with file-page pinning



(c) Throughput improvement with file-page pinning

Figure 7: Effects of pinning frequently-accessed file pages to a physical memory in AcmeAir. (a) shows the original amount of disk accesses per request without file-page pinning. (b) shows the memory breakdown of JVMs without file-page pinning. (c) shows the amount of disk accesses per request with file-page pinning.

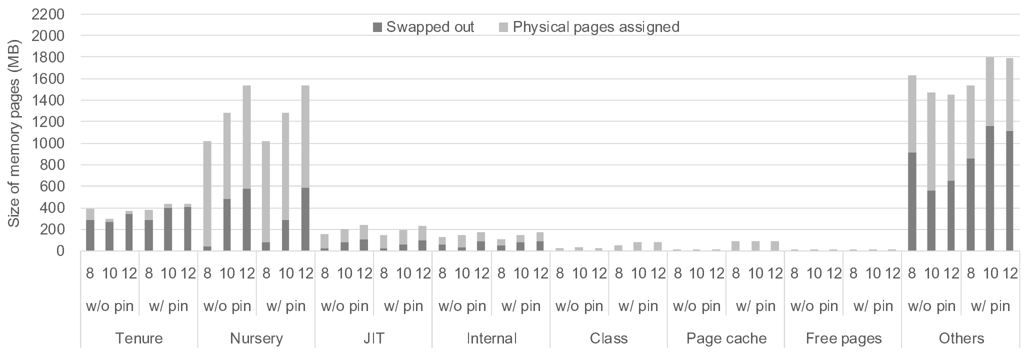


Figure 8: Breakdown of memory pages for AcmeAir. We changed the number of JVMs with fixing the heap size to 512 MB.

memory pages in a Java heap area by forcing a JVM to pin some of them and to reclaim the pinned pages for a hypervisor. However, our experimental results show that swapping out memory pages in a Java heap area does not always cause substantial performance degradation. Swapping out memory pages of a tenure space could be another effective option to reclaim memory without much impact on the performance.

Many techniques have been used to reduce the overhead to access the swapped-out memory pages for guest OSes [17] [24] [13] [21] [1] [16]. CMM2 [21] and VSwapper [1] keep track of I/O activities of guest OSes to identify rarely-accessed memory pages for swapping-out targets. Overdriver [24] reduces the disk-access overhead on swapped-out memory pages by allowing a host machine to use the memory areas of other host machines as its swap area through a network. Hypervisor exclusive cache [16], Tmem [17], and Mortar [13] are the hypervisor-level caches to store the memory pages that are swapped out by guest OSes. We consider that the current Linux memory management system needs to incorporate these kinds of techniques because our experimental results show that the page-cache reclamation algorithm is too aggressive.

The efficiency of the memory management policy in Linux is more important for workloads running in Docker containers [10] since Docker is an OS-based virtualization technology that relies on the host operating system to manage memory pages. Nakazawa et al. already showed that swappiness has major performance impacts on container workloads that oversubscribe memory especially when the load is unbalanced [18]. Our future work is to enhance the Linux memory management system to achieve a better trade-off between swapping and page-cache reclamation for Docker containers.

Many researchers have evaluated the performance of Java applications when memory swapping occurs [2] [26] [4] [3] [25] [22]. They have shown that memory swapping degrades the performance substantially, and they have proposed techniques to resize the heap sizes of JVMs to fit the total working set in a physical memory size. However, they have not analyzed which memory areas are swapped out and why the performance degradation occurs. Although Grzegorczyk et al. [2] have pointed out that the tenure space is swapped out first under a high memory pressure environment, they have not shown any data as evidence. We are the first to clarify the swapped-out memory areas of Java applications and the root cause of the performance degradation.

7 Conclusion

In this paper, we characterized the performance of two benchmarks that represent two different Java-based Web application frameworks on a Linux OS: DayTrader based on a stateful EJB framework and AcmeAir based on a stateless REST API. We investigated how we should enhance a virtual-memory system to improve the performance in a memory-oversubscribed environment. Our major finding is that the page-cache reclamation on a Linux virtual-memory system is a major cause of performance degradation when Java workloads oversubscribe memory. Although many memory pages are swapped out, accesses to the swapped-out memory pages are not the primary reason for the performance degradation. The Linux virtual memory system tends to reclaim memory pages from the page cache (i.e. file pages) more aggressively than those from the user memory space (i.e. anonymous pages). As a result, it frees a part of the page cache that stores files to generate excessive disk accesses to re-read such files when the memory pressure is high. We expect that a virtual memory system can reduce this type of performance degradation by using disk access information to select a victim page to be reclaimed. The thorough understanding of the interaction between the application framework and the memory oversubscription resiliency requires studies on a larger number of applications and is beyond the scope of this paper.

References

[1] Amit, N., Tsafrir, D., and Schuster, A.: VSwapper: A Memory Swapper for Virtualized Environments, *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, 2014.

[2] Grzegorczyk, C., Soman, S., Krintz, C., and Wolski, R.: Isla Vista heap sizing: Using feedback to avoid paging, *Proceedings of the International Symposium on Code Generation and Optimization*, 2007.

[3] Hertz, M., Feng, Y., and D, E.: Garbage collection without paging, *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.

[4] Hertz, M., Kane, S., Keudel, E., Bai, T., Ding, C., Gu, X., and Bard, J. E.: Waste not, want not: resource-based garbage collection in a shared environment, *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*, 2011.

[5] Hines, M. R., Gordon, A., Silva, M., Silva, D. D., Ryu, K., and Ben-Yehuda, M.: Applications Know Best: Performance-Driven Memory Overcom-

mit with Ginkgo, *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, 2011.

[6] http://geronimo.apache.org/GMOxDOC22/daytrader-a-more-complex application.html: Daytrader - a more complex application.

[7] http://linux mm.org/PageReplacementDesign: Page Replacement Design.

[8] https://developer.ibm.com/wasdev/websphere liberty/: About WAS Liberty.

[9] https://github.com/acmeair/acmeair: A Java implementation of the Acme Air Sample Application.

[10] https://www.docker.com/: docker.

[11] https://www.mongodb.org/: MongoDB for GIANT Ideas.

[12] http://www 01.ibm.com/software/data/db2/: IBM DB2 database software.

[13] Hwang, J., Uppal, A., Wood, T., and Huang, H.: Mortar: Filling the Gaps in Data Center Memory, *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 2014.

[14] Jacobi, C., Slegel, T., and Greiner, D.: Transactional Memory Architecture and Implementation for IBM System Z, *Proceedings of the 2012 45th Annual IEEE/ACM Interna-tional Symposium on Microarchitecture*, 2012.

[15] Kim, J., Fedorov, V., Gratz, P. V., and Reddy, A. L. N.: Dynamic Memory Pressure Aware Ballooning, *Proceedings of the 2015 International Symposium on Memory Systems*, 2015.

[16] Lu, P. and Shen, K.: Virtual machine memory access tracing with hypervisor exclusive cache, *Proceedings of the USENIX Annual Technical Conference*, 2007.

[17] Magenheimer, D., Mason, C., McCracken, D., and Hackel, K.: Transcendent Memory and Linux.

[18] Nakazawa, R., Ogata, K., Seelam, S., and Onodera, T.: Taming Performance Degradation of Containers in the Case of Extreme Memory Overcommitment, *Proceedings of the 10th IEEE International Conference on Cloud Computing*, 2017.

[19] Ogata, K., Mikurube, D., Kawachiya, K., Trent, S., and Onodera, T.: A Study of Java's Non-Java Memory, *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, 2010.

[20] Salomie, T., Alonso, G., Roscoe, T., and Elphinstone, K.: Application level ballooning for efficient server consolidation, *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013.

[21] Schwidefsky, M., Franke, H., Mansell, R., Raj, H., Osisek, D., and Choi, J.: Collaborative memory management in hosted Linux environments, *Ottawa Linux Symposium*, 2006.

[22] Tay, Y. C., Zong, X., and He, X.: An equation-based heap sizing rule, *Performance Evaluation*, 2013.

[23] Waldspurger, C. A.: Memory resource management in VMware ESX server, *Proceedings of USENIX OSDI*, 2002.

[24] Williams, D., Jamjoom, H., Liu, Y., and Weatherspoon, H.: Overdriver: Handling Memory Overload in an Oversubscribed Cloud, *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 2011.

[25] Yang, T., Berger, E. D., Kaplan, S. F., and Moss, J. E. B.: Cramm: Virtual memory support for garbage-collected applications, *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[26] Zhang, C., Kelsey, K., X. Shen, C. D., Hertz, M., and Ogihara, M.: Program-level adaptive memory management, *Proceedings of the 5th International Symposium on Memory Management*, 2006.

[27] Zhao, W. and Wang, Z.: Dynamic memory balancing for virtual machines, *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 2009.