Cider: a Rapid Docker Container Deployment System Through Sharing Network Storage

Lian Du, Tianyu Wo, Renyu Yang[*], Chunming Hu
State Key Laboratory of Software Development Environment
BDBC, Beihang University, Beijing, China
Email: *{dulian, woty, hucm}@act.buaa.edu.cn; renyu.yang@buaa.edu.cn*[*]

***Abstract* — Container technology has been prevalent and widely-adopted in production environment considering the huge benefits to application packing, deploying and management. However, the deployment process is relatively slow by using conventional approaches. In large-scale concurrent deployments, resource contentions on the central image repository would aggravate such situation. In fact, it is observable that the image pulling operation is mainly responsible for the degraded performance. To this end, we propose Cider - a novel deployment system to enable rapid container deployment in a high concurrent and scalable manner at scale. Firstly, on-demand image data loading is proposed by altering the local Docker storage of worker nodes into all-nodes-sharing network storage. Also, the local copy-on-write layer for containers can ensure Cider to achieve the scalability whilst improving the cost-effectiveness during the holistic deployment. Experimental results reveal that Cider can shorten the overall deployment time by 85% and 62% on average when deploying one container and 100 concurrent containers respectively.**

Keywords — container; network storage; copy-on-write; application deployment

I. Introduction

Recent years have witnessed the prosperity of container technique and Docker is undoubtedly the most representative among them [1]. Compared with virtual machine, container can provision performance and user-space isolation with extremely low virtualization overheads [2]. The deployment of applications (especially in large clusters) increasingly tends to depend on containers, driven by recent advances in packing and orchestration. Massive-scale container deployment is becoming increasingly important for micro-service composition and execution. Unfortunately, the enlarged system latency introduced by slow container deployment is becoming a non-negligible problem, which is critical in scenarios such as bursting traffic handling, fast system component failover etc. According to our in-depth investigation of conventional deployment approaches, we found that the mean deployment time of the top 69 images in Docker Hub is up to 13.4 seconds and 92% of the holistic time is consumed by transferring image data through network. Even worse, in large-scale concurrent deployments, resource contentions on the central image repository would aggravate such situation.

To cope with slow container deployment and improve the launch efficiency, cluster management systems (such as Borg [12], Tupperware[13], Fuxi[25] etc.) adopt a P2P-based method to accelerate the package distribution. VMware open-sourced project Harbor [19] is an enterprise-class container registry server and it also integrates the decentralized image distribution mechanism. However, application's images still need to be fully downloaded to worker nodes, resulting in an extended latency derived from transferring large amounts of data through local network. Harter et al. [18] propose a lazy fetching strategy for container data on single node and the method can effectively reduce the container provision time. Nevertheless, large-scale and concurrent deployment experiments have not yet been conducted, leading to difficulties in validating their effectiveness and understanding their operational intricacies at scale. In reality, these scenarios are commonly-used and scalability bottleneck might manifest very frequently. Therefore, it is particularly desirable for a container deployment system to rapidly provision container especially at large scale.

In this paper, we present Cider - an innovative deployment system for Docker containers, which can enable rapid container deployment in a high concurrent and scalable manner. Specifically, we alter the local Docker storage of worker nodes into all-nodes-sharing network storage, allowing for on-demand image data loading when deploying containers. This approach can dramatically reduce the transferred data, thereby considerably accelerating the single container deployment. Additionally, to achieve higher scalability and efficiency of memory usage, we design and implement a local copy-on-write (COW) layer for setting-up containers. The approach assigns concurrent COW request flooding in network storage into local worker nodes. Experimental results show that compared with conventional approach, Cider is able to shorten the average deployment time by 85% in single container case. A 62% reduction can still be achieved when concurrently deploying 100 containers, with no conspicuous runtime overheads. In particular, the major contributions in this paper can be summarized as follows:

- Introducing a rapid and cost-effective container deployment system based on sharing network storage, which can reduce the amount of transferred data during the deployment.
- Combining network storage with the local file-level COW layer to guarantee the system scalability and memory efficiency in face of jobs of high concurrent deployment.

The remaining sections are structured as follows: Section II introduces the background and our motivation; Section III

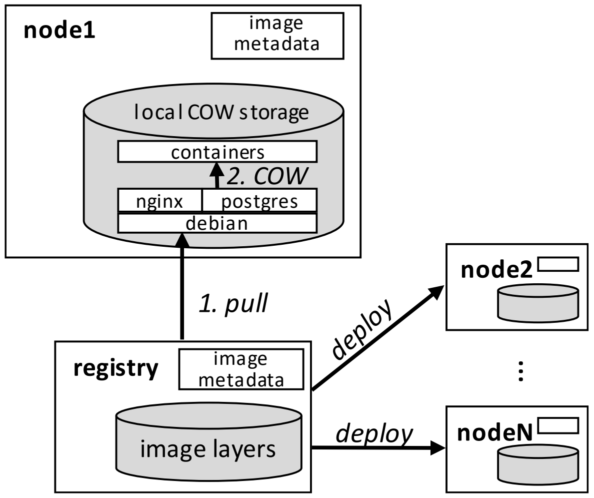


Fig. 1. Conventional container deployment architecture.

presents the core idea of design and implementation in Cider; Section IV shows the experimental results and Section V discusses the related work; Finally, Section VI presents the conclusion followed by the discussion of future work.

II. Background and Motivation

Container technique has been widely adopted in the production environment over a decade[12][13] as it provides several critical capabilities for large-scale cluster management such as resource constraint, process isolation etc. It did not become so prevalent until the release of Docker in 2013. As reported in a recent survey [1], 76% of the polled organizations leverage container technologies in production environments and Docker is the dominant container engine technology with a share of 94%. Hence we mainly discuss Docker container in this paper.

The primary reason for the success of container technology is due to the convenience when packing and deploying applications. All binaries and libraries that an application depends on can be effectively encapsulated within an image along with the application. This ensures the consistency of application runtime environment and greatly simplifies the procedure of deployment. Nevertheless, the downside cannot be neglected. With the image size drastically growing, the efficiency of container deployment will be negatively impacted.

A. Conventional Container Deployment Approach

Figure 1 shows the centralized architecture adopted by conventional container deployment approaches. The *registry* module plays a central role in image distribution within the cluster. Typically, a Docker image is made up of multiple layers, representing different file sets of software or libraries. All image layers are stored in the form of gzip-compressed files in *registry*. The image metadata is also stored in the *registry* and actually indicates the mapping relationship between images and layers. To deploy containers across the compute cluster, worker nodes (i.e., node 1 to node N) have to initially pull the complete image and then store it into the local copy-on-write (abbreviated as COW) storage. The local storage in the worker node is managed by the storage driver such as Aufs [3], Devicemapper [4], Overlay or Overlay2 [5]. After the image is pulled, the driver will setup another COW layer on top of the image for the container.

In effect, current deployment approaches [8] are far from enough, especially in cases that highly require rapid deployments. To illustrate this phenomenon, we evaluate the deploy time of the *top 69* images from Docker Hub [6] in sequence (Images are ordered by pull times and the selected images are pulled by more than 10 million times; the image size is 347 MB on average). The evaluation is actually conducted through setting up a private registry and deploying containers on another node which resides in the same LAN of the registry. The result demonstrates that the mean deploy time can be up to 13.4 seconds. Furthermore, there are a number of scenarios that require low deploy latency. For instance, services should scale-out instantly to tackle bursting traffic whilst applications should recover as soon as possible upon the arrival of failover event to retain the system availability [26]. Developers might intend to acquire the high efficiency and productivity in Continuous Integration/Continuous Deployment work flow. All these requirements motivate us to boost the performance of container deployment.

B. Deep Dive into Container Deployment Process

In order to investigate the root cause of slow deployment, we conduct a fine-grained analysis into the deployment procedure. In general, the process can be divided into two main operations - pull and run.

1) Pull: Image layers are stored in the registry in the form of gzip-compressed files. A single image consists of several layers and worker nodes can download these layers concurrently. Once a layer was downloaded, it can be decompressed into local storage immediately. Despite this, the decompression procedure cannot be parallelized due to the existing order of dependencies. E.g., local storage structure depends on the storage driver. Devicemapper driver stores images in thin-provisioned dm-devices, and Overlay driver stores images in a hierarchical structure in local file system.

2) Run: Once the image is successfully imported into the local storage, containers can derive from the same image. To run a container, storage driver will firstly create an *Init* layer on top of the image, initializing some container-specific files such as hostname, DNS address etc. Afterwards, a second layer will be created on top of the *Init* layer to serve as container's root file system. All these layers in local storage are generated by leveraging the copy-on-write technique for improving space utilization and startup performance. Additionally, different drivers implement the COW mechanism in different granularities: Aufs and Overlay is at the file level while Devicemapper is at the block level. Eventually, container's root directory will be changed to the top layer and application is ready to be started.

We evaluate the deploy time of top images on Docker Hub. Figure 2 depicts the time spent on pull and run operations during the deployment. Each point represents an iteration of an

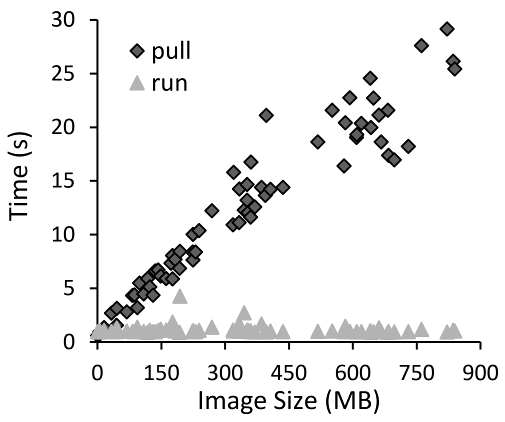


Fig. 2. Time consumed on pull and run during the deployment.

image deployment. It can be obviously observed that the run time remains steadily (roughly 1.1 seconds on average) when image size grows. By contrast, there is a significant positive relationship between the pull time and the image size. The average pull time is 12.3 seconds, taking up 92% of the average deploy time (13.4 seconds). This indicates that the slow pulling operator tends to be the core performance bottleneck and the scalability issues should be carefully considered.

Although many efforts such as concurrent download and data compression are made by Docker Registry [7] to accelerate the pulling process, pulling an entire huge image is still the dominating cause of the poor deploy performance. In particular, in the scenario of large-scale deployment, contentions on the registry would further slowdown the pulling procedure. To address this problem, we should find a way to minimize the data amount transferred during deployment.

III. Design and Implementation

In this paper, we present Cider (**C**eph **i**mage **d**eploy**er**) - a network storage based container deploy system for Docker. It can dramatically reduce the data transferred during the deployment thereby improving the holistic system performance.

A. *Using Ceph As the Underlying Storage*

Ceph is a scalable, high-performance distributed storage system [14] and is widely-used in cloud computing. It can provision several storage services, such as block storage, object storage and POSIX file system etc. In our architecture design, any network storage that has *snapshot* and *clone* features can be adopted to satisfy system fundamental functionalities. The reasons why we choose Ceph are as follows: 1) Ceph is open source and can be well-integrated with Linux kernel. 2) The Ceph community is very active and Ceph is promising as a cloud storage backend. 3) The performance, scalability and fault tolerance of the Ceph system are fairly suitable.

B. *Cider Architecture*

Figure 3 demonstrates the architecture overview of Cider system. Rather than storing images in the registry and dispersing image copies across worker nodes, Cider stores all data including both images and containers of the cluster in

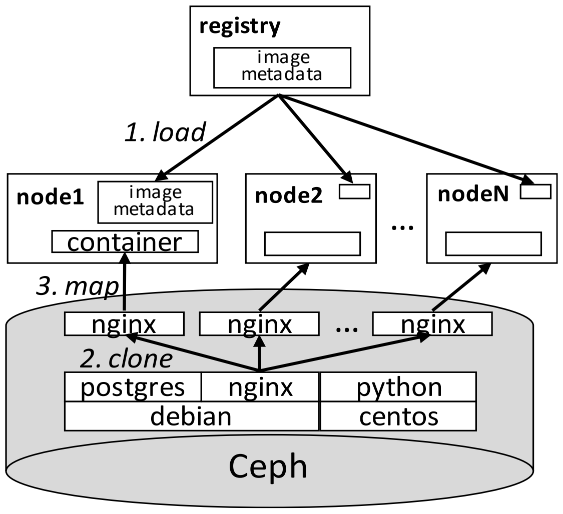


Fig. 3. Cider architecture completely based on network storage.

Ceph *RADOS Block Device (RBD)* pool. RBD is the block storage service of Ceph. In RBD pool, Docker images are stored in a hierarchical structure and each image corresponds to one RBD. By means of the Copy-on-Write mechanism, data is de-duplicated between parent and child images. For example, it is unnecessary for the Nginx image to actually copy all data from Debian. The image only needs to be built on top of the parent image Debian and it is sufficient to merely augment a COW layer onto the Debian image and write incremental data. Creating a child RBD includes two main procedures: taking a snapshot over the parent RBD and cloning the snapshot. The clone operation will generate an identical RBD of the parent at the snapshot moment. COW in Ceph is in the granularity of objects. An Object is the basic storage unit used by Ceph. Registry in Cider serves as a metadata server. It only stores image list, layer information of an image and the mapping relationship between layers and RBDs. Likewise, worker nodes in Cider do not have a local storage either. In particular, the process has three steps (see Figure 3):

1) *Image Metadata Loading*

When a worker node receives the container deploy command, it will load the related image metadata from the registry and then parse the metadata to figure out the required RBD. Compared with the whole image size, the metadata size is marginal enough to be negligible - it is roughly several hundred KBytes. Accordingly, the time to load and parse the metadata is quite short, approximately 0.2~0.3 seconds. In addition, the heavy workload on the registry caused by concurrent download requests can be significantly mitigated.

2) *Image Snapshot Clone*

The *clone* operation is executed on the snapshot of the currently deployed image. Since images will not change unless being updated, snapshots could be reused in multiple clones. Clone generates a COW copy of the image, which is served as the read/write layer for container.

3) *Mapping Container RBD to Worker Nodes*

The container RBD generated by clone can be utilized as a normal RBD. To use them on worker nodes, we should *map* them as local block devices. The *map* operation herein signifies registering a network RBD in local kernel (RBD will show up as */dev/rbd**). Mounting this local RBD to a specific directory is the final step to setup a root file system for containers. To run the container, the worker node needs to read data such as program binaries and configurations from the container RBD. Data blocks will only be fetched from Ceph on demand.

Moreover, the aforementioned lightweight operations *load*, *clone*, *map* will replace the heave *pull* operation adopted in the conventional approach. Most operations are performed on metadata: image metadata in *load*, RBD metadata in *clone*, kernel metadata in *map*. The only data required to startup container are lazily read. All these approaches are beneficial to minimize the data transferred during the deployment process, thereby substantially improving the deployment effectiveness.

C. *Implementation*

We have implemented a prototype system of Cider. Cider is modular-designed and has very few modifications on the original Docker program to maintain the compatibility.

1) *Image Metadata Loading*

In our preliminary implementation, we use *rsync* to retrieve image metadata from Registry. Once the metadata is downloaded, we send a signal to Docker daemon to reload the metadata from local disk. Since there is no metadata reload method in Docker daemon, we implement one and invoke it when getting the reload signal.

Although the current metadata loading process is simple and efficient, we plan to firstly refine it in a more automatic way where the Docker daemon is able to pull and load image metadata automatically. We also intend to transform the data request protocol to http protocol to generalize the applicability.

2) *Ceph Storage Driver Plugin*

Docker project allows developers to develop storage driver in the form of plugin, which could be easily installed and removed from Docker daemon. We adopt this way to develop the Ceph storage driver. It is similar to the bond between Docker daemon and Ceph, because it receives commands from Docker, translates them into Ceph operations, and returns the result to Docker. All the *clone* and *map* operations in Section III. *B* are performed by the Ceph storage driver. Every Docker daemon in worker node is equipped with this Ceph plugin. Essentially, the storage driver plugin is a process running inside a Docker container. It communicates with Docker daemon through UNIX Domain Socket. In addition, we use the APIs of *librbd* to manipulate RBDs in Ceph.

The interfaces that a developer should implement to build a Docker storage driver are descripted below. Due to the space limitations, some trivial interfaces and parameters are omitted.

Create (id, parent) - Create a new layer with the specified *id* from the *parent* layer. In Cider, we take a snapshot of the *parent* RBD and clone the snapshot, generating a new RBD named *id*. All RBDs in Cider are generated by this way except for the bottom base RBD.

Remove (id) - Remove the specified layer with this *id*. In Cider, we delete the RBD named *id*. The corresponding snapshot is reserved to speed up the next iteration of clone.

Get (id) - Return the mount point referred to by this *id*. Cider maps the RBD named *id* as a block device on worker node, mounts it to a local directory and returns the mount point.

Put (id) - Releases the system resources for the specified *id*. Cider will unmount the block device by this id and unmap the corresponding RBD from the worker node.

Diff (id, parent) & **ApplyDiff (id, diff)** - *Diff* produces an archive of the changes between the specified layer *id* and *parent*. Correspondingly, *ApplyDiff* extracts the change set from the given diff into the layer with the specified *id*. In Cider, we make use of the *NaiveDiffDriver*, provided by Docker project, to implement these two interfaces. We will further optimize the implementation and customize the Diff Driver in the future to achieve a better performance.

D. *Scalability and Memory Usage Considerations*

Although Cider works well in deploying one single container, we find it urgently necessary to tackle the scalability issue when deploying more containers concurrently. In a preliminary experiment, it is observable that the RBD clone operation gives rise to the scalability problems. Figure 6 depicts the scalability of all key operations in deploying

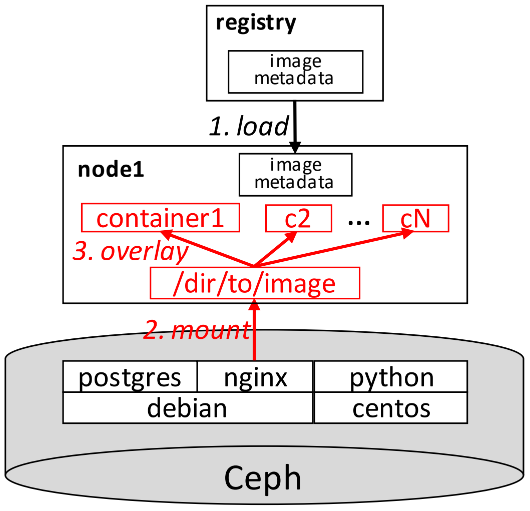


Fig. 4. Cider architecture with local container layer.

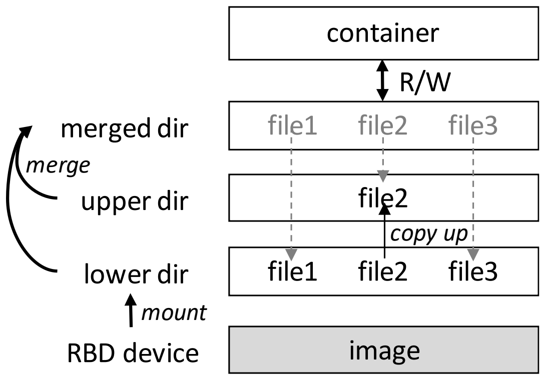


Fig. 5. Detailed description of the overlay.

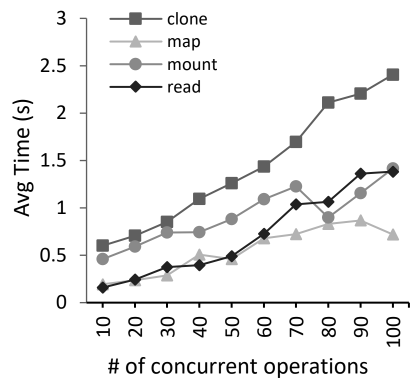


Fig. 6. Operation scalability.

containers. In addition, page cache cannot be effectively reused when reading the same file copy in different containers that derived from the same image. This is because Cider implements the COW of the container layer under local file system (inside Ceph). It could lead to excessive memory consumption.

Scalability. To address the scalability bottleneck, we made some modifications on the container layer. Figure 4 shows the revised architecture of Cider to achieve such objectives. The main optimizations are fulfilled by leveraging the local container layers. As shown in Figure 4, the different portion is marked in red. For clarification, we omit other worker nodes in the figure. Similarly, the deploy process can be divided into three main steps:

1) Load Image Metadata

This step remains the same with that in section III. B. 1).

2) Mount Image RBD

Instead of performing a clone operation on the image, we maps the image RBD directly from the pool to a local block device and mounts it on a local directory. The directory will contain a full file system tree called *rootfs*.

3) Do Overlays on Rootfs

We leverage *overlayfs* to implement the file-level COW and finally generate the top layer for container. In fact, it is also the underlying technique of the Overlay driver adopted by Docker. Through this new architecture design of Cider, we can eliminate the clone operations during deployment and assign COW jobs to each worker node. Therefore, the contentions introduced by concurrent clones on global metadata of Ceph will be diminished. Also, we can observe that the local COW implemented by overlayfs is far more efficient than the RBD clone. Thus we can use it as another optimization to better the performance and scalability for concurrent deployment, especially in the multi-nodes scenario.

Memory Optimization. The COW mechanism implemented by Ceph internally is in Ceph Object granularity. This mechanism is under local file system. Thus, the OS cannot recognize the same file data block in different containers generated by the same image as one data block. In this context, caching will perform repeatedly when reading the same data block. Running N containers on the same node will consume N times more page cache than running one container.

Using *overlayfs* as the COW mechanism for container layer can facilitate the problem-solving. Figure 5 describes the details of *overlay* operations in Figure 4. The image RBD is directly mounted to a local directory, which is the “lower directory” of the *overlayfs* stack. All files in “merged directory” are actually hard links that point to the “lower directory”. Read/write operations of the container will be performed on the “merged directory”. If a file needs to be changed, it will be copied up to the “upper directory” first, covering the original file. The introduction above is a brief explanation of how file-level COW works. By means of this method, page cache can be reused when booting many containers from the same image on one node.

Discussion. The page cache reuse problem also exists in other storage drivers that implement COW under file level, such as Devicemapper and Btrfs. Wu et al. [17] solve the problem by endowing the driver with the ability to recognize reads on the same blocks. Harter et al. [18] modify the Linux Kernel to enable the caching on block level. The file-level COW approach we adopted might have overheads to some extent on the copy-up operation in terms of big files. However, the file size in the container rootfs is very small (dozens of KB on average) and the file-modify operation is atypical and occur much less frequently in rootfs (it usually manifests in Docker volumes). Therefore, the reliability of file-level COW brought by its simple design is more valuable in the Docker container case.

IV. EVALUATION

In this section, we evaluate the performance and scalability of Cider against the conventional approach represented by Registry-Overlay2.

A. Experiment Setup

TABLE I. SOFTWARE STACK

Name	Version
Linux Distribution	Ubuntu 16.04.2 LTS
Kernel	4.4.0-71
Docker	17.05-ce
Ceph	12.0.3

Environment - A group of experiments were conducted in a cluster with 12 identical physical machines to evaluate the effectiveness of the proposed deployment approach. Each machine has two Intel Xeon E5-2650v4 (12 cores) processors and 256 GB memory. The machines interconnect with each other in 10 Gbps network. The software stack we use is listed in Table I. We deploy Ceph on the cluster with 3 Monitor nodes and 12 OSD nodes. Namely, each OSD occupies one machine. The disk for OSD storage is a 4 TB SATA HDD. We turn on the Bluestore feature of Ceph, which enables Ceph to manage block device directly (not via local file system). The

Docker code we modified is based on the version 17.05-ce, with Ceph storage driver installed.

Methodology and Metrics - In the experiments, we adopt Overlay2 as the local storage driver for the baseline because Overlay driver is widely acceptable as the future option of storage driver and has excellent performance [15][16]. Also, Overlay2 is able to resolve the *inode* exhaustion problem and a few other bugs that were inherent to the old design of Overlay. Furthermore, we use the official Registry application to distribute images. We suppose that the Registry together with Overlay2 is the representative combination of conventional architecture. As for Cider, we abbreviate Cider running completely on network storage as *Cider-network* and that with a local container layer as *Cider-local*.

To demonstrate the system efficiency, we firstly measure the consumed time when deploying a single container and verify the deployment performance of several applications in a single node and multiple nodes respectively. To assess the container execution performance and efficacy, we measure the page cache amount to reflect the cache reuse rate. Finally, we evaluate the impact on running applications by using several benchmarks with diverse parameters. For example, Nginx and Postgres are representative for web servers and database respectively. Python is utilized for programming language. In this context, the throughput statistics are recorded and compared between our approach and the baseline.

B. Experimental Results

1) Single Container Deployment

The top 69 images (see Section II.A) are deployed using Cider. Figure 7 shows the time elapsed for deploying one single container. It is observable that by using Cider-network the deploy time remains stable when image size grows. The average deploy time is merely 2.5 seconds which reduced by 82% compared to that in Registry-Overlay2 (13.4 seconds). These improvements benefits from the loaded-on-demand mechanism. Cider-local has a further performance improvement. More specifically, the deploy time decreases by 18% and 85% compared with Cider-network and Registry-Overlay2 respectively, because of the faster local COW. We can also observe that the consumed time of our method will keep steady even with the increment of the image size.

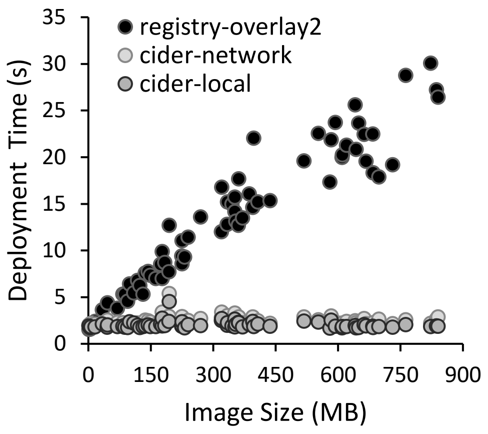


Fig. 7. Time consumed for deploying one container.

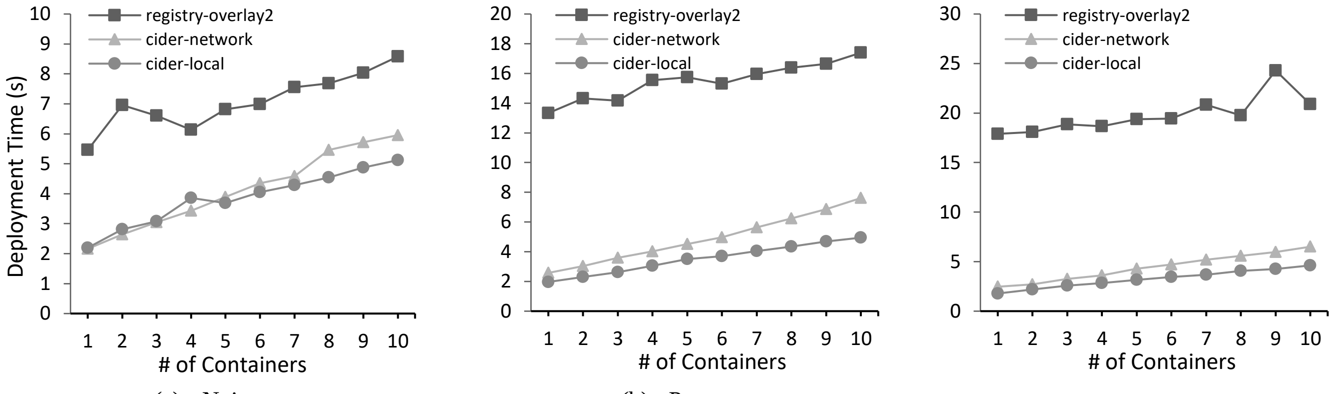
2) Single Node Deployment

In this experiment, we adopt three representative Docker images of different categories - Nginx (109 MB), Postgres (269 MB) and Python (684 MB). We deploy three applications on one single node, using different mechanism: Registry-Overlay2, Cider-network, and Cider-local. The number of containers deployed varies from 1 to 10. All containers in the same round are deployed concurrently.

Figure 8 demonstrates the average time for a container to finish startup under different concurrency conditions. Apparently, Cider outperforms Registry-Overlay2 and there is a substantial decrement of deployment time in all cases from 1 to 10 containers. Additionally, Cider-local can achieve an improved scalability compared with Cider-network because of the highly scalable local COW. For instance, in the scenario of 10 concurrent deploying containers, the consumed time can be reduced by 52% on average with Cider-network, while the mean reduction can reach 63% with Cider-local.

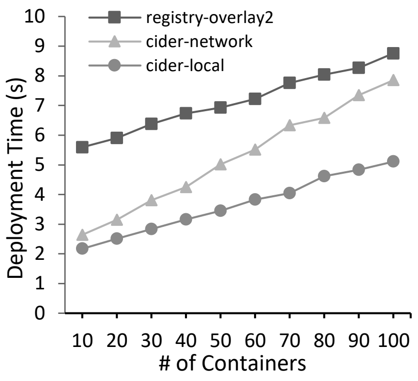
3) Multi-nodes Deployment

We repeat the experiment and vary the number of nodes to verify the system scalability. In each round, the same number of containers will be dispersed onto each node. We also vary the container number (from 1 to 10) on each node to demonstrate the performance impact derived from different configurations. In this context, at most 100 containers will be

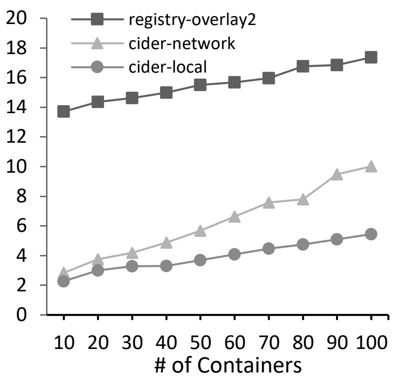


(a) Nginx (b) Postgres (c) Python

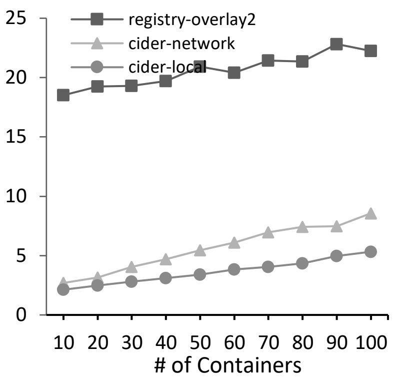
Fig. 8. Average deployment time on a single node.



(a) Nginx



(b) Postgres



(c) Python

Fig. 9. Average deployment time on multiple nodes.

deployed each time.

As shown in Figure 9, the required time for deployment with Cider-network will dramatically soar with the increment of total container number. More precisely, the deployment time of Cider-network is approximately twice as much as that of Cider-local. By contrast, the proposed Cider-local can still achieve a relatively steady performance gain no matter how many containers are launched from 10 to 100 containers. The results can be regarded as a good demonstration of the scalability enhancement brought by container layer localization. Even when 100 concurrent containers are simultaneously deployed, the time is no more than 5 seconds, with an approximately 62% reduction compared with the baseline.

4) *Page Cache Consumption*

We count the page cache consumed by Overlay2, Cider-network and Cider-local respectively when deploying 1 container and 20 containers from the Python image in the same node. The page cache consumption value is obtained by the *buff/cache* column shown by *free* command. The experiment is conducted in a virtual machine outside the cluster to eliminate the cache interference brought by Ceph. Page cache will be cleaned each time before conducting the experiment.

Figure 10 depicts the page cache statistics. The page cache consumptions are close when deploying one container. However, the pattern is quite different when 20 containers are deployed. In fact, values of Overlay2 and Cider-local are similarly close to each other, but the consumption of Cider-network grows rapidly. The consumed value is actually proportional to the number of containers. This observation conforms to our analysis in section IV. B. The reason is both Overlay2 and Cider-local implement COW on file level hence they are able to reuse page cache when access the same file, while Cider-network's COW mechanism is under file system, results in excessive cache allocation for same data.

5) *Application Running Performance*

Apart from the direct deployment effects, the running performance of applications on Cider is another big concern due to the fact that data blocks are lazily fetched from network. To quantitatively measure the runtime effects, we utilize several benchmarks with different workloads - For Nginx, web page requesting is conducted using wrk [9]; For Postgres, TPC-B transactions are submitted by using pgbench [10]; For Python, sudoku puzzles solving program [11] is utilized. All workloads are executed for 5 minutes.

Figure 11 describes the normalized throughput for each application. Due to no obvious differences between Cider-network and Cider-local, we do not distinguish them and merge the results in this experiment. As shown in the figure, throughputs of Overlay2 and Cider are very close, only Python has very slight performance loss (which is less than 0.3%).

V. Related Work

Container deployment is a relatively new topic arising from container management in clusters or clouds. P2P technique is widely adopted to balance the load on central repository and accelerate package distribution, which has been

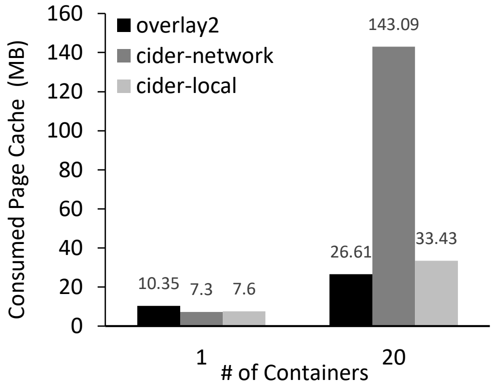


Fig. 10. Consumed page cache.

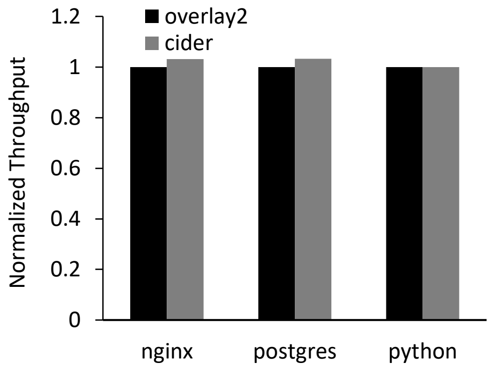


Fig. 11. Application-level running performance.

integrated in cluster management system such as Google's Borg [12], Facebook's Tupperware [13], Alibaba Fuxi[25]. However, they are not dedicated for Docker images. VMware's Harbor [19] optimizes the P2P distribution technique for Docker by setting up multiple replicated registries and transferring data in the granularity of layers. Nathan et al. [22] also introduce a co-operative registry mechanism that enables distributed pull of an image to lower the application provisioning time. Harter et al. [18] speed up container deployment on single node by lazily fetching container data from centralized storage. However, verifications of concurrent deployment at scale have not been conducted in their work.

Virtual machine provisioning is a similar subject with container deployment but has been studied intensively in the cloud environment. Solis et al. [27][28] propose a VM deployment approach by a heuristic strategy considering the performance interference of co-located containers and its impact onto the overall energy efficiency of virtual cluster systems. However, the scalability issue derived from the increasing system and workload scale has not yet been investigated. Wartel et al. [20] use a binary tree and a BitTorrent algorithm to distribute VM images. Nicolae et al. [23] propose an optimized virtual file system for VM image storage based on a lazy transfer scheme. Zhang et al. [24] address the fast provision challenge through downloading data blocks on demand during the VM booting process and speeding up image streaming with P2P streaming techniques. Lagar-Cavilla et al. [21] achieve rapid VM cloning by lazy state replication and state propagation in parallel via multicast. All these findings and innovations inspire the design of Cider.

VI. Conclusion and Future Work

Massive-scale container deployment is becoming increasingly important for micro-service composition and orchestration. In this paper, some preliminary observations reveal that the heave pull operation is the main source of performance bottleneck during the container deployment. We therefore present Cider to tackle the corresponding performance and scalability issues. By means of loading-on-demand network storage for images and local copy-on-write layer for containers, Cider can lower the overall deployment time by 85% and 62% on average respectively in one container and 100 concurrent containers deployment scenarios. Regarding the future work, we plan to further enhance the scalability of Cider to eliminate contentions on local disk existing in concurrent deployment. Mechanisms such as lazy merge of overlayfs layers would mitigate these contentions, thereby shortening the local provision latency.

Acknowledgment

This work is supported by the National Key Research and Development Program (2016YFB1000103), the National 863 Project (2015AA01A202). This work is also supported by Beijing S&T Committee as part of Beijing Brain Inspired Computing Program in BCBD innovation center. For any correspondences, please refer to Dr. Renyu Yang.

References

[1] Container Market Adoption Survey, 2016. https://clusterhq.com/assets/pdfs/state-of-container-usage-june-2016.pdf

[2] Felter, W., Ferreira, A., Rajamony, R., & Rubio, J. (2015, March). An updated performance comparison of virtual machines and linux containers. In Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On (pp. 171-172). IEEE.

[3] Aufs. http://aufs.sourceforge.net/aufs.html

[4] Device-mapper. https://www.kernel.org/doc/Documentation/device-mapper/

[5] Overlay Filesystem. https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt

[6] Docker Hub. https://hub.docker.com/

[7] Docker Registry. https://docs.docker.com/registry/

[8] Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. Linux Journal, 2014(239), 2.

[9] wrk - a HTTP benchmarking tool. https://github.com/wg/wrk

[10] pgbench. https://www.postgresql.org/docs/9.6/static/pgbench.html

[11] Solving Every Sudoku Puzzle. http://norvig.com/sudoku.html

[12] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., & Wilkes, J. Large-scale cluster management at Google with Borg. In ACM Eurosys 2015.

[13] Narayanan, A. (2014). Tupperware: containerized deployment at Facebook.

[14] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D., & Maltzahn, C. Ceph: A scalable, high-performance distributed file system. In USENIX OSDI 2006.

[15] Jeremy Eder. Comprehensive Overview of Storage Scalability in Docker. https://developers.redhat.com/blog/2014/09/30/overview-storage-scalability-docker/

[16] docker-storage-benchmark. https://github.com/chriskuehl/docker-storage-benchmark

[17] Wu, X., Wang, W., & Jiang, S. Totalcow: Unleash the power of copy-on-write for thin-provisioned containers. In ACM APSys 2015.

[18] Harter, T., Salmon, B., Liu, R., Arpaci-Dusseau, A. C., & Arpaci-Dusseau, R. H. Slacker: Fast Distribution with Lazy Docker Containers. In USENIX FAST 2016.

[19] Harbor. https://github.com/vmware/harbor

[20] Wartel, R., Cass, T., Moreira, B., Roche, E., Guijarro, M., Goasguen, S., & Schwickerath, U. Image distribution mechanisms in large scale cloud providers. In IEEE Cloudcom 2010.

[21] Lagar-Cavilla, H. A., Whitney, J. A., Scannell, A. M., Patchin, P., Rumble, S. M., De Lara, E., ... & Satyanarayanan, M. SnowFlock: rapid virtual machine cloning for cloud computing. In ACM EuroSys 2009.

[22] Nathan, S., Ghosh, R., Mukherjee, T., & Narayanan, K. CoMICon: A Co-Operative Management System for Docker Container Images. In IEEE IC2E 2017.

[23] Nicolae, B., Bresnahan, J., Keahey, K., & Antoniu, G. Going back and forth: Efficient multideployment and multisnapshotting on clouds. In ACM HPDC 2011.

[24] Zhang, Z., Li, Z., Wu, K., Li, D., Li, H., Peng, Y., & Lu, X. (2014). VMThunder: fast provisioning of large-scale virtual machine clusters. In IEEE TPDS, 2014.

[25] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu. Fuxi: a fault tolerant resource management and job scheduling system at internet scale. In VLDB 2014.

[26] Yang R, Xu J. Computing at massive scale: Scalability and dependability challenges. IEEE SOSE 2016.

[27] Solis I, Yang R, Xu J & Wo T. Improved Energy-Efficiency in Cloud Datacenters with Interference-Aware Virtual Machine Placement. IEEE ISADS 2013.

[28] Yang R, Solis I, Xu J & Wo T. An Analysis of Performance Interference Effects on Energy-Efficiency of Virtualized Cloud Environments. IEEE Cloudcom 2013.