N-Docker: A NVM-HDD Hybrid Docker Storage Framework to Improve Docker Performance

Lin Gu, Qizhi Tang, Song Wu, Hai Jin, Yingxi Zhang, Guoqiang Shi, Tingyu Lin, Jia Rao

N-Docker: a NVM-HDD Hybrid Docker Storage Framework to Improve Docker Performance

Lin Gu[1], Qizhi Tang[1], Song Wu[1], Hai Jin[1], Yingxi Zhang[2], Guoqiang Shi[2], Tingyu Lin[2], and Jia Rao[3]

[1] National Engineering Research Center for Big Data Technology and System
Services Computing Technology and System Lab, Cluster and Grid Computing Lab
School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, 430074, China
[2] State Key Laboratory of Intelligent Manufacturing System Technology
Beijing, 100854, China
[3] The University of Texas at Arlington, Arlington, TX 76019, USA
wusong@hust.edu.cn

Abstract. Docker has been widely adopted in production environment, but unfortunately deployment and cold-start of container are limited by the low speed of disk. The emerging *non-volatile memory* (NVM) technology, which has high speed and can store data permanently, brings a new chance to accelerate the deployment and cold-start of container. However, it is expensive to replace the whole *hard disk driver* (HDD) with NVM. To achieve the fastest deployment and cold-start with lowest cost, we conduct in-depth analysis on the Top-134 images in Docker Hub and obtain two main insights as: 1) the storing latency of layered image has become the bottleneck of container deployment; 2) only a few image layers are required for container cold-start. Based on these two findings, we propose a NVM-HDD hybrid docker storage framework as N-Docker. It can effectively accelerate container cold-start by detecting the bottleneck layers as well as cold-start required layers and storing them into NVM for faster container startup with limited NVM capacity. Experimental results show that N-Docker can accelerate the container deployment by 1.21X and cold-start by 2.96X. Compared to NVM-Docker, which stores all images into NVM, N-Docker achieves the same performance improvements while reducing the usage of NVM by 88.22%.

Keywords: Container Deployment · Container cold-start · Docker · Image · NVM.

1 Introduction

Docker [2] is a lightweight virtualization system, with the advantages of continuous integration, version control, portability and fast migration, which has been widely used in industry. Different from virtual machines [5], containers share the operating system kernel with the underlying host, which enables rapid deployment with low performance overhead. Docker packages everything needed by

an application as an image, including runtime tools, system tools, and system dependencies. Images are layered and read-only, and adopt copy-on-write to reduce the usage of storage space. Despite being lightweight, container's startup is much slower in practice due to deploying image and file-system provisioning bottlenecks. The startup of non-local containers includes two processes: deployment and cold-start.

The non-local containers images must be first downloaded from remote registry, then stored in local disk [9]. Note that the image downloading latency is usually determined by the image size, network dynamics and available bandwidth. To cope with slow downloading, a method based on peer-to-peer is adopted by some cluster management systems (such as Tupperware [12], Borg [15] etc.) to accelerate the distribution of package. Slacker [10] accelerates container deployment by lazily pulling image data when needed. Unfortunately, these pioneer work all focus on reducing the downloading latency, but fail to take the image storing latency into consideration. When the image layers are downloaded, they must be stored layer by layer sequentially into local disk. Note that the upper layer downloading may complete first but still have to wait for the lower ones, leading to a long storing latency after the network download is completed. Our analysis shows that with a network speed of 100Mbps, image storing accounts for at least 23.5% of container deployment latency, and should be carefully analysed and studied to improve the overall deployment latency.

After successful deployed in local disks, the containers are ready to be launched to provide certain services with a cold-start latency. It widely agrees that containers are usually short-lived and dynamically activated/deactivated according to real time service demands such as serverless computing, hence the long cold-start latency severely hinders the service quality [7]. Moreover, the cold-start latency of launching multiple container simultaneously increases significantly with the container number. For example, launching 20 containers is about 7 times slower than launching 1 container [16]. According to the studies conducted by Google Borg [15], the median task cold-start latency is 25 seconds, and above 80% of the latency is caused by the slow I/O speed of local disk. To mitigate this problem, Akkus et al. [7] try to lower the function instances from container to separate process to share libraries with other functions of an application. Oakes et al. [13] create a cache of pre-warmed Python interpreters to speed-up the I/O process. However, they either weaken the function isolation by sharing libraries or are designed for specialized system with limited application scenarios.

Facing the above problems, it is desired to design a container acceleration framework to speed up both deploying and cold-start without loss of generality. Recently, the development of new hardware, i.e. non-volatile memory (NVM) [17], brings a new opportunity. NVM shows excellent characteristics of non-volatility, byte addressing, and superior reading performance. However, due to the high cost of NVM, its size is usually limited, which makes it impossible to store all images in NVM. To make full use of the limited NVM resource, and accelerate the deployment and cold-start of container, we should carefully select the image layers and schedule the NVM resource accordingly.

To address this issue, we conduct an in-depth analysis on bottlenecks of container deployment and cold-start on the Top-134 images downloaded more than 1 million times on the Docker Hub [3]. Based on our analysis, we propose a N-Docker framework to optimize the deployment and cold-start of container with limited NVM resource. The main contributions of this paper are as follows:

- We analysis the deployment and cold-start on the Top-134 images and find two key issues of container startup: 1) container deployment latency can be greatly reduced by improving the image layer storing; 2) container cold-start only requires a small part of files in the image. Based on these two findings, we discuss the opportunity and challenges of adapting NVM to speedup container startup.
- A N-Docker framework is designed to improve container deployment and cold-start. We focus on the storing layers and leverage NVM to accelerate its storing. Furthermore, we detect and write the hot image files required by container cold-start to NVM in order to reduce cold-start latency.
- We implement N-Docker, a NVM-HDD hybrid docker storage framework. The experimental results demonstrate that N-Docker achieves the same performance as NVM-Docker. Moreover, N-Docker can reduce the size usage of NVM by 88.22%. Compared to traditional Docker which stores all images in hard disk, N-Docker can speed up the deployment and cold-start of containers by 1.21X and 2.96X separately.

The rest of this paper is organized as follows. We discuss the design and implementation of N-Docker in Section 2. In Section 3 we evaluate the effectiveness of N-Docker as well as the overhead. In Section 4 we discuss the related work. Finally, Section 5 concludes this paper.

2 Design and Implementation

2.1 Opportunities and Challenges from Emerging NVM

Container technique has been widely used in the industry during the past few years [12, 15]. In this section, we introduce the deployment and cold-start of container in detail and discuss the opportunities of NVM.

Time-consuming image deploying Before a container can be started up, its image has to be downloaded and stored from the Internet first. In detail, the data downloaded from the Internet is just some compressed files, which will be decompressed and stored as layered image later. We do a lot of research on the Top-134 images downloaded more than 1 million times on the Docker Hub and find that the average deployment latency of the Top-134 images is about 20.7 seconds, and find that storing latency accounts for 23.4% of deploying time. As a result, the storing latency should also be considered during container deployment acceleration.

Slow container cold-start Similarly, container cold-start is also slower in practical cases due to the poor performance of local disk I/O. We do comprehensive analysis and research on the Top-134 images and find out only a few base

files in the large image, called *Hot Image File* (HIF), are required during the cold-start for different types of containers. The HIF usually includes bin files, system dependencies, application files, and execution engines, and will be access when a container is launched.

Now the server platform supports NVM in the form of NVDIMMs [6]. However, naively store the whole image into the NVM is not realistic in practical cases, since the price of NVM devices is relatively high. To this end, we must take the characteristics of container into consideration to accelerate container startup with limited NVM resource. Through the above analysis and discussion, we conclude that the deployment and cold-start of container are the main bottleneck of container startup. In this section, we introduce N-Docker, a NVM-HDD hybrid docker storage framework, to speed up its deployment and cold-start.

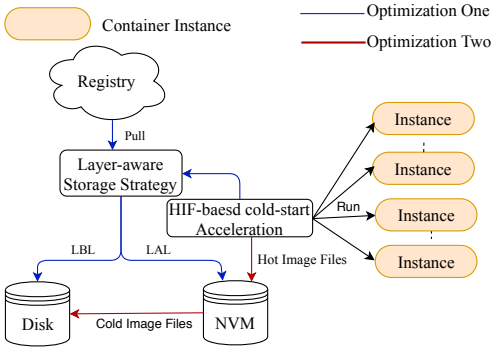

Fig. 1. N-Docker overview

2.2 Overview

According to the previous researches of the process on container deployment, analysis of the files used for container cold-start and the characteristics of NVM, we design N-Docker based on the following three objectives:

- **Container deployment acceleration:** In general, we can accelerate the container deployment by storing the images in NVM. However, considering the capacity limitation of NVM devices, we design a container deployment strategy to store the bottleneck layers in NVM instead of the entire images in NVM during the process of container deployment.
- **Container cold-start improvement:** Similarly, to improve the container cold-start via NVM, we also need detect and store the HIF in NVM, achieving fast container cold-start and high NVM resource utility at the same time.
- **Generality and transparency:** In terms of generality, N-Docker should support a wide range of workflows to accelerate deployment and cold-start. As for transparency, N-Docker should support these workflows without modifying the application or weaken the isolation.

Figure 1 describes the overview of the N-Docker architecture. It is clear that N-Docker has two core components. According to the finding that storing image is one of the reasons for the slow deployment of containers, we design *Layer-aware Storage Strategy* (LASS) to store partial image in NVM during the deployment of container. Based on the finding that the cold-start of container only needs Hot Image Files, we propose the *HIF-based Cold-start Acceleration* (HBCSA) method to acquire HIF, store them in NVM, and write other cold image files back to hard disk.

2.3 Layer-aware Storage Strategy

To achieve container deployment acceleration, we design LASS which speeds up the deployment of container while reducing the space usage of NVM. The latency of container deployment is mainly resulted from the download image and storing image. In order to speed up container deployment, we take NVM instead of the traditional disks to store the images. However, the capacity of NVM is usually limited, since it is more expensive in the price. So it is not economical to store all images in NVM. Our goal is to minimize usage of NVM while enabling rapid container deployment.

During the image deployment, there are three threads downloading different layers of one image parallelly. A ChainID is attached to each downloaded layer to identify a layer, and its value is calculated by sha256 algorithm according to layer's diffID and its parent chainID. Therefore, image layers must be stored layer by layer sequentially into local disk. The different layer sizes usually lead to different download latency. That is, one layer may still in download while the others have finished. We refer to the layer being downloaded lastly as *Last Downloading Layer* (LDL), which is usually the largest layer. As shown in Figure 3, the lower image layers of LDL is called *layers below LDL* (LBL) and its status is *Pull complete*, which has already been downloaded and stored in the hard disk. The higher image layer above LDL is called *Layers above LDL* (LAL) and its status is *Download Complete*, which has been downloaded while not yet stored in the hard disk. The storing of LAL will be postponed until LDL has been downloaded and stored due to layer sequential storing. Once the LDL is downloaded, LAL and LDL will be stored together, which may lead to high latency of intensive writes after the network download is completed. To address this issue, we design LASS which can identify LDL and only store LDL and LAL in NVM.

Layer-aware Storage Strategy Design To take advantages of fast I/O speed of NVM with a constrained capacity, we need design LASS to determine the layers that should be stored in NVM and disk, towards the goal of fastest storing and fewest NVM usage. Hereafter, we investigate different schemes and find the optimal strategy. To answer this question, we use a *Boundary Layer* (BL) to divide the layers of one image into two parts, namely L_n (above BL) and L_d (below BL) with the sizes of S_n and S_d, to be stored in NVM and local disk,

respectively. BL stores in NVM and its size is S_{bl}. We suppose that an image has N layers, LDL is the K_{th} layer and BL is the M_{th} layer. In order to find the optimal layer aware storage strategy, we introduce two indicators as the criteria to evaluate the performance of different strategies, namely T_{Total} (the total latency of container deployment) and U_{Total} (the total NVM usage of container deployment). T_{Total} and U_{Total} are calculated according to the equations as follows.

$$T_{Total} = T_D + T_{Disk} + T_{NVM} \tag{1}$$

As shown in Equation 1, T_{Total} consists of three parts. T_D represents the latency caused by the download image. T_{Disk} and T_{NVM} are the latencies storing image in hard disk and storing image in NVM after the network download is completed, separately.

$$T_{Disk} = \begin{cases} 0 & M \leq K \\ \alpha * (S_{LDL} + S_{LAL} - S_n - S_{bl}) & M > K \end{cases} \tag{2}$$

T_{Disk} is equal to the total image size written to the hard disk after the network download is finished divided by the hard disk write speed. As shown in Equation 2, α is the reciprocal of the hard disk write speed. When the network download is completed, LBL has been stored. Therefore, if BL is LBL or LDL, T_{Disk} is 0. If BL is LAL, layers between LDL and BL store in disk after network download is completed.

$$T_{NVM} = \begin{cases} \beta * (S_{LDL} + S_{LAL}) & M \leq K \\ \beta * (S_{bl} + S_n) & M > K \end{cases} \tag{3}$$

T_{NVM} is equal to the total image size written to NVM after the network download is completed divided by the NVM write speed. As shown in Equation 3, β is the reciprocal of NVM write speed. After network download is completed, if BL is LBL or LDL, layers between LDL and the highest layer store in NVM. If BL is LAL, layers between BL and the highest layer store in NVM.

$$T_{Total} = \begin{cases} T_D + \beta * (S_{LDL} + S_{LAL}) & M \leq K \\ T_D + \alpha * (S_{LDL} + S_{LAL} - S_{bl} - S_n) + \beta * (S_{bl} + S_n) & M > K \end{cases} \tag{4}$$

$$U_{Total} = S_{bl} + S_n \tag{5}$$

Combining 1, 2, and 3, we can easily get Equation 4. Since the write speed of NVM is several orders of magnitude faster than that of hard disks, it is assumed that $\alpha \gg \beta$. Equation 5 shows that U_{Total} is the space usage of NVM, which is another performance indicator. Because we only care about latency caused by storing image, we set T_D as a constant. Our goal is to minimize U_{Total} on the premise of minimizing T_{Total}. According to the location of boundary layers, we design the following three strategies as shown in Figure 2.

Strategy 1 set the Boundary Layer as the LBL. At this time, $M < K$ and $S_{LDL} + S_{LAL} < S_{bl} + S_n$. The result is shown in Equation 6. T_Total is the sum of T_D and the latency caused by storing S_{LAL} to NVM, and U_{Total} is S_n.

$$\begin{cases} T_{Total} = T_D + \beta * (S_{LDL} + S_{LAL}) \\ U_{Total} = S_{bl} + S_n \end{cases} \tag{6}$$

Strategy 2 takes LAL as the Boundary Layer. At this time, $M > K$. The result is shown in Equation 7. Because $\alpha \gg \beta$, the delay T_{Total} of strategy 2 is higher than that of strategy 1, so strategy 1 is better than strategy 2.

$$\begin{cases} T_{Total} = T_D + \alpha * (S_{LDL} + S_{LAL} - S_{bl} - S_n) + \beta * (S_{bl} + S_n) \\ U_{Total} = S_{bl} + S_n \end{cases} \tag{7}$$

Strategy 3 selects the LDL as the Boundary Layer. At this time, $M = K$, $S_{bl} + S_n = S_{LDL} + S_{LAL}$. The result is shown in Equation 8. The strategy 3's T_{Total} is the same as the strategy 1. In strategy 1, $S_{LDL} + S_{LAL} < S_{bl} + S_n$. So the U_{Total} of strategy 3 is smaller than strategy 1, strategy 3 is better than strategy 1.

$$\begin{cases} T_{Total} = T_D + \beta * (S_{LDL} + S_{LAL}) \\ U_{Total} = S_{LDL} + S_{LAL} \end{cases} \tag{8}$$

It is easy to conclude that strategy 3 is the best choice with the lowest latency. On the premise of enabling rapid deployment of containers, the usage of NVM is minimized. Therefore, we adopt strategy 3 and set the Boundary Layer as LDL, as shown in Figure 3. While LBL are storing, other image layers are also being downloaded from the network. We chose to store LBL in disk to reduce the usage of NVM. When LAL and LDL are being stored, the network download process has ended. At this time, the latency of container deployment depends entirely on the storing LDL and LAL. We store LDL and LAL in NVM. As a result, containers deployment is significantly accelerated.

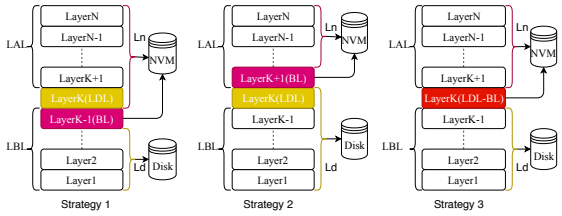

Fig. 2. Strategy overview

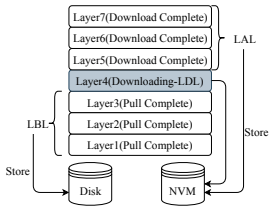

Fig. 3. Image storage

2.4 HIF-based Cold-Start Acceleration

To achieve fast cold-start, we propose HBCSA to speed up the cold-start of containers while reducing the usage of NVM. Note that containers cold-start only requires HIFs, hence only storing the HIFs in NVM devices during cold-start can speedup the cold-start. To accelerate the cold-start of container for the first time, we execute static analysis to identify image layers including HIFs, and

store them in NVM during the deployment of container. The other is dynamic analysis. HIFs obtained by static analysis are redundant. Therefore, we execute dynamic analysis to obtain accurate HIFs during the cold-start of container. Static and dynamic analysis are detailed separately as follows.

Static analysis In the process of deploying the image, if the layer contains HIFs, the whole layer will be stored in NVM to obtain some HIFs initially. Dockerfile consists of a series of commands which can be obtained by a simple "string parsing" method. From section 2.1, we can know that the HIFs include Bin files, system dependencies, application files, and execution engines. Bin files and system dependencies account for a small proportion of the total HIFs. And the image layers containing bin files or system dependencies are generally large. So an image layer that contains only bin files or system dependencies is stored in Disk without wasting NVM resources. For an image layer containing the execution engine or application files, we choose to store it in NVM as a coarse-grained HIFs.

Dynamic analysis Once the container cold-start is finished, the application files in need will be loaded into memory. Dynamic analysis mainly analyzes the necessary files and file dependencies in the image by tracking system calls, changes of files or directories, and running of processes. These files are HIFs. In order to improve the utilization rate of NVM, we only store HIFs in NVM, with other image files brushed back to the hard disk. In this way, the utilization rate of NVM is greatly improved, and the cold-start speed of the container is also accelerated. Compared with the traditional architecture, HIFs will not be replaced back to disk due to memory collection in the multi-container scenario with the same host. In this way, container running reduce the disk I/O overhead caused by missing page interruptions. When a container is suspended for a period of time or restarted, it can start running faster by reducing I/O latency caused by page missing interruptions.

3 Evaluation

We implement N-Docker, a NVM-HDD Hybrid Docker Storage Framework to accelerate container deployment and cold-start. In order to evaluate the performance of N-Docker, we conduct a comparative experiment between N-Docker and native Docker, and a comparative experiment between N-Docker and NVM-Docker. Our experiments are based on 134 images in Docker hub, which are downloaded more than 1 million times.

3.1 Experiment Setup

Environment Table 1 provides a detailed description of memory configuration. We simulate NVM as a fast block device [4] and install ext4 with DAX(direct access) [1] on it. The machines interconnect with each other in 1Gbps network. We implement N-Docker based on Ubuntu 16.04 and Docker 18.06-ce.

Table 1. Memory configuration

	DRAM	NVM
Capacity	4G	4G
Channels	1	2
Bandwidth	8GB/s	3.6GB/s(Read)
		1.3GB/s(Write)
Read/Write Latency	1	4.4x(Read)
(Normalized to DRAM)	1	12x(Write)

3.2 Deployment

N-Docker divides the image into two parts, one of which is stored in NVM and the other is stored in Disk. In this section, to compare the performance of N-Docker with that of NVM-Docker, we deploy Top-134 containers through N-Docker and NVM-Docker respectively. The experimental results of container deployment latency are shown in Figure 4. As can be seen from the figure, the deployment latency of NVM-Docker container in each category is larger than that of N-Docker by more than 97%. Therefore, it can be concluded that N-Docker divides the image into two parts without incurring additional latency. The space usages of NVM of N-Docker and NVM-Docker are shown in Figure 5. With regards to the category of distro, N-docker uses the same space size of NVM as NVM-Docker. The main reason is that the distro category is the basic image, and the only one layer or the first layer accounts for most of the entire image size. In this case, we store the entire image in NVM. The category of web fwk, which has the most decrease in NVM usage, has a 38.5% decrease in NVM usage. The reason for it is that the LDL of this category of image is located further back in the image layer, and more image layers are stored in DISK. In addition, N-Docker's container deployment is almost as fast as NVM-Docker. On average, N-Docker's NVM usage is 28.53% less than NVM-Docker.

In section 2, we have seen the latency of various container deployments, with downloading latency accounting for 76.6% and storing latency accounting for 23.4%. In this section, we evaluate the performance of N-Docker in container deployment by deploying Top-134 containers separately through N-Docker and Docker. Experimental results are shown in Figure 4. The most significant drop in container deployment latency is 26.7% for the category of distro, as the entire image of distro is stored in NVM. The percentage of image layer stored in NVM is the highest in all categories, and the benefits brought by accelerating container deployment through NVM is the largest. The lowest reduction in container deployment latency is 19.3% for the category of web fwk, since the percentage of image layer stored in NVM by web fwk is the lowest in all categories, and the benefits of accelerating container deployment through NVM is the least. On average, N-Docker's container deployment latency is 21.14% lower than Docker's. We speed up the entire container deployment process by reducing the latency of storing images. So we evaluate the latency caused by NVM-Docker, N-Docker and Docker in the process of storing images. The results are shown in Figure 6.

On average, the latency of the N-Docker storing image after the network download is finished is reduced by 90.3% compared to Docker, and is comparable to NVM-Docker.

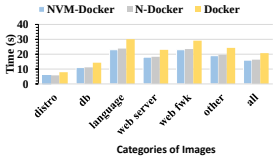

Fig. 4. Deployment time

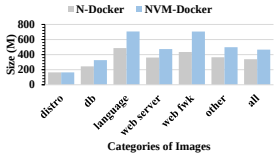

Fig. 5. NVM usage in docker deployment

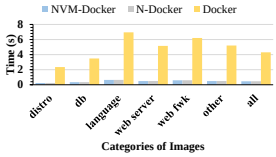

Fig. 6. Storing image time

3.3 Cold-start

We store Hot Image Files in NVM to speed up the cold-start of the container. In order to verify that the Hot Image Files selected by our scheme is indeed the file necessary for container cold-start, we conduct the experiments on the Top-134 containers's cold-start through N-Docker and NVM-Docker. The result of cold-start latency is shown in Figure 7, which demonstrates that N-Docker's cold-start latency is only 2% slower than NVM-Docker. The space usage of NVM is shown in Figure 8. As can be seen from the figure, the largest reduction in NVM usage is 97.12 % for the category of distro, as the distro class is the basic image. The vast majority of such images are auxiliary tools, package managers, and dependencies. The files needed for by the category of distro are very few. The minimum reduction in NVM usage is 70.12% for the category of web server. This type of container contains more executable files, configuration files and the underlying execution engine. Taking the JVM as an example of execution engine, common versions of JVM exceed 100M, which makes Hot Image File larger. In summary, the Hot Image File used by N-Docker contains almost all the files necessary for container cold-start, and the NVM's usage of N-Docker is 88.22% less than NVM-Docker.

In order to evaluate the cold-start performance of N-Docker, we compare the cold-start latency of containers by N-Docker and Docker. As shown in Figure 7, the maximum reduction of cold-start delay is 76.1% for distro container. The reason for it is that containers of the distro category are the simplest, which requires only a small number of files and then builds an independent execution environment. The cold-start latency of containers in the distro category is mainly resulted from the overhead of I/O. Containers of the distro category get higher promotion by using NVM to store Hot Image Files to speed up container cold-start. The minimum reduction in container cold-start latency is 62.6% in the category of web server, as containers in the category of web servers are the most complex. Such containers's cold-start requires not only building an independent execution environment, but also starting the server's daemon process. Therefore, accelerating container cold-start by taking NVM devices to store Hot Image Files gets the least benefits. On average, N-Docker can reduce the latency of the containers's cold-start by 33.8%, compared to that of Docker.

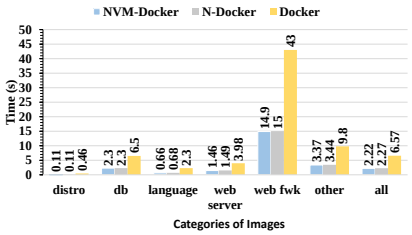

Fig. 7. Cold-start time

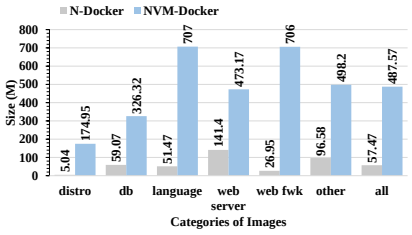

Fig. 8. NVM usage

4 Related Work

This research work is to accelerate container deployment and cold-start based on emerging NVM. The slow deployment and cold-start of containers has also been widely discussed by other researchers.

Deployment: Some cluster management systems, such as Tupperware [12], Borg [15], use peer-to-peer technique to reduce the load on the central repository and speed up packet distribution. However, they are not applicable to Docker images. Slacker [10] accelerates container deployment by reducing network I/O, which lazily pulls image data when needed. However, slacker needs a longer time to build image and a greater demand for storage in the registry. Cider [9] changes the working node's local Docker storage to an all-nodes-sharing network storage, allowing image data to be loaded on demand when deploying containers.

Cold-start: CNTR [14] divides the traditional image into two parts: the "fat" image contains complete functions, while the "slim" image contains only the core files needed by common user-case. CNTR reduces image size, which makes Docker lighter. However, CNTR incurs overhead for some benchmarks. Unikernel [11] uses the library OS [8] to screen out the required operating system components to construct a lighter-weight executable application operating system. But Unikernels cannot debug and require static linking tools in the library OS. SAND [7] weakens the function instances from container-level isolation to separate process-level to share libraries, which only require to be loaded into container once, with other functions of an application. SOCK [13] create a cache of pre-warmed Python interpreters to avoid that Python runtime is initialized repeatedly.

Existing work does not consider storing image during container deployment, weaken function's isolation or cannot be applied to general containers for cold-start acceleration.

5 Conclusion

Rapid deployment and cold-start of container are very important, such as in the serverless computing scenario. To achieve this goal, we leverage the emerging NVM device and design N-Docker, a NVM-HDD hybrid docker storage framework. N-Docker stores LAL and LDL in NVM during container deployment and

Hot Image Files in NVM during container cold-start. Through extensive experiments, we validate the efficiency of N-Docker by the fact that it can accelerate the median container deployment by 1.21X and cold-start by 2.96X with very few NVM. Compared to NVM-Docker, which stores all images in NVM, the proposed N-Docker achieves the same performance improvements while reducing the usage of NVM by 88.22%.

References

1. Add support for nv-dimms to ext4. https://lwn.net/Articles/613384/
2. Docker. https://www.docker.com/
3. Docker hub. https://hub.docker.com/u/library/
4. Emulate persistent memory. http://pmem.io/2016/02/22/pm-emulation.html
5. Linux kernel virtual machine. https://www.linux-kvm.org/page/MainPage
6. Nvdimm. https://www.micron.com/products/dram-modules/nvdimm
7. Akkus, I.E., Chen, R., Rimac, I., Stein, M., Satzke, K., Beck, A., Aditya, P., Hilt, V.: Sand: Towards high-performance serverless computing. In: Proceedings of the 2018 USENIX Annual Technical Conference. pp. 923–935 (2018)
8. Belay, A., Bittau, A., Mashtizadeh, A.J., Terei, D., Mazières, D., Kozyrakis, C.: Dune: Safe user-level access to privileged cpu features. In: Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation. pp. 335–348. USENIX Association (2012)
9. Du, L., Wo, T., Yang, R., Hu, C.: Cider: A rapid docker container deployment system through sharing network storage. In: Proceedings of 19th International Conference on High Performance Computing and Communications. pp. 332–339. IEEE (2017)
10. Harter, T., Salmon, B., Liu, R., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Slacker: Fast distribution with lazy docker containers. In: Proceedings of the 14th USENIX Conference on File and Storage Technologies. pp. 181–195. USENIX Association (2016)
11. Madhavapeddy, A., Scott, D.J.: Unikernels: the rise of the virtual library operating system. Communications of the ACM **57**(1), 61–69 (2014)
12. Narayanan, A.: Tupperware: containerized deployment at facebook (2014)
13. Oakes, E., Yang, L., Zhou, D., Houck, K., Harter, T., Arpaci-Dusseau, A., Arpaci-Dusseau, R.: Sock: Rapid task provisioning with serverless-optimized containers. In: Proceedings of the 2018 USENIX Annual Technical Conference. pp. 57–70. USENIX Association (2018)
14. Thalheim, J., Bhatotia, P., Fonseca, P., Kasikci, B.: Cntr: Lightweight OS containers. In: Proceedings of the 2018 USENIX Annual Technical Conference. pp. 199–212. USENIX Association (2018)
15. Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., Wilkes, J.: Large-scale cluster management at google with borg. In: Proceedings of the 10th European Conference on Computer Systems. p. 18. ACM (2015)
16. Wang, L., Li, M., Zhang, Y., Ristenpart, T., Swift, M.: Peeking behind the curtains of serverless platforms. In: Proceedings of the 2018 USENIX Annual Technical Conference. pp. 133–146. USENIX Association (2018)
17. Xu, J., Zhang, L., Memaripour, A., Gangadharaiah, A., Borase, A., Da Silva, T.B., Swanson, S., Rudoff, A.: Nova-fortis: A fault-tolerant non-volatile main memory file system. In: Proceedings of the 26th Symposium on Operating Systems Principles. pp. 478–496. ACM (2017)