# YOLO: Speeding up VM and Docker Boot Time by reducing I/O operations

Thuy Linh Nguyen, Ramon Nou, Adrien Lebre

# YOLO: Speeding up VM and Docker Boot Time by reducing I/O operations

Thuy Linh Nguyen[1], Ramon Nou[2], and Adrien Lebre[1]

[1] IMT Atlantique, Inria, LS2N, France
{thuy-linh.nguyen,adrien.lebre}@inria.fr
[2] Barcelona Supercomputing Center (BSC), ramon.nou@bsc.es

**Abstract.** Although this comes as a surprise, the time to boot a Docker-based container can last as long as a virtual machine in high consolidated cloud scenarios. Because this time is critical as boot duration defines how an application can react w.r.t. demands' fluctuations (horizontal elasticity), we present in this paper the *YOLO* mechanism (*You Only Load Once*). *YOLO* reduces the number of I/O operations generated during a boot process by relying on a *boot image* abstraction, a subset of the VM/container image that contains data blocks necessary to complete the boot operation. Whenever a VM or a container is booted, *YOLO* intercepts all read accesses and serves them directly from the boot image, which has been locally stored on fast access storage devices (*e.g.*, memory, SSD, etc.). In addition to *YOLO*, we show that another mechanism is required to ensure that files related to VM/container management systems remain in the cache of the host OS. Our results show that the use of these two technics can speed up the boot duration 2-13 times for VMs and 2 times for containers. The benefit on containers is limited due to internal choices of the docker design. We underline that our proposal can be easily applied to other types of virtualization (*e.g.*, Xen) and containerization because it does not require intrusive modifications on the virtualization/container management system nor the base image structure.

**Keywords:** virtualization, containerization, boot duration

## 1 Introduction

The promise of the elasticity of cloud computing brings the benefits for clients of adding and removing new VMs in a manner of seconds. However, in reality, users may have to wait several minutes to get a new environment in public IaaS clouds [10] such as Amazon EC2, Microsoft Azure or RackSpace. Such long startup duration has a strong negative impact on services deployed in a cloud system. For instance, when an application (*e.g.*, a web service) faces peak demands, it is important to provide additional resources as fast as possible to prevent loss of revenue for this service. DevOps expects that the use of container technologies such as Docker [11] would tackle such issues. However as discussed in this article, provisioning a container can last as long as a VM under high

consolidated scenarios. Therefore, the startup time of VMs or containers plays an essential role in provisioning resources in a cloud infrastructure.

Two parts should be considered for the startup: (i) the time to transfer the VM/container image from the repository to the selected compute node and (ii) the time to perform the boot process. While a lot of efforts focused on mitigating the penalty of the image transferring time for VMs [7,16,17] as well as Docker [6,12], only a few works addressed the boot duration challenge for VMs [8,18,23] and to the best of our knowledge, none for containers. The process to boot a VM (or a container) leads to I/O and CPU operations that should be handled by the compute node. As a consequence, the duration of the boot process depends on the effective system load, in particular, the interference on the I/O path [13,14,21].

To deal with the aforementioned limitation, we investigated in this article the use of cache strategies that allow us to mitigate the number of I/O operations and thus to reduce the boot time. Concretely, we consolidated previous observations which have shown that only a small portion of the image is required to complete the VM boot process [15,17,23]. More precisely, we analyzed the I/O operations that occur during a boot process of a VM and a container. This analysis enabled us to conclude that (i) like VMs, containers only require to access a small part of the image to complete the boot process and (ii) unlike VMs, the amount of manipulated data for a container is much smaller in comparison to the I/O operations performed by the container management system itself.

Leveraging these results, we designed *YOLO* (*You Only Load Once*) as an agnostic mechanism to cache the data of a VM/container image that are mandatory to complete the boot operation: For each VM/container image, we construct a *boot image*, *i.e.*, a subset of the image that contains the mandatory data needed for booting the environment, and store it on a fast access storage device (memory, SSD, etc.) on each compute node. When a VM/container boot process starts, *YOLO* transparently loads the corresponding boot image into the memory and serves all I/O requests directly. In terms of storage requirements, the size of a boot image is in the average of 50 MB and 350 MB for respectively Linux and Windows VMs (storing boot images for the 900+ VM images from the Google Cloud platform would represent 40 GB, which is acceptable to be locally stored on each compute node). Regarding container technologies, the size of a boot image is much smaller with an average of 5 MB. For the I/O operations that are related to the VM/container management system, we simply use the `vmtouch` [4] program that enables to lock specific pages in the Linux system. We underline that using `vmtouch` for boot images is not relevant as it will be not acceptable to populate the cache with all possible boot images.

By mitigating the I/O operations that are mandatory to boot a VM or a container, *YOLO* can reduce the boot duration 2-10 times for VM and 2 times for containers according to the system load conditions.

The rest of this paper is organized as follows. Section 2 gives background elements regarding VM/container boot operations. Section 3 introduces *YOLO*. Section 4 describes the setup for all of our experiments. Section 5 and 6 discuss

the results we obtained. Section 7 deals with related works. Finally, Section 8 concludes the article and highlights future works.

## 2 Background

In this section, we give the background about QEMU-KVM and Docker that we used to perform our analysis, we choose these two virtualization solutions because of their wide used. For each technique, we first describe the boot process so that readers can understand clearly different steps of the boot operation. Second, we discuss the types of virtual disks that can be used in a QEMU/KVM-based or a Docker environment. Finally, we give details regarding access patterns and amount of manipulated data that a VM or a container performed during a boot operation.

### 2.1 QEMU-KVM Virtual Machine

**Boot process** The boot operation of a VM is managed by the QEMU-KVM hypervisor that is in charge of creating the virtual abstraction of the machine (*e.g.*, CPU, memory, disks, etc.) and launching the boot process. The boot process follows the usual workflow: first, the BIOS of the VM checks all devices and tests the system, then it loads the boot loader into memory and gives it the control. The boot loader (GRUB, LILO, etc.) is responsible for loading the guest kernel. Finally, the guest kernel invokes the `init` script that starts major services such as SSH. A QEMU-KVM VM can rely on two different VM Disk as discussed in the following.

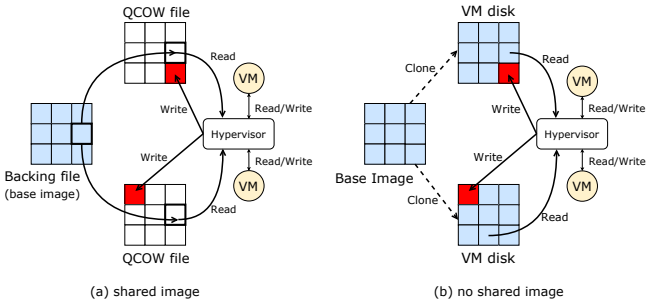
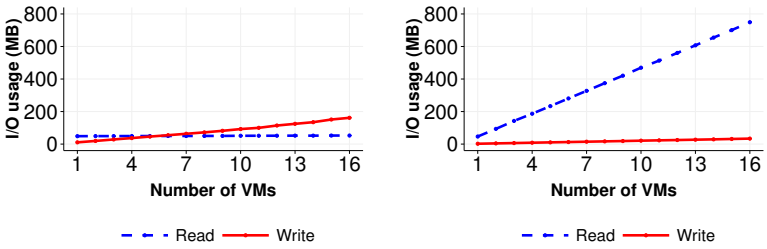



Fig. 1: Two types of VM disk

**VM images** QEMU offers two strategies to create a VM disk image from the VMI (*a.k.a.* the VM base image). For the sake of simplicity, we call them *shared image* and *no shared image* strategies. Figure 1 illustrates these two strategies. In the *shared image* strategy, the VM disk is built on top of two files: the backing

and the QCOW (QEMU Copy-On-Write) files. The backing file is the base image that can be shared between several VMs while the QCOW file is related to a single VM and contains all write operations that have been previously performed. When a VM performs read requests, the hypervisor first tries to retrieve the requested data from the QCOW and if not it forwards the access to the backing file. In the *no shared image* strategy, the VM disk image is cloned fully from the base image and all read/writes operations executed from the VM will be performed on this standalone disk.

**Amount of manipulated data** To identify the amount of data that is manipulated during VM boot operations, we performed a first experiment that consisted in booting up to 16 VMs simultaneously on the same compute node. We used QEMU/KVM (QEMU-2.1.2) as the hypervisor, VMs are created from the 1.2 GB Debian image (Debian 7, Linux-3.2) with *writethrough* cache mode (*i.e.*, each write operation is reported as completed only when the data has been committed to the storage device).




(a) *shared image* disk (b) *no shared image* disk

Fig. 2: The amount of manipulated data during boot operations (reads/writes)

Figure 2 reveals the amount of read/write data. Although the VMs have been created from a VMI of 1.2 GB, booting 1 VM only needs to read around 50 MB from kernel files in both cases of *shared image* and *no shared image*. In addition to confirming previous studies regarding the small amount of mandatory data w.r.t. the size of the VMI [17], this experiment shows that booting simultaneously several instances of the same VM leads to the different amount of manipulated data according to the disk strategy. When the VMs share the same backing file (Fig. 2a), the different boot process benefit from the cache and the total amount of read data stays approximately around 50 MB whatever the number of VMs started (the mandatory data has to be loaded only once and stays into the cache for later accesses). When the VMs rely on different VM disks (Fig. 2b), the amount of read data grows linearly since each VM has to load 50 MB data for its own boot process. Regarding write accesses, both curves follow the same increasing trend. However, the amount of manipulated data differs: the *shared*

*image* strategy writes 10 MB data when booting one VM and 160 MB for booting 16 VMs while the *no shared image* strategy slightly rises from 2 MB to 32 MB. The reason why the *shared image* strategy writes 5 times more data is due to the *"copy-on-write"* mechanism: when a VM writes less than cluster size of the QCOW file (generally 64 kB), the missing blocks should be read from the backing file, modified with the new data and written into that QCOW file [5].

In addition to reading from the base image, the QEMU-KVM process (*i.e.*, the daemon in charge of handling the boot request) has to load into the memory a total of 23 MB. This amount of data correspond to host libraries and the QEMU binary file. The write operations performed by the QEMU-KVM process are negligible (a few KBytes).

### 2.2 Docker Container

**Boot process** Although we use the words *Docker boot process* in comparison with the virtualization system terminology, it is noteworthy that a Docker container does not technically boot, but rather start. Booting a docker starts when the *dockerd* daemon receives the container starting request from the client. After verifying that the associated image is available, *dockerd* prepares the container layer structure, initializes the network settings, performs several tasks related to the specification of the container and finally gives the control to the *containerd* daemon. *containerd* is in charge of starting the container and managing its life cycle.

**Docker images** From the storage viewpoint a docker container is composed of two layers: the image layer and the container layer (*a.k.a.* the *lowerdir* and *uppperdir* files). These two layers can be seen as the backing and COW files in the VM terminology. The image layer is a read-only file that can be shared between multiple containers. The container layer contains differences w.r.t. the base image for each container. The unified view of the two directories is exposed as the *merged* union mount that is mounted into the container thanks to the *overlayfs* file system. This file system implements the copy-on-write strategy.

**Amount of manipulated data** Although the order of magnitude differs, the amount of manipulated data when booting several times the same container follows the same trend of VMs sharing the same backing file: thanks to the cache, the amount of read data is constant. However, at the opposite of VMs, we observed that the significant part of read accesses when booting one container is related to the host directories and not the docker image. In other words, loading the docker binaries (*docker*, *docker-containerd-shim* and *docker-runc*), their associated libraries and configuration files represent much more Bytes than the I/O accesses that are performed on the docker image. Table 1 gives the details for different kinds of containers. Regarding the write operations, they are related to the creation of the container layer and the union mount. Although this amount is not significant w.r.t read operations, we noticed that the creation of

the merge union mount point is a synchronous process: the docker daemon has to wait the completion of this action before progressing in the boot process. This is an important point as the more competition we will have on the I/O path, the longer will be the time to start the container.

Table 1: The amount of read data during a docker boot process

| | Host OS | Docker image |
|---|---|---|
| debian | 62.9 MB | 3.7 MB |
| ubuntu | 62.6 MB | 4.1 MB |
| redis | 61.8 MB | 8.2 MB |
| postgres | 60.1 MB | 24.4 MB |

## 3 YOLO Overview

Booting a VM or a container leads to a significant number of I/O operations. Because these operations can interfere with each other, in particular, in high consolidated scenarios, it is critical to mitigate them as much as possible. For such a purpose, we implement *YOLO* as a first mechanism to limit the impact of I/O operations related to the VM or container image on the boot duration. In the following, we give an overview of *YOLO* foundations and its implementation. First, we explain how boot images are created. Second, we introduce how *yolofs*, our custom file system, intercepts I/O requests to speed up a boot process.

### 3.1 *YOLO* Boot Image

*YOLO* relies on the boot image abstraction, *i.e.*, a subset of the VM (or container) image that corresponds to the data mandatory to complete the boot operation. To create a boot image, we capture all read requests generated when we boot completely a VM (or respectively a container). Each read request has: (i) a *file_descriptor* with *file_path* and *file_name*, (ii) an *offset* which is the beginning logical address to read from, and (iii) a *length* that is the total length of the data to read. For each read request, we calculate the list of all *block_id* to be read by using the *offset* and *length* information and we record the *block_id* along with the data of that block. In the end, a boot image contains a dictionary of key-value pairs in which the key is the pair (*file_name*, *block_id*) and the value is the content of that block. Therefore, with every read request on the VM (or container) image, we can use the pair (*file_name*, *block_id*) to retrieve the corresponding data of that block.

To avoid generating I/O contention with other operations, boot images should be stored on dedicated devices for *yolofs*, which can be either local storage devices (preferably SSD), remote attached volumes or even memory. To give an order of magnitude, we created the boot images for 900+ available VMIs from Google

Cloud and the result depicts that the space needed to store all these boot images is around 40 GB, which is less than 3% of the original size of all VMIs (1.34 TB). Storing such an amount on each compute node looks to us an acceptable tradeoff.

### 3.2 *yolofs*

To serve all the read requests from the boot image instead of the VM or container image, we developed a new FUSE file system, entitled *yolofs*. In addition to not being intrusive, recent analysis [19] confirmed that the small overhead of FUSE for read requests is acceptable.

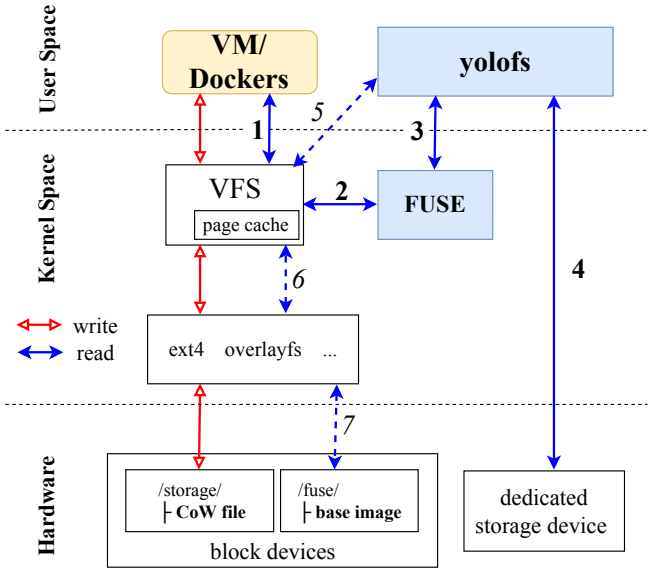



Fig. 3: yolofs read/write data flow

Figure 3 depicts the workflow of *yolofs* along with the read/write data flow for a QEMU-KVM VM or a Docker container. *yolofs* is executed as a daemon on each compute node (that is before any boot operation). When a VM/container issues read operations on its base image, which is linked to our mounted *yolofs* file system, the VFS routes the operation to the FUSE's kernel module, and *yolofs* will process it (*i.e.*, Step 1, 2, 3 of the read flow). *yolofs* then returns the data directly from the boot image (Step 4). If the boot image is not already into the memory, *yolofs* will load it from its dedicated storage device to the memory. Whenever the VM/docker wants to access data that is not available in the boot image, *yolofs* redirects the request to the kernel-based file system to read the data from the disk (Step 5, 6, and 7 of the read flow). Regarding write operations, they are not handled by *yolofs* and are forwarded normally to the corresponding COW file (the write flow in Figure 3).

## 4 Experimental Protocol

In this section, we discuss our experiment setup and scenarios. The code of *YOLO* as well as the set of scripts we used to conduct the experiments are available on public git repositories [1]. We underline that all experiments have been made in a software-defined manner so that it is possible to reproduce them on other testbeds (with slight adaptations in order to remove the dependency to Grid'5000). We have two sets of experiments for both VMs and containers. The first set is aimed to evaluate how *YOLO* behaves compared to the traditional boot process when the VM/container disks are locally stored (HDD and SSD). The second set investigates the impact of collocated I/O intensive workloads on the boot duration.

### 4.1 Experimental Conditions

Experiments have been performed on top of the Grid'5000 Nantes cluster [1]. Each physical node has 2 Intel Xeon E5-2660 CPUs (8 physical cores each) running at 2.2 GHz; 64 GB of memory, a 10 Gbit Ethernet network card and one of two kinds of storage devices: (i) HDD with 10 000 rpm Seagate Savvio 200 GB (150 MB/s throughput) and (ii) SSD with Toshiba PX02SS 186 GB (346 MB/s throughput). Regarding the VMs' configuration, we used the QEMU-KVM hypervisor (Qemu-2.1.2 and Linux-3.2) with *virtio* enabled (network and disk device drivers). VMs have been created with 1 vCPU and 1 GB of memory and a disk using QCOW2 format with the *writethrough* cache mode. For container, we used Docker (18.06.3-ce) with overlay2 storage driver. Each VM/container has been assigned to a single core to avoid CPU contention and prevent non-controlled side effects. The I/O scheduler of VMs and the host is CFQ. We underline that all experiments have been repeated at least ten times to get statistically significant results.

*VM boot time:* we assumed that a VM is ready to be used when it is possible to log into it using SSH. This information can be retrieved by reading the system log, and it is measured in milliseconds. To avoid side effect due to the starting of other applications, SSH has been configured as the first service to be started.

*Docker container boot time:* the main idea behind a container is running applications in isolation from each other. For this reason, docker boot duration is measured as the time to get a service runs inside a docker.

### 4.2 Boot Time methodologies

We considered three boot policies as depicted as follow:

- *all at once*: using a normal boot process we boot all VMs/dockers at the same time (the time we report is the maximum boot time among all VMs/dockers).
- *YOLO*: All VMs/dockers have been started at the same time, and when a VM/docker needs to access the boot data, *YOLO* will serve them. We underline that boot images have been preloaded into the *YOLO* memory

[1] https://github.com/ntlinh16/vm5k

before starting a boot process. This way enables us to emulate a non volatile device. While we agree that there might be a small overhead to copy from the non-volatile device to the *YOLO* memory, we believe that doing so is acceptable as (i) the amount of manipulated boot images in our experiments is just 50 MB for a VM or 5 MB for a container and (ii) the overhead to load simultaneously 16 boot images from a dedicated SSD is less than 1%, as depicted in Figure 4.

– *YOLO* + *vmtouch*: we use *vmtouch* to enforce QEMU and Docker daemon data to stay in the cache before we boot VMs/dockers by using *YOLO*.

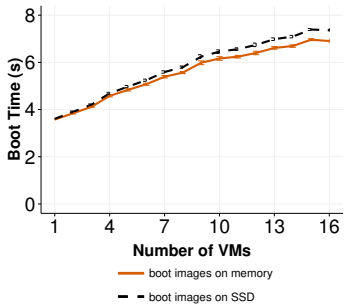


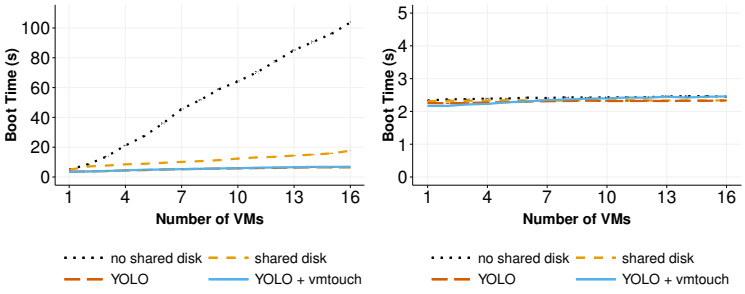

Fig. 4: Overhead of serving boot's I/O requests from the memory *vs.* a dedicated SSD

## 5 VM Boot Time analysis

### 5.1 Booting multiple VMs simultaneously

For the first experiment, we investigated the time to boot up to 16 VMs in parallel using three boot policies mentioned above. With *all at once* policy, we used two different VM disk strategies: *shared image* and *no shared image* (see Section 2). There is no different between these VM disk strategies for *YOLO* because all necessary data for the boot process is already served by *YOLO*. Our goal was to observe multiple VMs deployment scenarios from the boot operation viewpoint.

Figure 5 shows the time to boot up to 16 VMs on a cold environment (*i.e.*, there is no other VMs running on the compute node). On HDD (Figure 5a), the *all at once* boot policy with *no shared image* disk has the longest boot duration because VMs perform read and write I/O operations at the same time for their boot processes on different VM disks. This behavior leads to I/O contentions: the more VMs started simultaneously, the less I/O throughput can be allocated to each VM. Because read operations of boot process access the same backing file for VMs with *shared image* disks, the boot duration is noticeably faster than the VMs with *no shared image* disks. Using *YOLO* speeds up the boot time (from




(a) HDD (b) SSD

Fig. 5: Time to boot multiple VMs with shared and no shared disks

4-13 times) since VMs always get benefit from the cache for reading mandatory data. *YOLO + vmtouch* has basically the same performance as *YOLO* alone because time to load the additional read data for qemu beside boot data from VMI is not significant. On SSD (Figure 5b), the boot time of several VMs is mostly constant for all boot policies. The I/O contention generated during the boot process on SSD is not significant enough to observe performance penalties (the I/O throughput of the SSD is much higher than HDD).

### 5.2 Booting one VM under I/O contention

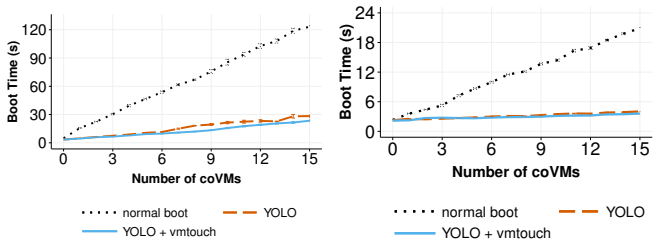
This experiment aims to understand the effect of booting a VM in a high-consolidated environment. We defined two kinds of VMs :

- *eVM* (*experimenting* VM), which is used to measure the boot time;
- *coVM* (*collocated* VM), which is collocated on the same compute node to run competitive workloads.

We measured the boot time of one *eVM* while the $n$ *coVMs* ($n \in [0, 15]$) are running the I/O workloads by using the command `stress`[1]. Each *coVM* utilises a separate physical core to avoid CPU contention with the *eVM* while running the *Stress* benchmark. The I/O capacity is gradually used up when we increase the number of *coVMs*. There is no difference between VMs with *no shared image* and *shared image* disks because we measure the boot time of only one *eVM*. Hence, we simply started one *eVM* with the *normal* boot process.

Figure 6 shows the boot time of one eVM under an I/O-intensive scenario. *YOLO* delivers significant improvements in all cases. On HDD, booting only one eVM lasts up to 2 minutes by using the *normal* boot policy. Obviously, *YOLO* speeds up boot duration much more than the *normal* one because the data is loaded into the cache in a more efficient way. *YOLO + vmtouch* can further improves the boot time by preloading the data for the VM management system.

[1] http://people.seas.harvard.edu/apw/stress/




(a) HDD (b) SSD

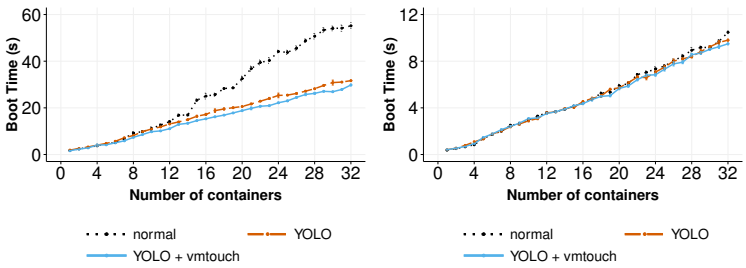
Fig. 6: Boot time of 1 VM (with *shared image* disk, *write through* cache mode) under I/O contention environment

The same trend can be found on SSD in Figure 6b where the time to boot the eVM increased from 3 to 20 seconds for the *normal* strategy, and from 3 to 4 seconds for *YOLO*. *YOLO* is up to 4 times faster than *all at once* policy under I/O contention of 15 coVMs.

## 6 Docker Container Boot Time Analysis

### 6.1 Booting multiple distinct containers simultaneously

Similarly to VMs, we discuss in this paragraph the time to boot several different containers simultaneously. Figure 7 presents the results. Although *YOLO* reduces the time to boot containers, the time increases more significantly in comparison to VMs. This is due to the write operations that need to be completed as explained in Section 2.2. Understanding how such writes can be handled more efficiently is let as future works. Overall, *YOLO* enables the improvement of the boot time by a factor 2 in case of HDD (Figure 7a). The trend for SSD is similar to the VM one: there is not enough competition on the I/O path to see an improvement.

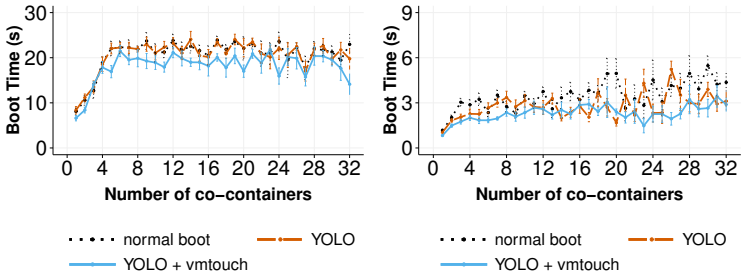



(a) HDD (b) SSD

Fig. 7: Boot time of different docker containers on different storage devices

### 6.2 Booting one docker container under I/O contention

In this paragraph, we discuss the time to boot a container under I/O contention. Figure 8 depicts the results: the boot time is increasing until it becomes quite stable. When a container is started, Docker needs to generate the container layer with all the directories structure for that container. As mentioned, this action generates write operations on the host disk, which suffer from the I/O competition. Although *YOLO* and *YOLO + vmtouch* help mitigate the read operations, Docker still waits for the finalization of the container layer to continue its boot process. Therefore, the gain of *YOLO* is much smaller than for VMs.




(a) HDD (b) SSD

Fig. 8: Boot time of one debian docker container under I/O contention

## 7 Related Work

To improve Docker startup time, most works only tackle the image pulling challenge of Docker because they assume that the container boot time is negligible. However, improving the docker image pulling is out of scope of this article. To the extent of our knowledge, our work is the first one to take into account the boot duration of a Docker container. Meanwhile, there are some solutions that improved the VM boot time, which utilize two main methods: cloning techniques or suspend/resume capabilities of VMs.

Kaleidoscope [2], SnowFlock [9] and Potemkin [20] are similar systems that can start stateful VMs by cloning them from a parent VM. While Potemkin marks a parent VM memory pages as copy-on-write and shares these states to all child VMs, SnowFlock utilises lazy state replication to fork child VMs and Kaleidoscope has introduced a novel VM state replication technique that can speed up VM cloning process by identifying semantically related regions of states. These systems clone new VMs from a live VM so that they have to keep many VMs alive for the cloning process. Another downside is that the cloned VMs have to be reconfigured because they are the exact replica of the original VM so they have the same configuration parameters like MAC address as the original one.

Other works [3,8,22] attempt to speed up VM boot time by suspending the entire state of a VM and resuming when necessary, which leads to a storage challenge. VMThunder+ [23] boots a VM then hibernates it to generate the persistent storage of VM memory data and then use this to quickly resume a VM to the running state. The authors use hot plug technique to re-assign the resource of VM. However, they have to keep the hibernate file in the SSD devices to accelerate the resume process. Razavi et al. [18] introduce prebaked $\mu$VMs, a solution based on lazy resuming technique to start a VM efficiently. To boot a new VM, they restore a snapshot of a booted VM with minimal resources configuration and use their hot-plugging service to add more resources for VMs based on client requirements. However, The authors only evaluated their solution by booting one VM with $\mu$VMs on a SSD device.

## 8 Conclusion

Starting a new VM or container in a cloud infrastructure depends on the time to transfer the base image to the compute node and the time to perform the boot process itself. According to the consolidation rate on the compute node, the time to boot a VM (or a container) can reach up to one minute and more. In this work, we investigate how the duration of a boot process can be reduced. Preliminary studies showed that booting a VM (or container) generates a large amount of I/O operations. To mitigate the overhead of these operations, we proposed *YOLO*. *YOLO* relies on the boot image abstraction which contains all the necessary data from a base image to boot a VM/container. Boot images are stored on a dedicated fast efficient storage device and a dedicated FUSE-based file system is used to load them into memory to serve boot I/O read requests. We discussed several evaluations that show the benefit of *YOLO* in most cases. In particular, we showed that booting a VM with *YOLO* is at least 2 times and in the best case 13 times faster than booting a VM in the normal way. Regarding containers, *YOLO* improvements are limited to 2 times in the best case. Although such a gain is interesting, we claim that there is space for more improvements. More precisely, we are investigating how the creation of the container layer can be performed in a more efficient manner in order to mitigate the dependencies of the write requests with respect to the storage layer.

## Acknowledgment

All experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr). This work is also a part of the BigStorage project, *H2020-MSCA-ITN-2014-642963*, funded by the European Commission within the Marie Skodowska-Curie Actions framework. This work was partially supported by the Spanish Ministry of Science and Innovation under the TIN2015–65316 grant, the Generalitat de Catalunya undercontract 2014–SGR–1051.

## References


1. Balouek, Daniel et al.: Adding Virtualization Capabilities to the Grid'5000 Testbed. In: Cloud Computing and Services Science, Communications in Computer and Information Science, vol. 367, pp. 3–20. Springer International Publishing (2013)
2. Bryant, Roy et al.: Kaleidoscope: cloud micro-elasticity via VM state coloring. In: EuroSys 2011. pp. 273–286. ACM (2011)
3. De, P., Gupta, M., Soni, M., Thatte, A.: Caching VM instances for fast VM provisioning: a comparative evaluation. In: Euro-Par 2012. pp. 325–336. Springer
4. Doug, H.: vmtouch: the Virtual Memory Toucher, https://hoytech.com/vmtouch/
5. Garcia, A.: Improving the performance of the qcow2 format. https://events.static.linuxfound.org/sites/events/files/slides/kvm-forum-2017-slides.pdf
6. Harter, Tyler et al.: Slacker: Fast distribution with lazy docker containers. In: FAST 2016. pp. 181–195 (2016)
7. Jeswani, D., Gupta, M., De, P., Malani, A., Bellur, U.: Minimizing Latency in Serving Requests through Differential Template Caching in a Cloud. In: IEEE CLOUD'2012. pp. 269–276. IEEE (2012)
8. Knauth, T., Fetzer, C.: DreamServer: Truly on-demand cloud services. In: ACM SYSTOR'2014. ACM (2014)
9. Lagar-Cavilla, Horacio Andrés et al.: SnowFlock: rapid virtual machine cloning for cloud computing. In: EuroSys 2009. pp. 1–12. ACM (2009)
10. Mao, M., Humphrey, M.: A performance study on the VM startup time in the cloud. In: IEEE CLOUD 2012. pp. 423–430. IEEE (2012)
11. Merkel, D.: Docker: lightweight linux containers for consistent development and deployment. Linux Journal (239), 2 (2014)
12. Nathan, S., Ghosh, R., Mukherjee, T., Narayanan, K.: Comicon: A co-operative management system for docker container images. In: IEEE IC2E 2017. pp. 116–126
13. Nguyen, T.L., Lèbre, A.: Virtual Machine Boot Time Model. In: IEEE PDP 2017. pp. 430–437. IEEE (2017)
14. Nguyen, T.L., Lèbre, A.: Conducting thousands of experiments to analyze vms, dockers and nested dockers boot time. Tech. rep. (2018)
15. Nicolae, B., Cappello, F., Antoniu, G.: Optimizing multi-deployment on clouds by means of self-adaptive prefetching. In: Euro-Par 2011. pp. 503–513. Springer (2011)
16. Nicolae, B., Rafique, M.M.: Leveraging collaborative content exchange for on-demand vm multi-deployments in iaas clouds. In: Euro-Par 2013. pp. 305–316
17. Razavi, K., Kielmann, T.: Scalable virtual machine deployment using VM image caches. In: SC13. p. 65. ACM (2013)
18. Razavi, K., Van Der Kolk, G., Kielmann, T.: Prebaked $\mu$vms: Scalable, instant VM startup for IAAS clouds. In: IEEE ICDCS'2015. pp. 245–255. IEEE (2015)
19. Vangoor, B.K.R., Tarasov, V., Zadok, E.: To FUSE or Not to FUSE: Performance of User-Space File Systems. In: FAST 2017. pp. 59–72 (2017)
20. Vrable, Ma et al.: Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In: ACM SOSP'2005. vol. 39, pp. 148–162. ACM (2005)
21. Wu, Ren et al.: A reference model for virtual machine launching overhead. IEEE Transactions on Cloud Computing **4**(3), 250–264 (2016)
22. Zhang, I., Denniston, T., Baskakov, Y., Garthwaite, A.: Optimizing VM Checkpointing for Restore Performance in VMware ESXi. In: USENIX Annual Technical Conference. pp. 1–12 (2013)
23. Zhang, Z., Li, D., Wu, K.: Large-scale virtual machines provisioning in clouds: challenges and approaches. Frontiers of Computer Science **10**(1), 2–18 (2016)