ACDC : Advanced Consolidation for Dynamic Containers

Damien Carver, Julien Sopena, Sébastien Monnet

▶ To cite this version:

Damien Carver, Julien Sopena, Sébastien Monnet. ACDC : Advanced Consolidation for Dynamic Containers. NCA 2017 - 16th IEEE International Symposium on Network Computing and Applications, Oct 2017, Cambridge, MA, United States. pp.1-8. hal-01673304

HAL Id: hal-01673304

https://inria.hal.science/hal-01673304v1

Submitted on 29 Dec 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ACDC : Advanced Consolidation for Dynamic Containers

Damien Carver
Magency
60, Rue de Wattignies
75012 Paris, FRANCE
Email: damien@magency.fr

Julien Sopena
LIP6 - Laboratoire d'Informatique de Paris 6
4, Place Jussieu
75252 Paris Cedex 5, FRANCE
Email: julien.sopena@lip6.fr

Sebastien Monnet
Polytech' Annecy-Chambéry/LISTIC
5, chemin de bellevue
74944 Annecy-Le-Vieux
Email: sebastien.monnet@univ-smb.fr

***Abstract*—The thriving success of the Cloud Industry greatly relies on the fact that virtual resources are as good as bare metal resources when it comes to ensuring a given level of quality of service. Thanks to the isolation provided by virtualization techniques based on hypervisors, a big physical resource can be spatially multiplexed into smaller virtual resources which are easier to sell. Unfortunately, virtual machines have quickly shown their limit in terms of temporal multiplexing. It has been demonstrated that reclaiming the unused memory of a VM is a tedious task, infeasible in production. Today, containerization opens up a wide range of multiplexing opportunities that were not accessible through machine virtualization.**

However, in this article, we demonstrate, through a reproducible experiment, that the current implementation of memory consolidation can deteriorate the performance of applications deployed in Linux kernel containers. Indeed, we observed that when a new container boots, the memory of active containers is reclaimed while unused memory is still available in other containers that are inactive. To tackle these performance drop in active containers, we have rethought the hierarchical memory reclaim mechanism of the Linux kernel. We have implemented inside the kernel our new approach that tracks the container that has made a memory demand the least recently. Our evaluations show that our approach provides the ability to reclaim memory without disturbing performances.

***Index Terms*—cloud, memory, isolation, consolidation, containers, elasticity, Linux**

I. Introduction

Cloud computing is at the heart of the digital industry. One of the reasons for this success is undoubtedly the transition made from a model close to the rent of physical machines to a true resource pooling model. Today, the majority of Infrastructure as a Service (I.a.a.S.) platforms no longer offers physical machines but virtualized machines (VMs). Their thriving success greatly relies on the fact that virtual resources are as good as bare metal resources when it comes to ensuring a given level of quality of service (QoS). Cloud providers can spatially multiplex a big physical machine into smaller VMs which are easier to sell. For example, when users A and B only need half a machine, they can use two half-sized VMs mapped on the same physical machine. Moreover, as few users fully exploit their virtual resources at the same time, physical resources can be temporally multiplexed into virtual resources that cannot be used at the same time. For example, if A needs a machine in the morning and B needs one in the evening, they can use two fully-sized virtual machines mapped on the same physical machine. As long as users' needs do not overlap in time, they can use the same physical resources to host their virtual resources. Both of these techniques strive to minimize the amount of unused physical resources, resulting in the maximum return on investment. In the remainder of this article, spatial multiplexing will be referred as isolation and temporal multiplexing as consolidation.

Hypervisors excel at ensuring isolation but they showed their limit in terms of consolidation. Indeed, consolidation requires transfer phases where the physical resource has to be remapped from a virtual resource that is no longer used to another virtual resource that needs to be used. Even if hypervisors are able to efficiently consolidate vCPUs, they still struggle at quickly transferring unused memory between VMs. Some techniques to consolidate memory have emerged (Ballooning [1], PUMA [2]) but they are hard to implement and often intrusive (modification of the guest OS), slow (require synchronization between VMs) and manual (the automation proposals are still prototypes [1]). More recently, lightweight container-based virtualization solutions have begun to emerge. Indeed, they simplify the deployment and limit the extra cost of the virtualization. More importantly, they ensure a good level of isolation and speed up transfers of resources, especially that of memory.

Containerization opens up a wide range of multiplexing opportunities that were not accessible through machine virtualization. Apprehending how far that boundary can be pushed is a difficult task because one must not only be able to predict the needs of individual users, but he must also be able to aggregate their needs to fit the infrastructure's capacity. We decided to study these multiplexing opportunities through the lens of a model. In our model, users are classified into two classes. They are either **active**, meaning that they need all the resources they asked for, or **inactive**, meaning that they do not need any of their resources. The aggregate resource needs of active users do not exceed the physical capacity. Users can switch from one state to the other but they never partially need their resources.

We tested Linux containers against this model and discovered that performance degradation occurred in active containers during the transfer phases, *i.e.* when a container is activated

while an other one is deactivated. The performance degradations in active containers were caused by a temporary loss of memory. When an inactive container is activated, it requests its memory back, but the Linux kernel wrongly reclaims overused memory in active containers even though, underused memory is available in a recently deactivated container.

The purpose of this paper is (*i*) to demonstrate that consolidation deteriorates performance, (*ii*) to understand why, and (*iii*) to propose a new approach allowing containers to truly offer consolidation without performance loss.

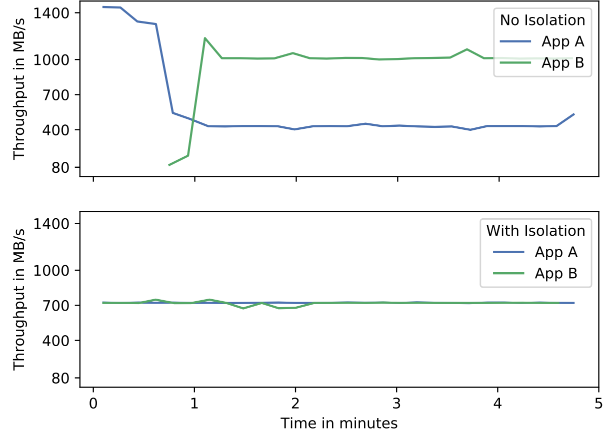
A thorough analysis of the Linux kernel code as well as a series of experiments on real-world applications and microbenchmarks have allowed us to highlight an intrinsic problem in the management of memory in the kernel. Indeed, isolation was implemented with a partition of the metadata related to the memory management. There used to be a global least recently used (LRU) order on memory pages, but now, the system maintains one LRU order per container. During consolidation, it becomes impossible to compare the recency of pages when they are not stored in the same LRU list. Therefore, instead of targeting deactivated containers, Linux reclaims memory in **all** containers, and in the process, degrades performance of active containers. Based on this observation, we proposed a new memory management algorithm to approximate a total activity order on containers.

We have implemented inside the kernel a new approach that tracks the container that has made a memory demand the least recently. It is non-intrusive with respect to sensitive structures and functions, and has a negligible overhead during the steady phases.

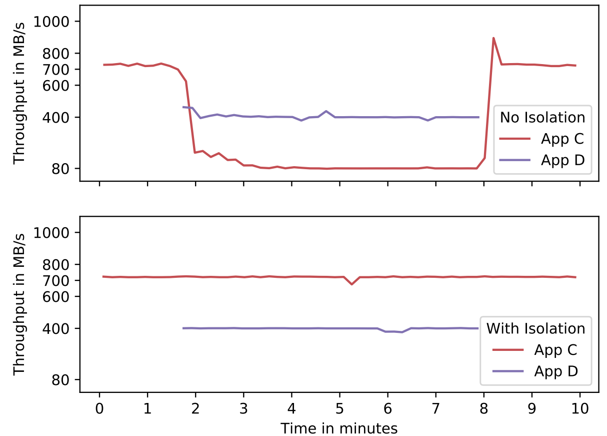
The main contributions of this work are:

- A reproducible experiment that highlights the problem with real applications used by the Google Perfkit [3]: MySQL [4] and Cassandra [5]. (see Section II-C and Figure 10).
- A synthesis of a careful analysis of the Linux kernel code grasping the root of the problem. (see Sections II-B III-B)
- The design and implementation of a kernel-level approach providing the ability to approximate a total order on containers according to the time of their last memory demand. (see Section IV).
- The evaluation of this solution, with MySQL [4] and Cassandra [5], showing that our approach can avoid QoS failures. (see Section V).

The remainder of this article is organized as follows. Section II presents a short background on containers and the reproducible experiment that highlights the QoS loss during consolidation, then Section III synthesizes our understanding of how the memory management mechanisms of the Linux kernel lead to the aforementioned issue. Section IV presents our solution and Section V evaluates it. Finally, Section VI presents the related works and Section VII concludes and gives some future work directions.



(a) Block I/O Isolation protects A from B



(b) Memory Isolation protects C from D

Fig. 1: Isolation is required because applications compete for resources

II. Background and Problem Highlighting

Isolation is a must-have property which relies on the cgroup kernel feature (see Section II-A). In particular, the memory cgroup feature allows users to control the memory consumption of their containers (see Section II-B). But during consolidation, containers fail to provide an acceptable QoS (see Section II-C).

A. Isolation is a must-have property

Applications sharing the same system tend to compete for resources because they were not designed to take into account possible impacts on other applications when asking for resources. To illustrate this harmful competition, we designed four Filebench [6] applications that access private data on the same disk and we measured their throughput as a performance metric. A (respectively C) and B (respectively D) are first

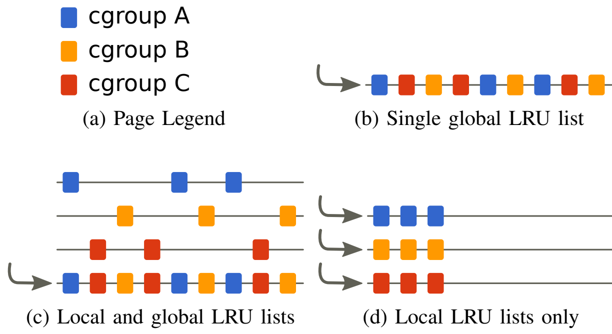


Fig. 2: Evolution of the LRU list in the Linux kernel

deployed without isolation and then with isolation. The difference between App A and B is that A spawns a single process while B spawns two processes. As the system imposes block I/O bandwidth fairness between processes, we can observe on the upper plot of Figure 1a that A is overtaken by B when there is no block I/O bandwidth isolation.

As the Linux kernel uses free RAM to cache data in memory, competition also occurs on this resource. The App D often changes its small workingset—*i.e.*, its set of most recently used data—but App C has a big static workingset. In the absence of memory isolation, the eviction policy of Linux cannot detect the dynamics of these workingsets. We can deduce on the upper plot of Figure 1b that the data of C was evicted by the data of D.

Cgroups, abbreviated from control groups, were developed to provide mechanisms for accounting and limiting resources (CPU, memory, I/O, network ...). In both cases, thanks to cgroup's isolation, the applications can be deployed on the same machine.

B. Memory Cgroup Subsystem

The Linux kernel tracks the utility of memory pages by maintaining a set of lists—a.k.a. the LRU list—which follows the least recently used eviction policy. In order to enforce isolation, the memory controller must be able to reclaim pages belonging to a specific cgroup whenever it reaches its limit. If no local page lists per cgroup were used and only one global set of lists was used, it would take a significant amount of time to reclaim pages of a specific cgroup because pages of all the other cgroups would have to be filtered out. For example, on Figure 2b, reclaiming the last page of A would require to go through the two pages of B and C first.

Before J. Weiner's work [7], cgroups were implemented with local lists "bolted" on the global ones (see Figure 2c). These additional lists allowed the kernel to quickly iterate through pages belonging to a specific cgroup by skipping pages from other cgroups, but J. Corbet reported that this design had at least three disadvantages [8]. First, the global reclaim did not take into account the cgroup limit policy. Second, to keep duplicated structures synchronized, only one cgroup could be reclaimed at a time but this created interference between cgroups. Third, duplicated structures not only added complexity in the code but also increased the memory footprint. As a result, J. Weiner removed the global set of lists and split it into local set of lists per cgroups (see Figure 2d).

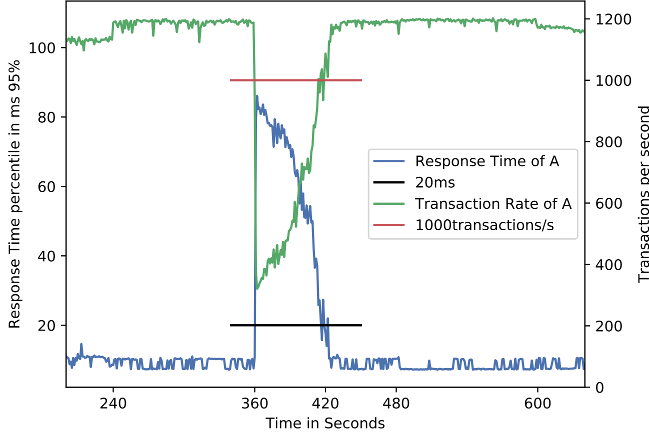
C. QoS Loss during consolidation

The activity model exposed in the introduction was inspired by a business pattern observed at Magency [9]. This company sells collaborative applications which target meetings, trainings and corporate events. The workload in this environment is very heterogeneous: during an event, the application is always active, but before and after the event, the application often changes its activity status because it is sporadically accessed to upload or download content. Magency wanted to use containers as a means to consolidate the resources of applications when they often change their activity. But during events, the applications encountered momentary slowdowns because they were running out of memory. Yet much of the memory was unused because allocated to other inactive applications.

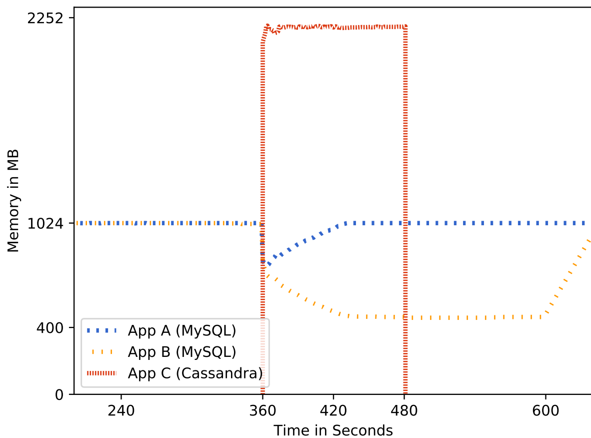
We have reproduced in laboratory the consolidation problem encountered at Magency with real applications that are MySQL (App A and B) and Cassandra (App C) by using the following scenario. A, B and C are deployed on a physical machine with enough memory so that only two of them can be active at the same time. A models an application during an event. It is active throughout the experiment $[0s, 840s]$ but takes $100s$ to warm up. B and C model applications that change their activity status, their resources are temporally multiplexed. B is active at the beginning $[0s, 240s]$ and becomes inactive in the middle $[240s, 600s]$, then it becomes active again until the end $[600s, 840s]$; C activates itself at the middle of the experiment $[360s, 480s]$ in the time frame in which B is inactive.

All the experiments in this paper were done on an Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz with two memory banks HMT351U6EFR8C-PB of 4GB and a Samsung SSD 840 of 128GB. The version of Linux kernel was 4.6.0 compiled with gcc 4.8.4. A and B were given two cores, 1GB of memory and 12MB/sec of bandwidth to the SSD. C was allowed to execute on B's cores and had to reuse about 512MB of B's memory. The queries to MySQL were generated using Sysbench [10] and we expected that 95% of the requests had to be executed in less than 20ms. The expected transaction rate had to be at least 1000 transactions per second. This QoS is fairly achievable on our machine and better results have been obtained by Felter *et al.* when they compared VMs to containers on a 16 cores machine with 256GB of RAM [11].

When the Cassandra application (App C) starts at time $360s$, the QoS of A is degraded by a factor of 4 during one minute. The 95%tile response time of the queries of A does not respect the expected level of 20ms and the transaction rate drops below a 1000/sec (see Figure 3a). Indeed, the memory of B is reclaimed and transferred to C, but despite being active, A loses about 256MB of memory (see Figure 3b). A is then forced to reload its data which causes more memory to be reclaimed; some from B and some from C. All these extra



(a) The quality of service of A is degraded when C boots.



(b) As C boots, memory is taken from the wrong container—i.e. A, the active container—when it should have been taken from B only.

Fig. 3: Reproduction of QoS losses encountered at Magency

transfers explain the QoS Loss in A during consolidation. The next Section investigates why memory is erratically transferred during consolidation.

III. Problem Analysis

Using a microbenchmark, we demonstrate that the QoS Loss described in Section II-C is indeed related to the fact that the applications had to be deployed in containers (see Section III-A). A thorough analysis of the Linux kernel code has allowed us to highlight why containers fail to ensure QoS during temporal consolidation phases (see Section III-B).

A. Are containers responsible?

We wanted to validate the hypothesis that the problem—*i.e.* the erratic transfers of memory during consolidation—was solely due to the fact that the applications were deployed in containers. Consequently, we crafted a microbenchmark to model A, B and C such that they could be easy to study out of containers. We replaced A and B with Filebench [6] processes that did not need isolation because we throttled them using Filbench's workload model language [12]. C was replaced with a simple program written in C which allocates memory through a single `malloc` call, accesses it in loop and finally calls `free` at the end.

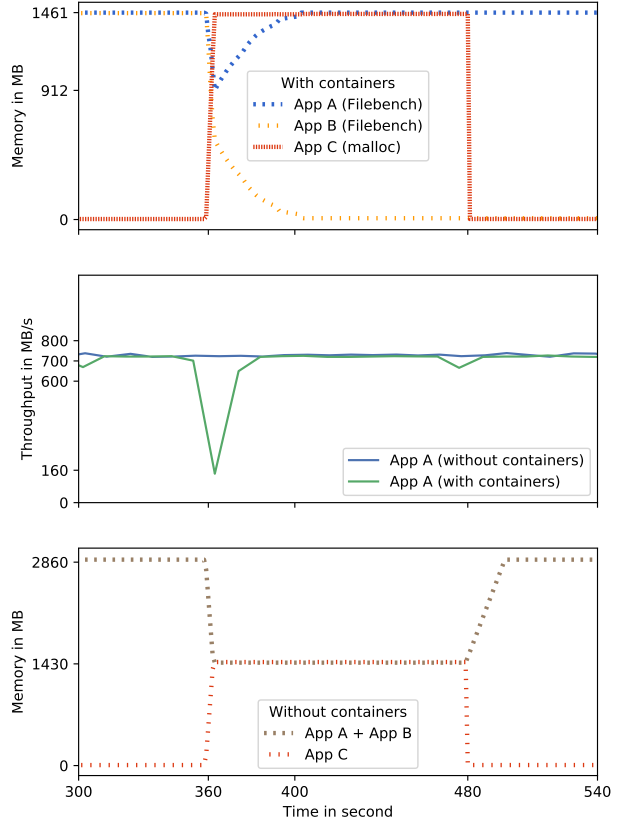
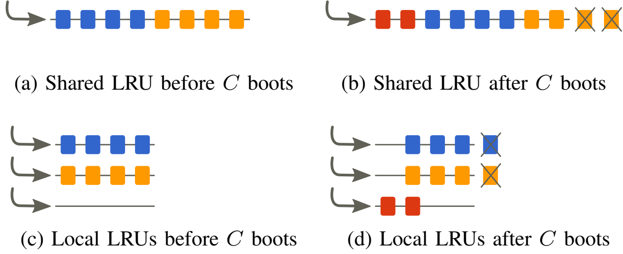


Fig. 4: Running Filebench processes in and out of containers

We repeated the scenario described in Section II-C twice: first by deploying the applications in containers and then by deploying them without containers. The results are reported in Figure 4. With containers, we observed a QoS Loss in A because its memory was collected along with B's when C booted. The disturbance lasted for $40s$ and the read throughput dropped from 700MB/sec to 160MB/sec. When deployed out of containers, the QoS of A was not impacted by the start-up of C because the only memory reclaimed was that of B[1]. We concluded that Linux containers were indeed responsible for this problem and that the Linux kernel code had to be

[1]The memory of A and B was measured with the *cache* counter and that of C was measured with the *anon*.



(a) Shared LRU before C boots (b) Shared LRU after C boots

(c) Local LRUs before C boots (d) Local LRUs after C boots

Fig. 5: Evictions in LRUs: A's pages are blue, B's are yellow and C's are red.

analyzed.

B. LRUs during consolidation

When a new page is demanded, the kernel first controls if the cgroup is allowed to grow. As cgroups can be nested, the Linux kernel recursively checks if the leaf and the internal nodes of the cgroup tree are not exceeding their isolation limit. This logic is described in the *mm/memcontrol.c* file. When a leaf reaches its limit, a local reclaim is triggered on that specific leaf only. But when an internal node reaches its limit, a hierarchical reclaim is triggered on all its leaves. During reclaims, the LRU page lists are scanned and trimmed to recycle pages. This process is known as the PFRA (Page Frame Reclaiming Algorithm) and is described in the *mm/vmscan.c* file. Our understanding of the code can be summarized as follows:

When applications are deployed without isolation, they share the same cgroup, *i.e.*, the same set of LRU page lists. Thanks to this sharing, the LRU order of their pages can be compared (see Figure 5a): the PFRA knows that the pages of B are less recently used than that of A. Consequently, the PFRA correctly evicts the pages of B and keeps the pages of A (see Figures 5b).

But when isolation is required between applications, there is no global LRU order because the kernel maintains a local set of LRU page lists per cgroups (as explained in Section II-B). The PFRA does not know that the pages of B are less recently used than that of A since the LRU order of pages from different cgroups cannot be compared (see Figure 5c). As a result, when C boots, the PFRA chooses to evict pages from both B and A (see Figures 5d). Based on this observation, we proposed a new memory management algorithm to approximate a total order on cgroups.

IV. ACDC

ACDC addresses an industrial problematic, the ultimate goal is at least to have our approach used in production at Magency to fix the QoS loss issue, and eventually to contribute to the Linux kernel. This imposes several design constraints on our approach (Section IV-A): it has to be non-intrusive with respect to sensitive structures and functions in order to generate only a negligible overhead during the steady phases (Section IV-B). Unfortunately, ACDC cannot solve all consolidation scenarios because it would require a non-negligible overhead during the steady phases (Section IV-C).

A. Constraints

As explained in Section II-A, containers rely on the kernel to offer isolation. Therefore, if containers have to improve their consolidation, some changes have to be done in kernel mode. We thus decided to implement our new approach inside the kernel. If the solution was implemented in user mode, it would not be as reactive as a kernel mode solution. Indeed, we want to transfer memory at the very last moment when the page demand occurs. This approach should allow us to handle unpredictable consolidation events.

In Section II-B, we mentioned previous work that has shown that the LRUs have to be strictly disjointed to preserve an isolated behavior during steady phases. We therefore decided to modify neither the state of LRUs nor the update mechanism of the LRUs. In Section III-B, we showed that it is hard to compare the utility of pages when they are stored in different LRU lists, but we also showed that the key to successfully preserve QoS during consolidation phases is to correctly reclaim unaccessed pages first, before considering useful pages.

B. ACDC behavior

Given the aforementioned constraints, we came to the conclusion that memory had to be taken from the container which would ask it back at the latest. We needed an oracle to predict the container whose next page demand will occur the farthest in the future. During the consolidation phase, this container would be targeted in priority by the PFRA in order to protect the memory of the containers which are actively demanding memory. We have implemented an approximation of the total order on containers with the following heuristic: the container whose next page demand will occur the farthest in the future is the container whose last page demand occurred the least recently. In a nutshell, ACDC reclaims the least recently used pages of the least recently upsized container.

Most of ACDC's logic is implemented in *mm/memcontrol.c*. It adds a global clock which is atomically increased every time a new page is charged into a cgroup (compute complexity: $\Theta(1)$ during steady phases). A timestamp per cgroup stores the last time a page was charged to the cgroup (memory complexity: $\Theta(n)$ where n is the number of cgroups). When pages have to be recycled, unlike the current strategy, ACDC does not try to reclaim memory in all cgroups. It carefully triggers the PFRA on one cgroup at a time and stops when enough memory has been reclaimed. The cgroups are ran down according to the approximated recency order (compute complexity $O(n \log n)$ during consolidation phases). The PFRA also had to be modified to focus on the targeted cgroup only.

C. ACDC's limit

The LRU list in the Linux kernel has a specific design which reduces unnecessary page movements in it. As a result, when

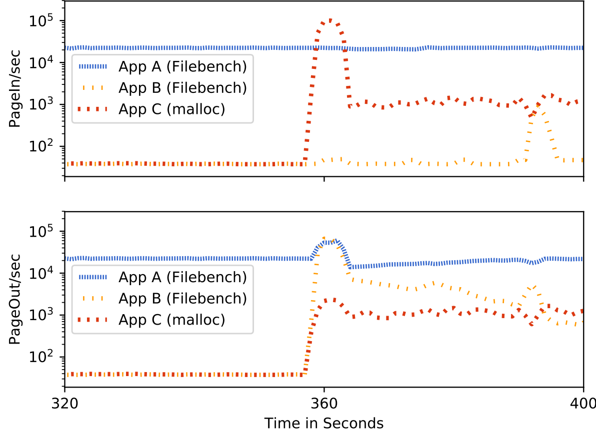


Fig. 6: Without ACDC pages are collected in all containers when C demands pages

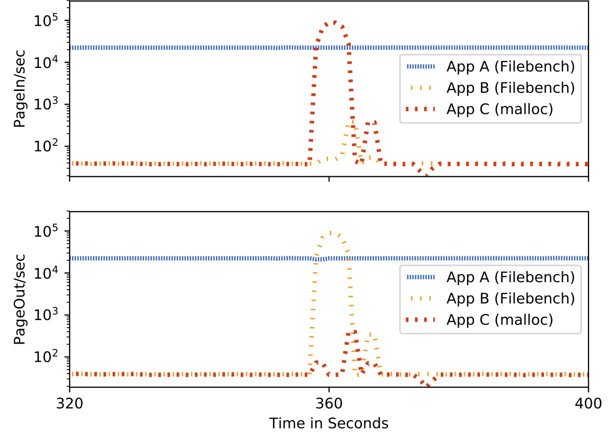


Fig. 7: With ACDC, B is detected as inactive and its pages are the firsts to be collected when C demands pages

applications can fit their data in memory, the state of the LRU list does not change and all the pages are set as active. The flaw in this design is that we cannot make the difference between an application which is still using all of it active pages and an application that stopped accessing part of its active pages.

ACDC inherits the same limitation and assumes that if a container fits its data in memory and stops asking for memory, then it might be under-using its memory. Therefore, during the consolidation phase, ACDC will target the container to check if that assumption was correct. Nevertheless, if ACDC mistakenly reclaims its pages, as soon as the container asks them back it will be protected.

ACDC assumes that there exists a total order on containers that is built on memory demands. But containers could all be demanding pages at a similar rate. This scenario could be solved by trying to predict if the new requested pages are going to be useful. Right now, ACDC does not solve the two aforementioned issues because it would require more resources to compute a solution.

V. Evaluation

ACDC was evaluated with the microbenchmark described in section III-A and with the benchmark described earlier in section II-C.

A. Microbenchmark

In order to observe ACDC's mechanisms, we measured the rate at which pages are allocated (PageIn) and collected (PageOut) (see Figure 6). At time $360s$, C starts-up and asks for memory (single big PageIn/sec spike). Without ACDC, we can observe that the PFRA is triggered in all containers (PageOut/sec spikes) despite that A needs its memory and that B does not (which confirms the earlier results in section III-A).

On the other hand, ACDC takes advantage of the fact that B demands pages at a very small rate compared to A. Indeed, in these circumstances, B is more likely to be the container

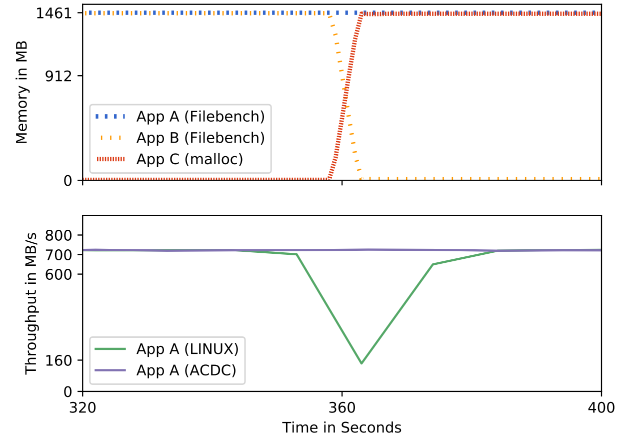
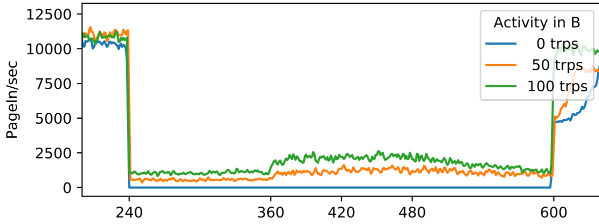


Fig. 8: Evaluation of ACDC with Filebench and *malloc*

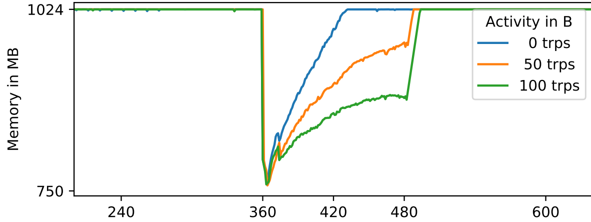
whose last page demand occurred the least recently. Thanks to its oracle (recall section IV-B), ACDC is able to predict that B is the container whose next page demand will occur the farthest in the future. Therefore, when C kicks in at time $360s$, ACDC cautiously triggers the PFRA in B as much as possible (single big PageOut/sec spike in Figure 7). ACDC avoids the QoS loss of Linux: it protects the memory of A and preserves its QoS (see Figure 8).

B. Benchmark

In section V-A, the measurements have shown that ACDC reclaims pages in the container that demands pages at the lowest rate. This section now focuses on what happens when B increases the rate at which it demands pages. As our activity model assumed that memory was never partially needed, we wanted to evaluate if ACDC could still deliver an acceptable performance outside of this assumption. We reused

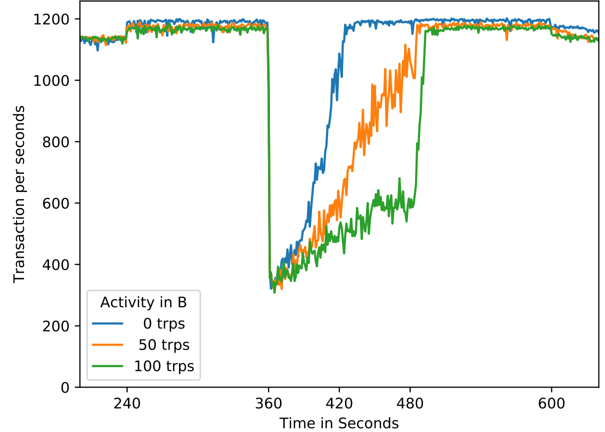


(a) Memory demand rate of B

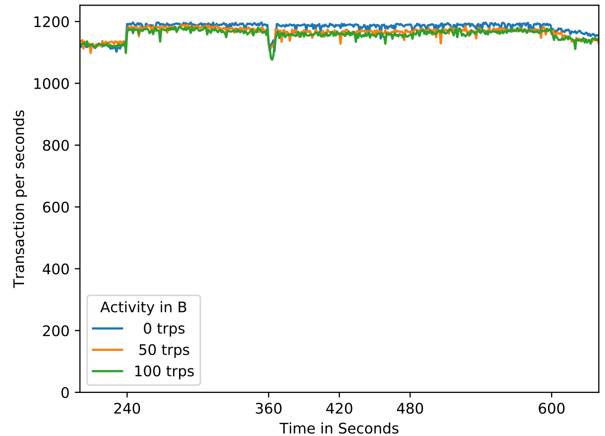


(b) Memory of A

Fig. 9: A's recovery slows down when B's activity increases



(a) Quality of Service of A with Linux



(b) Quality of Service of A with ACDC

Fig. 10: Evaluation of ACDC with MySQL and Cassandra

the MySQL and Cassandra scenario described in section II-C. In that scenario, B was receiving no transaction between $[240s, 600s]$. We ran that same scenario multiple times and varied the transaction rate to simulate different intensities of idleness of B. The PageIn rate of B is presented in Figure 9a and we observed that it is equal to about ten times the transaction rate of B when it has a full $1GB$ of memory.

When the transaction rate in B increases (from 0 to 100 per second), it becomes harder for A to recover its memory because Linux keeps triggering the PFRA in all containers and prevents A from reloading its data (see Figure 9b). Consequently, the QoS loss is extended until the shutdown of C at $480s$ (see Figure 10a. Due to lack of space, the response time percentile degradation was omitted).

As long as B remains the container with the lowest page demand rate, ACDC can handle the consolidation and does not generate severe QoS losses in A because the PRFA is mostly triggered in B (see Figure 10b). Unfortunately, ACDC is only an approximation, therefore when B's activity increases, its likelihood of not being the container whose last page demand occurred the least recently also increases. We measured that A lost 3% of its memory which is not enough to threaten its QoS (compared to Linux's 27%).

VI. Related Work

Improving resource usage through overcommitment is an important challenge of the Cloud Computing [13], [14]. Some works have attempted to consolidate the memory of VM such as Ballooning [15], Autoballooning [1], PUMA [2] and Bayllocator [16], but transfers between VMs are much heavier than transfers between containers. The problem is thus different, requiring less dynamicity.

Cloud providers offer "autoscaling" as a convenient means to resize resources allocated to hosted applications [17]–[20]. Sedaghat *et al.* showed that both horizontal (number of instances) and vertical (size of instances) scaling had to be taken into account during the resource reconfiguration [21]. ACDC can be applied to both methods to avoid QoS losses in containers during the reconfiguration.

Google, which massively uses containers, also seeks to automate the management of resources thanks to its container management systems (Borg [22], Omega [23] and Kubernetes [24]). Oversubscription is intensively used for work that can tolerate lower-quality resources and this practice has spread to other resource managers such as Mesos [25]. Borg have a user-space control loop that assigns memory to containers based on predicted future usage or on memory pressure. But we think that for better reactivity, part of these decisions should take place in the kernel. Indeed, ACDC is able to transfer memory at the very last moment when the page demand occurs.

Zhenyun Zhuang has experienced similar performance pit-

falls at LinkedIn on machines where memory was overcommitted [26]. By default, if containers are not finely tuned by the user, the current memory management will hurt the performance of the applications running in containers during the global memory pressure scenarios. We believe that ACDC might have avoided some of the pitfalls encountered in this paper. For example, Zhenyun Zhuang suggested to pretouch memory pages at boot. With ACDC, a container is protected whenever it demands a new memory page because it becomes the container that has made a memory demand the most recently. Therefore, with ACDC, applications can still use the demand-as-you-go resource model.

Vladimir Davydov highlighted an open problem in Linux memory management which is much harder to solve than the one described in this paper [27]. When all containers are actively demanding pages, the container whose demand rate is the highest can outrun the others, and hog most of the memory. He stated that this behavior was unfair, especially in the case where the former container reclaims useful pages from others in favor of useless pages, *i.e.* pages that are used once and never used again. This problem is still open but some solutions were discussed, one of them proposed to store the time at which **each** page was added to the LRU list, and to track the oldest page on each list. The proposed solution could then try to achieve an approximate balance of ages. Even if ACDC does not solve this problem, it shows that storing the age of a **single** page (the most recent page added to the LRU list) can preserve the QoS in our model of multiplexing scenarios.

VII. Conclusion and Future work

This work has shown that in order to avoid QoS losses, consolidation must not reclaim overused resources if there are underused resources. We have proposed a simple and reproducible benchmark scenario which shows that Linux currently degrades the QoS because it reclaims useful pages even if there exist unaccessed pages during consolidation phases. We understood that it is hard to compare the utility of pages when they are stored in different LRU lists. We proposed an approach which measures the activity of the containers based on the time of the last memory demands. Our evaluations show that it is possible to guarantee the performance of the most active containers during consolidation.

Unfortunately, our approach cannot detect active containers if they do not frequently demand memory. Therefore, if an active container fits its data in memory and stops asking for more memory, it will be detected as inactive. Nevertheless, if ACDC mistakenly reclaims its pages, as soon as the container asks them back it will be protected. Moreover, if containers are all demanding pages, ACDC will not try to predict if the new requested pages are going to be useful. In our ongoing work, we are trying to address these issues by monitoring the distribution of memory accesses.

References

[1] L. Capitulino, “Automatic memory ballooning,” 2013. [Online]. Available: http://www.linux-kvm.org/images/5/58/Kvm-forum-2013-automatic-ballooning.pdf

[2] M. Lorrillere, J. Sopena, S. Monnet, and P. Sens, “Puma: Pooling unused memory in virtual machines for i/o intensive applications,” in *Proceedings of the 8th ACM International Systems and Storage Conference*, ser. SYSTOR ’15. New York, NY, USA: ACM, 2015, pp. 1:1–1:11. [Online]. Available: http://doi.acm.org/10.1145/2757667.2757669

[3] Google, “Perfkit benchmarker.” [Online]. Available: https://github.com/GoogleCloudPlatform/PerfKitBenchmarker

[4] “Mysql docker image.” [Online]. Available: https://hub.docker.com/_/mysql/

[5] “Cassandra docker image.” [Online]. Available: https://hub.docker.com/_/cassandra/

[6] “Filebench : a file system and storage benchmark.” [Online]. Available: https://github.com/filebench/filebench

[7] J. Weiner, “memcg naturalization.” [Online]. Available: https://lwn.net/Articles/442615/

[8] J. Corbet, “Integrating memory control groups.” [Online]. Available: https://lwn.net/Articles/443241/

[9] “Magency : Engaging and collaborative digital solutions for your business.” [Online]. Available: http://magencydigital.com/

[10] A. Kopytov, “sysbench: Scriptable database and system performance benchmark.” [Online]. Available: https://github.com/akopytov/sysbench

[11] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, “An updated performance comparison of virtual machines and linux containers,” in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*. IEEE, 2015, pp. 171–172.

[12] “Filebench workload model : describe desired workloads from scratch.” [Online]. Available: https://github.com/filebench/filebench/wiki/Workload-model-language

[13] L. Tomás and J. Tordsson, “Improving cloud infrastructure utilization through overbooking,” in *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, ser. CAC ’13. New York, NY, USA: ACM, 2013, pp. 5:1–5:10. [Online]. Available: http://doi.acm.org/10.1145/2494621.2494627

[14] D. Williams, H. Jamjoom, Y.-H. Liu, and H. Weatherspoon, “Overdriver: Handling memory overload in an oversubscribed cloud,” *SIGPLAN Not.*, vol. 46, no. 7, pp. 205–216, Mar. 2011. [Online]. Available: http://doi.acm.org/10.1145/2007477.1952709

[15] C. A. Waldspurger, “Memory resource management in vmware esx server,” *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 181–194, Dec. 2002. [Online]. Available: http://doi.acm.org/10.1145/844128.844146

[16] V. Tasoulas, H. Haugerud, and K. Begnum, “Bayllocator: A proactive system to predict server utilization and dynamically allocate memory resources using bayesian networks and ballooning.” San Diego, CA: USENIX, 2012, pp. 111–121.

[17] Amazon, “Aws auto scaling.” [Online]. Available: http://aws.amazon.com/autoscaling/

[18] OpenStack, “Heat auto scaling.” [Online]. Available: https://wiki.openstack.org/wiki/Heat

[19] “Docker cloud : A hosted service for docker container management and deployment.” [Online]. Available: https://cloud.docker.com

[20] “Rancher : a complete platform for running docker applications in production.” [Online]. Available: http://rancher.com/rancher/

[21] M. Sedaghat, F. Hernandez-Rodriguez, and E. Elmroth, “A virtual machine re-packing approach to the horizontal vs. vertical elasticity trade-off for cloud autoscaling,” in *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*. ACM, 2013, p. 6.

[22] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, “Large-scale cluster management at google with borg,” in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys ’15. New York, NY, USA: ACM, 2015, pp. 18:1–18:17. [Online]. Available: http://doi.acm.org/10.1145/2741948.2741964

[23] Google, “Lessons learned from three container-management systems over a decade,” March 2016.

[24] ——, “kubernetes: managing containerized applications across multiple hosts.” [Online]. Available: https://github.com/kubernetes/kubernetes

[25] Apache, “Oversubscription in mesos.” [Online]. Available: http://mesos.apache.org/documentation/latest/oversubscription

[26] Z. Zhuang, C. Tran, J. Weng, H. Ramachandra, and B. Sridharan, “Taming memory related performance pitfalls in linux cgroups,” in *Computing, Networking and Communications (ICNC), 2017 International Conference on*. IEEE, 2017, pp. 531–535.

[27] J. Corbet, “Memory control group fairness.” [Online]. Available: https://lwn.net/Articles/684926/