Taming Memory Related Performance Pitfalls in Linux Cgroups

Zhenyun Zhuang, Cuong Tran, Jerry Weng, Haricharan Ramachandra, Badri Sridharan
2029 Stierlin Ct, Mountain View, CA 94043, USA
{*zzhuang, ctran, jweng, hramachandra, bsridharan*}*@linkedin.com*

***Abstract*—Linux kernel feature of Cgroups (Control Groups) is being increasingly adopted for running applications in multi-tenanted environments. Many projects (e.g., Docker) rely on cgroups to isolate resources such as CPU and memory. It is critical to ensure high performance for such deployments.**

At LinkedIn, we have been using Cgroups and investigated its performance. This work presents our findings about memory-related performance issues of cgroups during certain scenarios. These issues can significantly affect the performance of the applications running in cgroups. Specifically, (1) memory is not reserved for cgroups (as with virtual machines); (2) both anonymous memory and page cache are part of memory limit and the former can evict the latter; (3) OS can steal page cache from any cgroups; (4) OS can swap any cgroups. We provide a set of recommendations for addressing these issues.

I. Introduction

Container-based solutions [1] such as Docker [2] are gaining popularity for rapid and efficient application deployments. Compared with traditional virtual machine based solutions, containers are light-weight, particularly suitable for large-scale deployments. Various container projects and solutions (e.g., Docker and CoreOS [3]) rely on cgroups to provide resource isolations.

Cgroups (Control groups) [4] provides kernel mechanisms to isolate the resource usage of different applications. These resource types include memory, CPU and disk IO. Among them, memory usage is one of the most important resource types that impacts application performance.

On Linux, cgroups feature has a root cgroup which serves as the base of the cgroup hierarchy. Multiple non-root cgroups (i.e., regular cgroups) can be configured and deployed, each with fixed memory limit. A process can be explicitly assigned to regular cgroups which are bounded by certain memory limits. Any processes that are not assigned to regular cgroups are managed by the root cgroup.

Though cgroups is designed to isolate memory usage of each regular cgroup, based on our experiences, it fails to deliver good application performance in certain memory-pressured scenarios (i.e., the system lacks sufficient free memory). We have found multiple performance pitfalls with regard to cgroups usage. If not careful enough, applications deployed in cgroups could be badly hit by these performance pitfalls. Most of these problems are attributed to the fact that the root cgroup is unbounded in terms of memory usage, hence processes in root cgroup can starve other cgroups, regardless of memory limit sizing of regular cgroups. Specifically, (1) The memory limit imposed to cgroups is the sum of both user-space anonymous memory and kernel-space page cache. Due to lack of separate capping of anonymous memory and page cache, a cgroup's anonymous memory usage could starve its page cache usage, causing degraded application performance; (2) due to the nature of the cgroups, OS may steal page caches from all cgroups if needed, regardless of whether the cgroups have reached their memory limit or not; (3) similarly, OS may decide to swap the anonymous memory from all cgroups, completely ignoring the memory usage of the cgroups.

In this work, we studied cgroups performance under various types of memory pressure scenarios, and classified the performance issues when using cgroups. If not controlled carefully, these cgroups performance pitfalls can significantly affect applications running in cgroups. We also propose a set of recommendations to address these pitfalls.

II. Background

Before moving on to the issues, we use Figure 1 to present some backgrounds. A root cgroup exists on each system, and multiple cgroups can be configured. A regular cgroups memory usage includes anonymous (i.e., user space) memory (e.g., malloc() allocated), page cache and the memory yet to be allocated. The sum of these three parts is capped by the memory limit configured for the particular cgroup. The root cgroups memory is unbounded with no limit.

Each cgroup can have its own setting of swappiness value indicating the preference of swapping when reclaiming memory. Setting swappiness to 0 in a cgroup will prevent the cgroup's anonymous memory from being swaped by its own memory request, but all cgroups use the same swap space configured by OS. Similarly, though each cgroup can use page cache, all page caches are maintained by OS.

Processes can be assigned to regular cgroups. Once assigned, various types of memory usage by the particular process will be charged to the cgroup. Once the sum of all memory usage inside a cgroup reaches the configured memory limit, new memory requests will be rejected.

III. Performance Pitfalls

Typical business computing environments never lack surprises in terms of computing resource usage on a production machine. Take the example of memory resource, a machine

978-1-5090-4588-4/17/$31.00 ©2017 IEEE

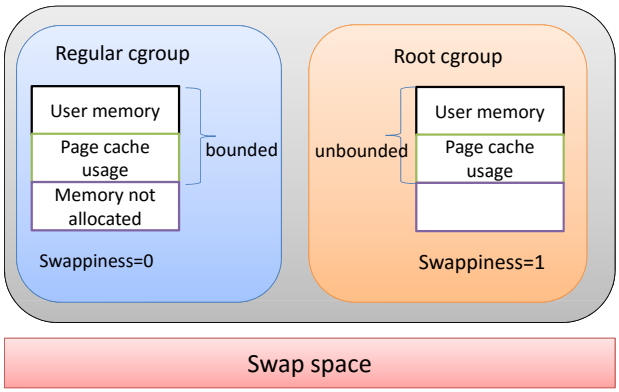


Figure 1. **Cgroups illustration: regular cgroups and root cgroup**

may lack enough free memory to supply all types of requests. For instance, users may remotely login (i.e., ssh) to a machine and do log analysis, which uses both page cache and anonymous memory. When the machine's free memory drops to near zero, it creates a memory pressure scenario.

There are many types of memory pressure scenarios when using cgroups, depending where the memory pressure occurs and what type of memory pressure is. When there is memory pressure in the root cgroup or regular cgroups, one party may affect the performance of other parties. The impacts of these issues include degraded application performance (e.g., application throughput, response latency, etc.). For instance, application startup can be much slower if OS has to free up memory to satisfy application memory request.

We have found multiple performance pitfalls in these memory pressure scenarios with regard to cgroups usage. For each of the pitfalls, we provide corresponding experiment results. The experiment setups are as follows. The machine runs RHEL 7 with Linux kernel of 3.10.0-327.10.1.el7.x86_64 and 64GB physical RAM. The hardware is dual-socket with totally 24 virtual cores (Hyper-threading enabled). OS level swapping is enabled (swappiness=1) and there are totally 16GB of swapping space. Swapping in all cgroups is disabled by setting swappiness=0.

The workload used to request anonymous memory is a JVM application which allocating and de-allocating objects. The baseline application throughput is 139K alloc/sec. Performance metrics include: application throughput, cgroups statistics such as page cache, swap and RSS size, OS "free" utility reported statistics such as swap and page cache size.

A. Memory is not reserved for cgroups (as with virtual machines)

A cgroup only imposes upper limit on memory usage by applications in the cgroup. It does not reserve (i.e., pre-allocate) memory for these applications and as such, memory is allocated on demand and applications still compete for free memory when deployed in cgroups.

One implication of this feature is that, when the cgroup later on requests for more memory (still within its memory limit), the requested memory needs to be allocated by OS on the fly. If OS does not have enough free memory, OS has to reclaim from page cache or anonymous memory, depending on the swapping setup on the OS (i.e., swappiness value and swap space). Depending on scenarios, the memory reclaiming by OS could be performance killer, affecting the performance of other cgroups.

Ensuring the cgroups memory usage has not reached its limit, and we run a process which requests anonymous memory. If OS needs to reclaim page cache to satisfy cgroup request and the page cache is dirty, then OS needs to "write back" (i.e., persisting the modified file data to disks) the dirty page cache, which is slow when the backup files are on HDD. The cgroup process requesting memory needs to wait for the memory request and experiences degraded performance. In one of the tests, the application throughput is 110K alloc/sec, or 16% lower than baseline throughput. Note that the actual throughput depends on the writeback amount of dirty page cache and the requesting memory size.

B. Both anonymous memory and page cache are part of memory limit and the former can evict the latter

A cgroups memory limit (e.g., 10GB) includes all memory usages of the processes running in it. Both anonymous memory and page cache of the cgroup are counted towards the memory limit. In other words, when the application running in a cgroup read/write files, the corresponding page cache allocated by OS is part of the cgroups memory limit. When the cgroup is reaching its memory limit, further anonymous requests by in-cgroup applications will cause the eviction of page cache of the cgroup.

For the particular application, the starvation of page cache leads to low page cache hit rate, which not only degrades the performance of the application and increases workload on root disk drive, but could severely degrade the performance of the entire system. Other cgroups (and the applications) may suffer from disk IO problems caused by the intensified disk usage by the particular cgroup.

A cgroup (cg1) is configured with 20GB memory limit. The process running inside firstly uses all page cache. Then the process requests anonymous memory. We found the page cache of cg1 dropped by 6GB (from 20GB to 14GB). The rss (Resident Set Size, as reported by the cgroups memory.stat counters) of the process increases by 7GB (from 0 to 7GB), due to its anonymous memory requests. It indicates that in a cgroup, the anonymous memory request can steal page cache.

C. OS can steal page cache from any cgroups

Though page cache is part of memory limit of each cgroup, OS manages the the entire page cache space and does not respect the owners during memory reclaiming. OS treats all page cache equally, using a single LRU algorithm. OS does not respect the owner (e.g., regular cgroups) of the

page cache during page cache reclaiming, and it may reclaim from system-wide (can be from any cgroups).

Two cgroups (cg1 and cg2) both have 20GB memory limit. Both cgroups have used about 20GB of page cache. Root cgroup requests for 24GB of page cache (i.e, copy big files). We found cg1 page cache dropped by 10GB (from 21GB to 11GB). Cg2 page cache dropped by 12GB (from 21GB to 9GB). The results show that OS can steal page cache from all cgroups.

D. OS can swap any cgroups

Similar to the control policy of page cache, though anonymous memory used by cgroups is part of the respective memory limit, OS manages the entire swap space and rules over regular cgroup's swapping settings. Swapping control policy (e.g., swappiness) of the system (i.e., root cgroup) takes precedence over regular cgroups'. When there is no memory pressure outside a regular cgroup, setting cgroup's swappiness to 0 prevents swapping of processes inside the cgroup. However, if system policy allows swapping, OS can swap out processes in any cgroup when under memory pressure even if the victim cgroup's swappiness is set to 0 and memory limit is not reached.

Two cgroups of cg1 and cg2 both have 20GB memory limit, and each uses 15GB of anonymous memory. Root cgroup then requests 35GB of anonymous memory. As shown in Figure 2, cg1 anonymous memory is swapped out by 1.4GB (from 0GB to 1.4GB). Cg1 anonymous memory is swapped out by 1.3GB (from 0GB to 1.3GB). Root cgroup swap size increases by 1.8GB (from 0GB to 1.8GB).

IV. Strategies

For all the memory related issues we listed, the key solution is to ensure the memory usage of each cgroups (both root cgroup and regular cgroups) is tightly controlled. Unlike regular cgroups, the root cgroup's memory usage is unbounded. So unless all processes in root cgroup being memory-controlled, root cgroup may use too much memory and starve non-root cgroups. We recommend a set of approaches to mitigate the performance issues. Each of the approaches targets a particular issue and belongs to a particular category, but they work better in tandem.

A. (Regular cgroups) Pre-touching the requested memory

Since the memory limit of cgroups is not allocated beforehand, it helps to "pre-touch" the requested anonymous memory and avoid use-as-you-go requests. The exact methods of pre-touch memory vary across languages. Take an example of Java application that uses heap. We can use the "-XX:+AlwaysPreTouch" flag to pre-touch the Java heap during JVM initialization. For off-heap memory, it also helps to ask the application to pre-allocate needed memory when possible.

B. (Application onboarding) Estimating page cache to decide memory limit

Since a cgroup's memory limit includes both anonymous memory and page cache used by the processes in the cgroup, deploying applications to cgroups should consider an application's memory usage of both types. Unfortunately estimating the page cache usage for an application is very difficult, as there is no direct Linux metric on page cache usage by processes. Moreover, there are other mechanisms that further complicate the issue. One of such mechanisms is the file system's prefetching (i.e., file data are read into memory based on heuristics rather than actual usage).

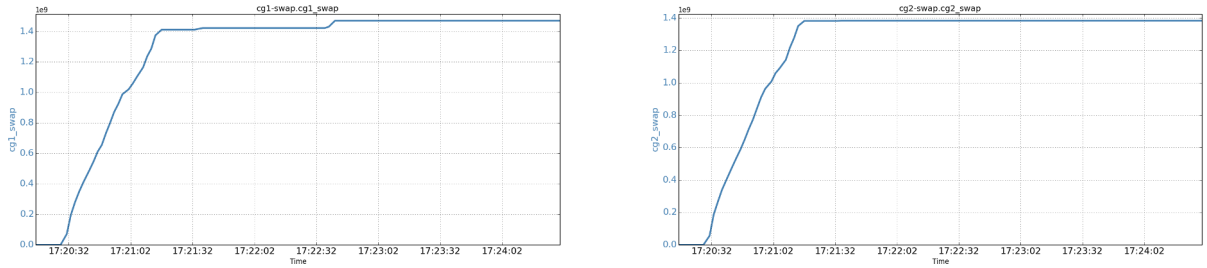
However, even though the exact size of the needed page cache is unavailable, an approximation would suffice for many deployments. Once an application is deployed to a cgroup with particular memory limit, it is possible to *resize* the memory limit by monitoring the memory usage (both anonymous memory and page cache) and correspondingly adjust the memory limit. Due to page limit, we will not elaborate on the details.

C. (Root cgroup) Isolating memory usage of system utilities and housekeeping processes

One of the causes of the performance issues is the unbounded memory usage of root cgroup. Different from typical applications, system or housekeeping processes ((e.g., sshd [5], crond [6])) by default run in root cgroup. The unbounded memory usage of these processes can use unexpected amount of memory, and hence affecting non-root cgroups. Take an example of sshd. When a remote user login to the machine using ssh, the memory usage of the particular user is charged to root cgroup (since sshd belongs to root cgroup). Since the user is not limited with regard to the page cache (e.g., copying files) and anonymous memory (e.g., run other processes), root cgroups memory usage could go limitless.

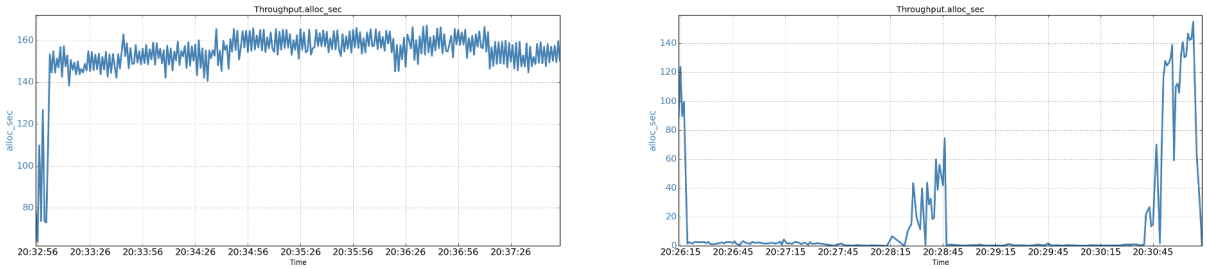
If processes in root cgroup occupy too much anonymous memory and/or produce too much dirty page cache, they may step on regular cgroups and cause performance problems of the latter. To address this, we should keep minimal housekeeping processes in root cgroup, and move all such processes to properly sized cgroups. By doing so, the root cgroups resource usage is kept as little as possible, hence reducing the chance of "pressuring" non-cgroups.

In particular, cron could run both OS utilities as well as application jobs. To avoid the pitfalls of cron jobs using too much anonymous memory and page cache, we should put cron in a tightly-controlled cgroup. Alternatively, we could prevent applications from being scheduled cron jobs. Similarly, remote ssh by users could also use a lot of anonymous memory and page cache, which may render root cgroup having less free memory for regular cgroups. We have to move sshd to a separate cgroup with tight memory control.



(a) Cg1's swap size (b) Cg2's swap size

Figure 2. **OS can swap memory pages from cgroups**



(a) Baseline (With pretouching) (b) Without memory pretouching

Figure 3. **Application throughput (K allocations/sec)**

V. Evaluation

We use a machine of Intel(R) Xeon(R) CPU E5-2680 v3@2.50GHz. The machine has 2 sockets, each socket has 6 physical cores. With hyper-threading enabled, it totally has 24 logical cores and 64GB of memory with NUMA. The OS is RedHat Enterprise Linux (RHEL 6, 2.6.32-573.7.1.el6.x86_64).

The machine runs Java HotSpot(TM) 64-Bit Server VM which runs "1.8.0_40" (build 1.8.0_40-b26) version. The JVM instance invokes 2 application threads, each thread aggressively allocating JVM objects to an internal array. The objects are periodically deallocated after the array is full.

A. Baseline

The goal of this work is to prevent certain undesired memory pressure scenarios from affecting cgroup's performance. Different from other works that present "optimizations" types of solutions and showcase how much performance improvements, this work takes the opposite approach. We first present the "expected" (i.e., desired) performance as the baseline. Such performance is obtained in scenarios with our recommendations (presented in Section IV) applied. We then present the "unexpected" (i.e., undesired) performance in various memory pressure scenarios where our recommendations are not applied.

We obtain the baseline results by creating a cgroup with 20GB of memory limit. Inside this cgroup, the JVM application is started with 20GB heap size and runs for 5 minutes. During the runtime, we collect various performance data including the application throughput, cgroup's statistics such as page cache usage and resident set size [1].

Since OS is able to swap the anonymous memory of the entire system (both root cgroup and non-root cgroups), it is important to prevent each cgroup's anonymous memory from being swapped out. For this purpose, we set the *swappiness* of non-root cgroups to 0. For the root cgroup, we set the *swappiness* to 1.

We plot the 5-minute run instantaneous throughput in Figure 3(a). The application throughput is 153.61 K allocations/second with JVM AlwaysPreTouch enabled (i.e., -XX:+AlwaysPreTouch).

B. Not *pretouching cgroup memory*

Without AlwaysPreTouch JVM flag, the heap space used by JVM will be gradually allocated as the applications use memory. When the requested memory is not readily available, OS needs to swap some anonymous memory out (when cgroup's swappiness>0). In one such scenario (swappiness=50), we plot the application throughput in Figure 3(b). We can see the application throughput is much lower and varying significantly. The average application throughput is 20.15 K allocations/second.

[1]Each cgroup has a memory.stat file which collects various usage statistics of the cgroup.

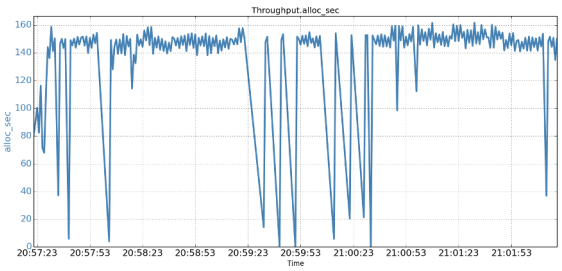


Figure 4. **Under-allocating memory limit due to less page cache**

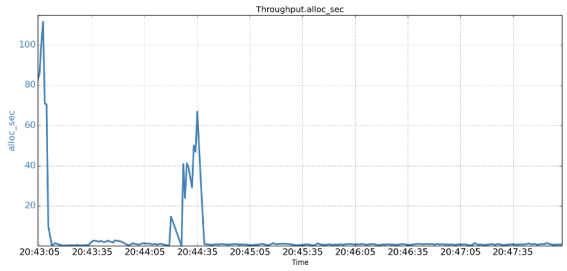


Figure 5. **Unprotected anonymous memory**

C. Not *including page cache into cgroup memory limit*

When the cgroup is not configured to have enough memory to cache file, the application performance suffers. We create such a scenario where the anonymous memory request is similar to the cgroup memory limit, while there are also page cache requests in the cgroup. Running the same JVM application with 20GB heap size and copying a 20GB file in a cgroup with 20GB limit, the application throughput is throughput 140.42 K allocations/sec (10% lower than the baseline), as shown in Figure 4.

D. Not *protecting anonymous memory of cgroups*

If the cgroup's anonymous memory is not protected from swapping, when application requests more memory than the memory limit, swapping will happen and the application will perform badly. We set swappiness to 50 to create a scenario. The cgroup has 10GB memory limit, and the JVM requests 20GB heap. The application throughput is mere 5.68 K allocations/second, as shown in Figure 5. The RSS size is constant at 10GB (i.e., memory limit of the cgroup).

VI. Related Work

There are very limited amount of related work in the area we address. Though there are various posts and blogs elaborating on how to use cgroups or cgroups-based containers, we have not seen any similar work in literature that identify or address the problems exposed in our work.

Virtual machine (VM) technologies [7], [8] have been in production for long time. VM allows multiple OS instances to share the same hardware to increase resource usage efficiency.

Compared to VM, containers (i.e., OS-level virtualization) allows multiple isolated user-space instances to co-exist on the same OS. Examples of containers include Docker [2] and CoreOS [3]), both are gaining in popularity.

There are various types of tradeoffs between them, but the industry trend is shifting from VM to containers in recent years. For instance, Docker is particularly popular thanks to its added features including imaging and namespace isolation. It has been integrated to various infrastructure tools including AWS (Amazon Web Services) [9] and Google Cloud Platform [10].

VII. Conclusion

While cgroups provides a decent mechanism to isolate the memory usage of regular cgroups, using it for application deployments requires special treatements to prevent applications running in cgroups from being affected in memory pressure scenarios. This work discusses several of these scenarios and present solutions to address the problems.

References

[1] P. Bellasi, G. Massari, and W. Fornaciari, "Effective runtime resource management using linux control groups with the barbequertrm framework," *ACM Trans. Embed. Comput. Syst.*, vol. 14, no. 2, pp. 39:1–39:17, Mar. 2015.

[2] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014.

[3] "Coreos," https://coreos.com/.

[4] "Linux cgroups," https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.

[5] "sshd - openssh ssh daemon," http://linux.die.net/man/8/sshd.

[6] "cron - daemon to execute scheduled commands," http://linux.die.net/man/8/crond.

[7] J. E. Smith and R. Nair, "The architecture of virtual machines," *Computer*, vol. 38, no. 5, pp. 32–38, May 2005.

[8] C. A. Waldspurger, "Memory resource management in vmware esx server," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 181–194, Dec. 2002.

[9] A. Wittig and M. Wittig, *Amazon Web Services in Action*, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2015.

[10] X. Jia, "Google cloud computing platform technology architecture and the impact of its cost," in *Proceedings of the 2010 Second World Congress on Software Engineering - Volume 01*, ser. WCSE '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 17–20.