Highlighting the Container Memory Consolidation Problems in Linux

Francis Laniel, Damien Carver, Julien Sopena, Franck Wajsburt, Jonathan Lejeune, Marc Shapiro

Highlighting container memory consolidation problem in Linux

Francis Laniel[‡], Damien Carver[‡], Julien Sopena[‡], Franck Wajsburt[*], Jonathan Lejeune[‡] and Marc Shapiro[‡]
[‡]*Sorbonne Université, CNRS, Inria, LIP6, DELYS Team, F-75005 Paris, France*
[*]*Sorbonne Université, CNRS, LIP6, F-75005 Paris, France*
[*‡]firstname.lastname@lip6.fr

***Abstract*—The container mechanism supports server consolidation; to ensure memory performance isolation, Linux relies on static memory limits. However, this results in poor performance, because an application needs are dynamic. In this article we will show current problems with memory consolidation for containers in Linux.**

***Index Terms*—Linux, container, memory, memory consolidation**

I. Introduction

This paper is about showing the problem of memory consolidation for containers in Linux.

Logical servers are *consolidated* by running several of them on a single physical machine, thereby saving money. The logical servers should remain mutually isolated, *i.e.* execution of one logical server should not impact any other.

One popular consolidation mechanism, in Linux is called "containers" [1]–[4]. A container is a group of processes isolated from the others. Containers are advantageous for security (*e.g.* a file that belongs to a container can not be read from another), deployment (*e.g.* running a container is as simple as running a shell command) and resource control (*e.g.* a container can be restricted to, say, a specific CPU core) [5].

This article focuses specifically on container *memory consolidation*, which consists of multiplexing the physical memory of the machine across the virtual memory demands of all containers. Memory consolidation can be effective if, at every point in time, the total memory demands of all containers is less than the available physical memory, which is the common case, because data center utilization usually remains well under 100% [6], [7].

The classical virtual memory mechanism ensures security isolation. However, it is insufficient for *memory performance isolation* (ensuring to each container enough memory for it to perform well) because it does not stop one container from starving the others of physical memory. Furthermore, the metrics available to the kernel (*e.g.* frequency of page faults, I/O requests, use of CPU cycle, *etc.*) are not suitable for performance isolation, because they are not directly relevant to performance as experienced from the application perspective, which is better characterized by, for instance latency or throughput.

To avoid a container starving the others, the system administrator can limit the total amount of its physical memory. If it would exceed its limit, some of its memory will be reclaimed, making it available to others. In practice, the kernel tends to reclaim preferably from the file page cache; this causes a performance decrease for containers that access files [8]. The limits to container size are static, and do not adapt to the containers' dynamic behaviors. This is problematic, because it is challenging to estimate the optimal amount of memory for an application to execute smoothly [9], [10].

Our main contribution consists in an experimental demonstration of the limitations of the existing Linux mechanisms.

We organize the remainder of our paper as follows. Section II presents some technical background. Section III shows the issues with existing Linux mechanisms. Finally, we conclude and discuss future work in Section IV.

II. Technical background

A container is represented using the Linux `cgroup` mechanism [11]–[14]. A `cgroup` is a set of processes whose collective usage of resources is limited. In particular, the total amount of physical memory used across all the processes of a memory `cgroup` is capped by the `max` and `soft` limits described hereafter. In the rest of this section, we study how Linux manages the memory of containers under *memory pressure*, *i.e.* when free physical memory is scarce.

A. No limits

When no limits are set and under memory pressure, memory will be reclaimed from all containers. If containers have different memory needs, the kernel will allocate memory to satisfy them, even as their needs change during execution. Indeed, if a container uses, at some moment of its execution, less memory than earlier the kernel will reallocate the unused memory to another container. This constitutes memory consolidation; however there will not be memory performance isolation. Indeed, if a container uses less its memory it will lose it, because it will be reclaimed. In summary, when no limits are set, memory consolidation occurs at the expense of memory performance isolation.

B. The `max` and `soft` limit mechanism

Linux has two mechanisms to control memory reclamation of containers. The first is the `max` limit; a container will not be allocated a memory footprint larger than its `max` limit, even if there exists unused memory elsewhere. If a process needs physical memory, and its container has already reached its `max`

limit, the kernel may reallocate memory from another process of the same container, but not from another container. This mechanism avoids situations where some containers would starve the others.

The `soft` limit is similar to the `max` limit, except that it is active only when there is memory pressure and it is only best-effort. The intent is that the system administrator will set a `soft` limit approximating the container's Working Set (WS) size [15]. Unfortunately, it is hard to estimate WS size [9], [10].

In other words, the size of a container is limited to its `soft` limit when there is memory pressure, otherwise, to its `max` limit. However, these limits are not directly related to the containers' current memory needs; they may be higher (impeding memory consolidation) or lower (impeding memory performance isolation).

III. Memory consolidation *VS.* memory performance isolation

A. Reference experiment

In order to demonstrate the issues experimentally, we first run a reference experiment. This experiment consists of executing a benchmark, detailed hereafter, to obtain reference number for throughput and memory footprint . The throughput was obtained from the benchmark output while memory and inputs were collected by `docker`. We measure the max throughput, under high offered load, to be 800 transactions per second. The reference memory footprint was measured to be 2.8 GB. These numbers are pictured in Figure 1 to Figure 3 as horizontal green lines. In the scenario that we will describe in next section, the high offered load corresponds to the generation of 800 transactions per second. We defined low offered load as what is required to reach 200 transactions per second. This number is showed as horizontal black line in Figure 1a to Figure 3a.

B. Experimental scenario

We run an experiment whose goal is to show that without limits there is no memory performance isolation, and that the `max` and `soft` limits impede memory consolidation. Our experimental scenario has two containers, A and B, which run an OLTP workload and experience changes in activity. We expect that, when both containers have high offered load, memory performance isolation should occur. When one container has high offered load and the other has low offered load or is stopped, memory consolidation should allow it to reach its reference performance. Specifically, the experiment has 6 steps:

- φ_1 A and B both have high offered loads. Containers will bring database records in memory but physical memory available is inferior to databases' sizes. So, there will be memory pressure. Containers will then have to access the disk so their throughputs will be low.
- φ_2 The high offered load continues for A, and it decreases for B. If memory consolidation is effective A should be able to take memory from B, and increase its throughput to the reference value.
- φ_3 B has high offered load. This step is similar to step 1. So, containers will have a low throughput.
- φ_4 Both containers have low offered loads. We expect that they will be able to answer to all the transactions.
- φ_5 B has high offered load, not A. This step is symmetric to step 2.
- φ_6 A is completely stopped. B still has high offered load. We expect B to perform the same as the reference.

Each step of the scenario lasts for 180 seconds.

C. Experimental environment

Our experimental machine is a workstation with an Intel®Xeon®E5-1603 clocked at 2.8 GHz, 8 GB of DDR3 and a SSD. To create memory pressure, we run our experiments in a virtual machine (VM) restricted to 4 CPU cores and 3 GB of memory.

We use `qemu 2.8.1` as VM hypervisor, `docker 18.09.6` and its `python` library `docker-py 3.7.2` to manage containers. Our containers execute the benchmark `sysbench oltp`. We modified this benchmark to dynamically change the rate of transactions generated. The benchmark makes request to a database managed by `mysql 5.7`. We run our experiments on Linux 4.19.

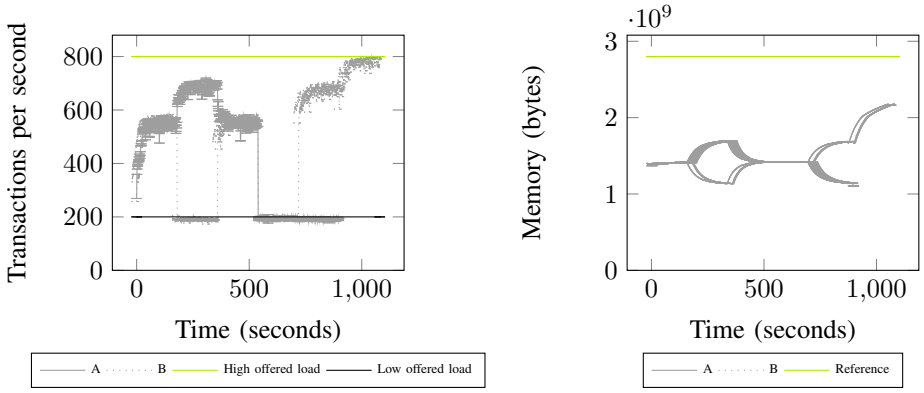
Our two containers (A and B) run with 2 cores each. They each read a database of 4 GB. Since our VM has only 3 GB of memory, the two databases will not fit in memory and some accesses to those databases will not be cached by Linux page cache resulting in physical I/O, which heavily decrease the throughput.

Then, we run the scenario described in Section III-B 10 times and compute the mean and standard deviation every second. Each point in curves depicted in Figure 1, Figure 2 and Figure 3 is the mean and its associated standard deviation for this second across the 10 runs.

D. Experiment with no limits set

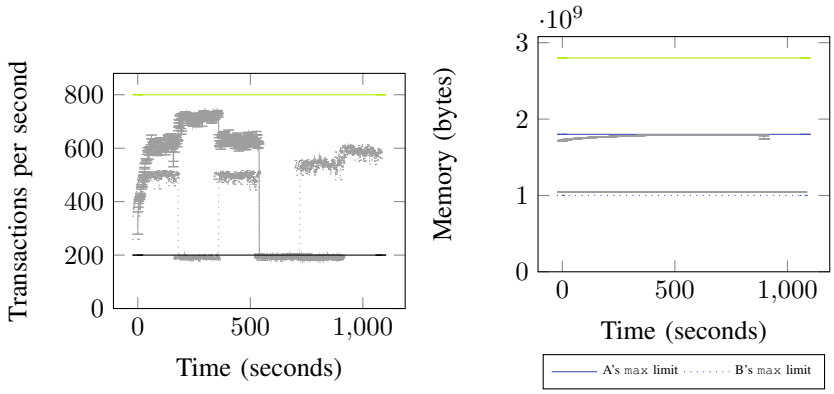
We run the experiment described in Section III-B without setting any limits. By doing so, we wish to demonstrate that all containers are reclaimed equally, and there is no memory performance isolation. Our results are depicted in Figure 1.

Figure 1a shows the containers' throughputs over time and Figure 1b depicts their memory footprints. In Figure 1a, the throughput of a container that has high offered load increases at the expense of one with low offered load (φ_2 and φ_5). Figure 1b shows that the memory footprint of the container with high offered load increases, while the container with low offered load shrinks. Memory consolidation is taking place but imperfectly, since an active container never reaches the reference level of performance. When A stops, B's footprint grows, so its performance increases, reaching the reference level (φ_6). When both containers have high offered loads they have the same throughput because they have the same memory footprint, due to the lack of memory performance isolation (φ_1 and φ_3).



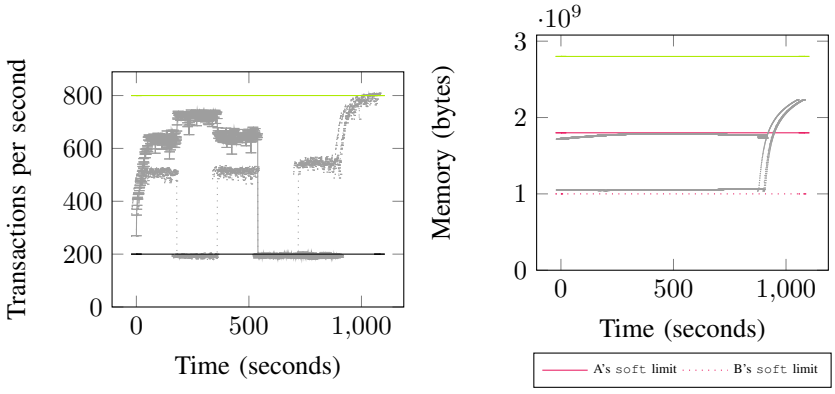
(a) Throughput (b) Physical memory footprint

Fig. 1: No limits set



(a) Throughput (b) Physical memory footprint

Fig. 2: `Max` limits set to 1.8 GB (A) and 1 GB (B)



(a) Throughput (b) Physical memory footprint

Fig. 3: `Soft` limits set to 1.8 GB (A) and 1 GB (B)

E. Experiment with `max` limit

We demonstrate that `max` limit impedes memory consolidation, by running the same experiment with `max` limits set to 1.8 GB for A and 1 GB for B. Those values were chosen such that their total equals the reference memory footprint.

In Figure 2a, A has better performance than B. This can be explained by looking at Figure 2b. Because the memory footprints reach the `max` limits, A has more memory than B; therefore it will be able to answer more requests (φ_1 and φ_3). Figure 2b shows that there is no memory consolidation, because even when B has low offered load, A's memory footprint does not increase, and it never reaches the reference level (φ_2). Conversely, when A has low offered load and B has high offered load, B's memory does not grow, because of its `max` limit (φ_5). Therefore, B reaches only 500 to 550 transactions per second. Worst, when A stops, B is not able to expand its memory footprint and experiences little increase in performance (φ_6).

To conclude, with the `max` limit there is memory performance isolation, since A acquires more memory than B, but memory consolidation is impeded.

F. Experiment with `soft` limit

Under memory pressure, the `soft` limit has a similar effect to the `max` limit, also impeding memory consolidation. We show this behavior by executing the experiment using `soft` limits. Figure 3 shows that the results are nearly the same as above. This is expected because during our experiment there is memory pressure, and containers will tend to remain under the `soft` limits, which equal the previous `max` limits. Note however how, in φ_6, B's footprint increases; because A was stopped, relaxing memory pressure, the `soft` limit mechanism is not activated anymore, allowing B to increase its memory allocation.

To summarize, the `soft` limit mechanism supports memory performance isolation (B has less memory than A) under memory pressure. This limits memory consolidation but when memory pressure disappears, memory consolidation becomes effective again.

G. Summary

Our study shows that the existing mechanisms are not effective at memory consolidation and memory performance isolation. Without limits, there is weak memory consolidation but no memory performance isolation. The `max` limit supports memory performance isolation, but impedes memory consolidation. The `soft` limit is analogous to the `max` limit under memory pressure, but when memory is plentiful it does enable memory consolidation. We summarize these results in Table I.

Mechanism used	Memory consolidation	Memory performance isolation
no limits	weak	no
`max` limit	no	yes
`soft` limit	no	yes

TABLE I: Summary of memory consolidation and memory performance isolation of existing mechanisms

IV. Conclusion and future works

In this paper, we showed the issues of memory consolidation *vs.* memory performance isolation for existing mechanisms with containers. Our experiments showed without limits, there is a weak memory consolidation, but no memory performance isolation since containers have the same memory footprints. The `max` limit offers memory performance isolation but consolidation is impeded. The `soft` limit, which activates only under memory pressure, acts similarly to the `max` limit.

In future work, we will extend the `soft` limit mechanism to consolidate memory. Our mechanism will be based on applicative probe to give information about containers performance to the kernel.

References

[1] Amazon, "Amazon Elastic Container Service." [Online]. Available: https://aws.amazon.com/ecs/?nc1=h_ls

[2] Microsoft, "Microsoft Web App for Containers." [Online]. Available: https://azure.microsoft.com/en-us/services/app-service/containers/

[3] Alibaba, "Alibaba Container Service." [Online]. Available: https://www.alibabacloud.com/product/container-service?spm=a2c5t.10695662.1996646101.searchclickresult.55a3212bnXyv1v

[4] OVH, "OVH Kubernetes." [Online]. Available: https://www.ovh.com/fr/kubernetes/

[5] Docker Inc., "What is a Container?" [Online]. Available: https://www.docker.com/resources/what-container

[6] D. Meisner, B. T. Gold, and T. F. Wenisch, "PowerNap: eliminating server idle power," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 1, p. 205, Mar. 2009. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2528521.1508269

[7] L. A. Barroso, J. Clidaras, and U. Hölzle, "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second edition," *Synthesis Lectures on Computer Architecture*, vol. 8, no. 3, pp. 1–154, Jul. 2013. [Online]. Available: http://www.morganclaypool.com/doi/abs/10.2200/S00516ED2V01Y201306CAC024

[8] The kernel development community, "Concepts overview." [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/mm/concepts.html

[9] B. Gregg, "Working Set Size Estimation," Feb. 2018. [Online]. Available: http://www.brendangregg.com/wss.html

[10] V. Nitu, A. Kocharyan, H. Yaya, A. Tchana, D. Hagimont, and H. Astsatryan, "Working Set Size Estimation Techniques in Virtualized Environments: One Size Does not Fit All," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 2, no. 1, pp. 1–22, Apr. 2018. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3203302.3179422

[11] K. Hiroyu, "Cgroup And Memory Resource Controller," Nov. 2008. [Online]. Available: https://www.static.linuxfound.org/jp_uploads/seminar20081119/CgroupMemcgMaster.pdf

[12] Rami Rosen, "Namespace and cgroups, the basis of Linux containers," Seville, Spain, Feb. 2016. [Online]. Available: https://www.netdevconf.org/1.1/proceedings/slides/rosen-namespaces-cgroups-lxc.pdf

[13] Linux, "Memory Resource Controller." [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v1/memory.txt

[14] Zhenyun Zhuang, Cuong Tran, J. Weng, H. Ramachandra, and B. Sridharan, "Taming memory related performance pitfalls in linux Cgroups," in *2017 International Conference on Computing, Networking and Communications (ICNC)*. Silicon Valley, CA, USA: IEEE, Jan. 2017, pp. 531–535. [Online]. Available: http://ieeexplore.ieee.org/document/7876184/

[15] P. J. Denning, "The working set model for program behavior," in *Proceedings of the ACM symposium on Operating System Principles - SOSP '67*. Not Known: ACM Press, 1967, pp. 15.1–15.12. [Online]. Available: http://portal.acm.org/citation.cfm?doid=800001.811670