Gear: Enable Efficient Container Storage and Deployment with a New Image Format

Hao Fan*, Shengwei Bian*, Song Wu*, Song Jiang†, Shadi Ibrahim§, and Hai Jin*
*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, China
†Department of Computer Science and Engineering, University of Texas at Arlington, U.S
§Inria, Univ. Rennes, CNRS, IRISA, France
{fanh, swbian, wusong, hjin}@hust.edu.cn, song.jiang@uta.edu, shadi.ibrahim@inria.fr

***Abstract*—Containers have been widely used in various cloud platforms as they enable agile and elastic application deployment through their process-based virtualization and layered image system. However, different layers of a container image may contain substantial duplicate and unnecessary data, which slows down its deployment due to long image downloading time and increased burden on the image registry. To accelerate the deployment and reduce the size of the registry, we propose a new image format, named Gear image, that consists of two parts: a Gear index describing the structure of the image's file system and a set of files that are required when running an application. The Gear index is represented as a single-layer image compatible with the existing deployment framework. Containers can be launched by pulling a Gear index and on demand retrieving files pointed to by the index. Furthermore, the Gear image enables a file-level sharing mechanism, which helps remove duplicate data in the registry and avoid repeated downloading of identical files by a client. We implement a prototype of the container framework, named Gear, supporting the new image format. Evaluation shows that Gear saves 54% storage capacity in the registry, speeds up container startup by up to 5×, and reduces 84% bandwidth demands.**

***Index Terms*—container, image format, deployment time, registry**

I. Introduction

Container-based virtualization has been widely used to enable isolated execution environments for applications [1–3]. This is due to its advantages of fast startup and low resource consumption [4, 5]. Take the most popular container framework, Docker [6] as an example. Docker packages an application and its dependencies into a self-contained file system named image. Users can upload (push) their images to a registry, such as the Docker Hub [7], for storage and sharing, or download (pull) required images from the registry to start container instances. In this way, a user can conveniently store and deploy containers. However, current image format does not support efficient image deployment and storage. On the one hand, long cold-start latency, which is a critical challenge for emerging computing paradigms, such as serverless computing [8], is mainly caused by the image downloading process [9]. On the other hand, the surge in the number of images puts high pressure on the registry in terms of bandwidth and storage capacity.

There are two reasons why the image format cannot support efficient image storage and deployment. (1) Regarding container storage, there is substantial redundant data between different layers of images, which results in a large waste of storage space [10, 11]. Although layered images allow layer-level deduplication to reduce storage footprint, there is still substantial data redundancy that cannot be detected and removed at this coarse deduplication granularity. (2) Regarding container deployment, an image often contains substantial unused data when launching an instance, which results in slow deployment. In addition, images cannot share files among each other and identical files are downloaded repeatedly.

There have been many works focusing on accelerating container deployment and saving space in the registry. A general approach to accelerate container deployment is to read data on demand through remote images [12–15]. However, these works lack either compatibility or flexibility, because they need to adhere to a specific file system [12, 13] or build a fixed-size virtual block device for each container [14] to enable on-demand downloading. Besides, they require significant modifications to the container I/O stack which prevents these methods from being widely applied. As for the registry, previous works [10, 16, 17] mainly use deduplication to save space. However, these works neither reduce bandwidth demands nor accelerate the deployment of a container, because an entire image has to be reconstructed and downloaded. As the image format dictates how containers are stored and deployed, we design a new image format that is compatible with the current Docker framework to speed up container deployment under existing container I/O stack and save space in the registry. On the one hand, only required data are downloaded to deploy containers in the new format [9, 18–20]. On the other hand, the new image allows file-level sharing among images [11, 21] to eliminate duplicate data in the registry as well as further reduce data to be downloaded in clients. To the best of our knowledge, this is the first work on improving both storage and deployment efficiencies of containers while being compatible with the current Docker framework.

In particular, the proposed image format, named Gear image, introduces an index structure to support on-demand downloading of selected image data. A Gear image consists

of two components, a Gear index and a set of Gear files. The Gear index retains the directory structure in the corresponding Docker image. The actual files in the Docker image are taken out and stored separately as Gear files. In place of the index where an entry for a regular file should be stored, we record the file's MD5 hash value. This regular file is then identified with its hash value. This decoupling of index structure and regular files enables file-level sharing for saving storage space. The process of starting a new container instance can be very fast, because only a very small Gear index (usually less than 1MB) needs to be retrieved before starting the container, and required Gear files will be retrieved on demand.

In summary, we make the following contributions:

- We analyze the impact of image format on container deployment and the effectiveness of file-level sharing in both registry and client.
- We propose a new image format (Gear image) that can run on top of existing container I/O stack, and design Gear framework that can store and deploy Gear images. Gear can ensure that files in images are shared, the required files that cannot be found in local images are downloaded on demand.
- We integrate Gear into Docker, implement a system that supports the storage and deployment of the Gear images compatible to the Docker framework, and evaluate the efficiency of container storage and deployment. Experimental results show that Gear can save the storage footprint of the registry by 54%, reduce bandwidth demands by 84%, and increase the container start speed by 1.6× and 5× under high and low bandwidths, respectively.

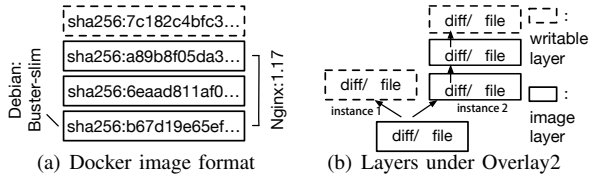
The rest of this paper is organized as follows. Section II describes how Docker stores and deploys containers, and why Docker image format is not efficient. Section III describes the design of Gear. Section IV describes the implementation of Gear. Section V presents extensive evaluation of a Gear prototype implementation. Section VI discusses the related work. And section VII concludes our work.

II. Background and Motivation

In this section, we describe the Docker image format (§II-A), introduce how Docker stores (§II-B) and deploys (§II-C) containers, and explain why Docker image format is not efficient and the consequences.

A. Docker Images

A Docker image is a read-only template for creating a container. It is composed of a series of *layers* that are stacked together. Each layer is identified by its *digest*, the SHA256 hash value of the layer's content. When launching a container, a writable layer will be created at the top of the image layer stack, and all changes (e.g., creation or modification of files) will be preserved in the writable layer, which is achieved with the *Copy-on-Write* (COW) mechanism. Using the "commit" command, a writable layer can be turned into a read-only layer to produce a new image from the container instance. Multiple images may have common layers. Figure 1(a) shows



(a) Docker image format (b) Layers under Overlay2

Fig. 1. Docker image format and its storage form under Overlay2. (a) Two images share the bottom layer; and (b) Layers of an image are COWed. And each image has its writable layer when its instance is launched.

a "Debian:buster-slim" image that has only one layer. Another image ("Nginx:1.17") is built on top of the "Debian:buster-slim" image and has two additional read-only layers. While "Nginx:1.17" has been used to launch a container, it has a writable layer on the top.

B. Storage of Docker Images

Docker images are stored in a centralized registry. The registry can be either a public (e.g., Docker Hub [7]) or a private one. In the registry, layers are stored in the form of compressed tarballs so that the storage space at the registry can be reduced. In addition to the layers, a JSON file named *manifest* is also stored in the registry together with the image. It records configuration information about this image. Among all the information, the most important one is the digests of the image's layers. With the layered image format, different images can be deduplicated at the granularity of layer. Layer-level deduplication is carried out by comparing the digests of the layers to be stored with the digests of the layers already in the registry. Unique layers will be sent to and stored in the registry. By combining layer-level deduplication and in-layer compression, Docker can reduce the storage footprint of the registry by 3.54× [21].

C. Deployment of Docker Containers

There are two steps for deploying (i.e., launching) a container instance, which are to download the image and then to use the image to start the instance. In the first step, the Docker daemon at a client retrieves the manifest of the target image and then downloads layers that are not yet present at the local storage. The graph driver is responsible for saving the image in the local storage and making image layers locally available for reuse. In the second step, the Docker daemon configures and launches the container instance. The graph driver is responsible for providing a complete and correct root file system for the container.

In particular, the graph driver determines how to store the layers and to construct the root file system. The graph driver is implemented based on the underlying file system, or format of the image layers. Currently, the most widely used, and also the officially recommended graph driver is the Overlay2 [22]. Figure 1(b) shows the storage format of the two images shown in Figure 1(a) and two containers launched from them using Overlay2. Each layer is represented as a directory, and each directory has two core components: the *diff/* directory and the lower file. All data of a layer are stored in the *diff/* directory in its original form (e.g., files, directories, symbolic links). The

TABLE I
THE WORKLOADS

Linux Distro:	alpine, amazonlinux, busybox, centos, debian, ubuntu
Language:	golang, java, openjdk, php, python, ruby
Database:	cassandra, couchbase, crate, elasticsearch, influxdb, mariadb, memcached, mongo, mysql, postgres, redis
Web Component:	consul, eclipse-mosquitto, haproxy, httpd, kibana, kong, nginx, node, telegraf, tomcat, traefik
Application Platform:	drupal, ghost, jenkins, nextcloud, rabbitmq, solr, sonarqube, wordpress
Others:	chronograf, docker, gradle, hello-world, logstash, maven, registry, vault

lower file holds the information of all the parent layers of the current layer. The writable layer of the container has the same structure as the normal layers. When launching containers, Overlay2 mounts all read-only and writable layers to a mount point. Through the mount point containers can access the root file system.

D. Motivation

On-demand remote image format. Currently, when launching a Docker container at a server, the entire image has to be downloaded beforehand. The time spent on the potentially lengthy downloading process may be unacceptable for the container deployment. For example, in CI/CD [23] and Dev/Ops [24] scenarios, container versions can be updated frequently [25, 26], and old images have to be replaced quickly by new images for security and performance. Accordingly, researchers propose remote image formats [12–14] that only download a small portion of data (about 6.4%-33.3%) on demand. However, existing remote image formats (i.e., file system-based and block-based) show either weak compatibility or poor flexibility. A file system-based remote image usually adheres to a specific file system [12, 13] (i.e., NFS or CIFS) to enable on-demand data access. This approach makes POSIX-compliant feature an unnecessary constraint for non-POSIX workloads, such as serverless applications. Furthermore, some desirable features (i.e., snapshot and compression) of popular file systems are missing for NFS and CIFS [14]. A block-based remote image needs to build a virtual block device for each container to enable layered image. This means that containers cannot share data among each other, potentially reducing the efficiency, and the size of virtual block device cannot be adjusted according to actual image size. Furthermore, existing remote image formats require significant modifications in I/O stack, i.e., designing new storage drivers, adhering to block devices, or specifying file systems. This holds back the wide adoption of these methods. Accordingly, we manage to design a remote image format based on Docker's default and preferred storage driver, Overlay2, which can be built on various file systems (i.e., EXT4, ZFS, BtrFS) for high compatibility and flexibility.

Management granularity of remote image format. We manage remote images in file-level for the simplicity and applicability of our design, because managing an image in chunks needs to introduce a virtual block device [14] or a specific file system [12, 13]. Furthermore, the management granularity of remote image not only affects the downloading efficiency of an image, but also determines the deduplication ratio in the registry. With the popularity of containers, registries have to manage a growing number of images. Docker Hub alone stores two million images with a total size of 1PB [27]. As a result, organizations spend an increasing amount of their storage and networking infrastructure on operating image registries.

TABLE II
STORAGE USAGE AND NUMBER OF OBJECTS UNDER DIFFERENT DEDUPLICATION GRANULARITIES

	No	**Layer-level**	**File-level**	**Chunk-level**
Storage Usage	370 GB	98 GB	47 GB	43 GB
Object Number	971	5,670	639,585	1,0478,675

Deduplication is required to reduce duplicate data among images [28]. To observe the deduplication ratio in different deduplication granularity, we create a private registry by downloading the images listed in Table I. The registry unpacks the layers and removes duplicate data at different granularity (image layer, file, or chunk with a chunk size of 128KB) and compresses data also in different granularities. Table II shows results for various deduplications. Compared to the images without any deduplication, space demand of layer-level, file-level, and chunk-level deduplication reduced by 74%, 87%, and 88%, respectively. With the number of images increasing, duplicate data grows dramatically in the registry. It is reported that when the total size of unpacked images is 167 TB (unpacked from 47 TB compressed images), only 3.2% of files left after file-level deduplication, which in total only occupy 24 TB [11]. While file-level and chunk-level deduplication show similar space-saving effort, chunk-level deduplication can cause a dramatic increase in the number of unique objects, i.e., the number of unique objects of chunk-

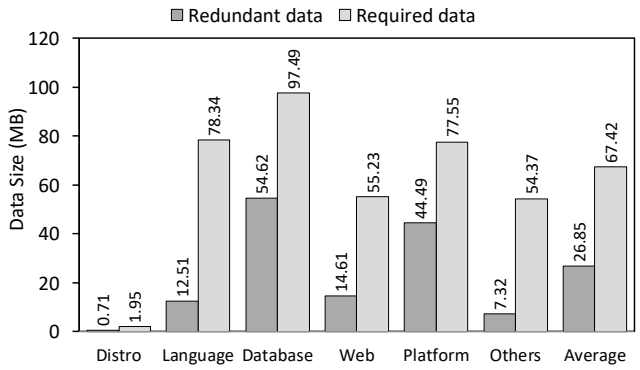


Fig. 2. Redundancy between necessary data among different images in a common image series

level deduplication increases by 16.4× compared to that of file-level deduplication, leading to high management cost. Our image format chooses to support file-level sharing for higher space-saving in deduplication.

Data sharing when launching different containers. We study the redundancy among sets of the necessary files required to launch containers from images in a common image series listed in Table I. Different images in the same series perform the same set of functions as we evaluate the deployment efficiency of Gear containers in §V-D after the launching. Figure 2 shows the amount of redundant data among the necessary data for launching the containers belonging to the same image series. As shown, the *Database* and *Platform* containers have the highest redundancy ratios (56.0% and 57.4%, respectively). On average, the redundancy ratio is 39.9%. This indicates that a local cache for file-level sharing can further accelerate deploying containers. Especially when deploying a new version of a container at a client with its old versions, only 60.1% of the necessary data needs to be downloaded on average.

Summary. Existing on-demand image formats cannot be widely adopted due to their compatibility or flexibility issues. By designing a new image format that can eliminate duplicate data in the file-level and only download locally missing files on top of Overlay2, we can save storage space at the registry and accelerate container deployment with high compatibility and flexibility.

III. The Design

From the above analysis, we see the opportunities of improving efficiency of container storage and deployment. However, these opportunities can only be exploited when the image data access and storage are in the granularity of files, which is not supported by the existing Docker image format. In this section, we introduce a new image format (§III-B), named Gear images, and describe the Gear framework to implement Docker-compatible storage (§III-C) and deployment (§III-D).

A. Overview

We design a Gear framework to enable four features: (1) storage of images at the file-level; (2) on-demand retrieval of only necessary files in an image; (3) sharing of files from different images in a local cache; and (4) full compatibility with existing Docker framework. Figure 3 shows the architecture of the Gear framework. It consists of four components: Gear Converter that builds a Gear image; Gear Registry that stores the Gear images; Gear Driver that works with the Docker daemon to deploy Gear containers; and Gear File Viewer that is responsible for starting a Gear container instance with its root file system.

To realize the Gear framework in practice, we need to address several challenges:

- *How to convert a Docker image into a Gear image?* A Docker image consists of Gear files, which are transformed from files in the Docker image, and Gear index, which records the directory structure of the image. Gear can initiate a container instance as long as the Gear index is downloaded (§III-B).
- *How to make Gear images compatible with Docker framework?* A Gear image is accessed through a Gear index. We keep the Gear index in a single-layer Docker image so that we can use Docker commands to store and distribute it (§III-C).
- *How to save Gear containers with efficient data sharing?* Data of containers resides in Gear files. Gear files are moved out from regular images and stored in storage pools in both client and registry. They are accessed using the content addressing via a cryptographic hash function to enable file-level sharing (§III-C and §III-D1).
- *How to provide the container with a correct file system view?* We design a Gear File Viewer based on Overlay2 that can union-mount a read-only Gear index and a writable layer to provide a file system view for a container. Read operations are redirected to Gear files based on Gear indexes, and write operations are issued to the writable layer (§III-D2).

B. Constructing Gear Images

The essential role of an image is to provide a root file system for a container. When we build the Gear image, we decouple the metadata and regular files of the Docker image. The Gear index is made up of metadata that contains the structure of the entire directory tree and metadata of regular files which replace the actual files in directories. The metadata of a file is the fingerprint of the file. Fingerprints are generated by

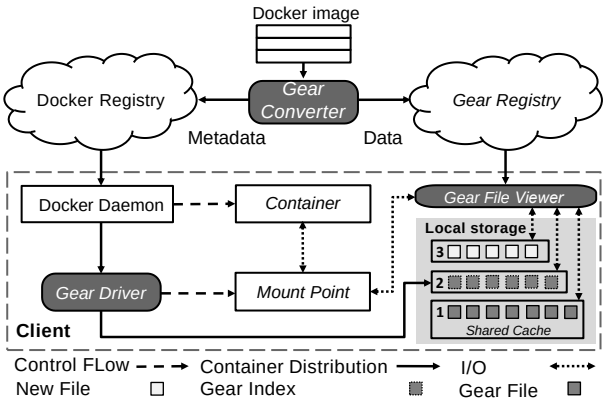


Fig. 3. The architecture of the prototype system

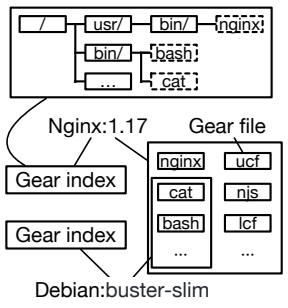


Fig. 4. Gear images

the MD5 [20] hashing of their contents. Regular files from different images are stored in a Gear file pool. These regular files are converted to Gear files by naming (or identifying) them by the fingerprints of the corresponding regular files. A container can access regular files in the pool through fingerprints in the Gear index. Figure 4 shows two Gear images, "Debian:buster-slim" and "Nginx:1.17". All Gear files that belong to the "Debian" image are a subset of Gear files of "Nginx". The index is tiny compared to the size of the rest of an image, i.e., the regular files. A Gear container instance can be launched as long as its corresponding Gear index is downloaded. Note that hash-based fingerprint collision-resistant but may still cause collision. However, the collision probability of MD5 is negligible for current Gear Registry as it is much smaller compared to the probability of having a disk error. According to the "birthday paradox" [29], the probability that one or more hash collisions occur among n files is bounded by:

$$p \leq \frac{n(n-1)}{2} \times \frac{1}{2^m} \tag{1}$$

where m is the number of bits of MD5 fingerprint (i.e., 128-bits). If we convert all images in Docker Hub to Gear images, there will be about 5×10^{10} deduplicated files [11]. Accordingly, the collision probability is about 5×10^{-18} (about 5×2^{-60}). In contrast, in computer systems, the probability of a disk error is about $10^{-12} \sim 10^{-15}$ [30], which is much higher than the collision probability of MD5 fingerprint collisions.

Gear Converter is responsible for automatically building a Gear image from a Docker image. It is in Docker Registry. When a regular image arrives, Gear Converter first retrieves the manifest of the image to obtain information about the image's layers. Since a Docker image is stored as compressed tarballs, the converter decompresses and then saves the layers starting from the bottom layer to the top layer. Finally, the converter traverses the re-constructed file system, and builds the Gear index and Gear files. Gear index and files are stored in Docker Registry and Gear Registry, respectively. The conversion of an image is performed only once. It is carried out in advance which will not affect the pulling of the corresponding Gear image.

It is noted that our design assumes that the probability of collision for a collision-resistant hashing function such as MD5 is 0. In practice, the probability is so low that Docker Hub [6] and IBM Cloud Container Registry [10] have simply used the fingerprints produced by the hash functions to identify files. In cases where concerns over the collision-resistant functions arise, we can detect the collision by comparing file contents after a fingerprint match occurs during the conversion phase. Each file involved in a collision is assigned a unique ID, which is used in the Gear index to take the place of the fingerprint. Note that use of fingerprint, instead of file ID, in the Gear's design is only to enable file deduplication for space-saving. Therefore, disabling the feature for selected files does not compromise the scheme's correctness.

C. Storage of Gear Images

To allow Gear images to be compatible with Docker, which recognizes only layered images for their storage and deployment, we store Gear index and Gear files separately. Gear index is organized as a single-layer Docker image so that it is accessible by Docker commands. In this way, the single-layer image can be managed in Docker Registry in the same way as regular images. We use Docker's "build" command to build a single-layer Gear image. When building the image, it is necessary to copy the environmental variables and the configuration from the original Docker image to the new image to ensure the applications are executed properly.

Different from the regular Docker images, the single-layer Docker image does not contain regular files in the file system represented by the image. Instead, these files are stored in a separate storage pool and can be reached from the Docker managed Gear index. Because MD5 is practically collision-free hash function, identical Gear files have the same fingerprint. In the storage pool, redundant Gear files with the same fingerprint are deduplicated. As files belonging to different images are stored in a common storage pool, deduplication across different images is enabled.

Gear Registry runs a file server to store Gear files. A Gear file can be found through its name (i.e., the fingerprint of the corresponding file). Gear files can be further compressed for higher space efficiency. When uploading a Gear image, we compare fingerprint of file to be uploaded with fingerprints of files already in Gear Registry. Only Gear index and the files whose fingerprints are absent need to be uploaded. When a client needs a Gear file from Gear Registry, it sends the file's fingerprint (available in the Gear index) to Gear Registry, and the file with the same fingerprint will be sent back.

D. Deployment of Gear Containers

When deploying Gear containers, there are two issues to be addressed. The first is how to download and store the Gear images. And the second is how to provide the correct root file system for the container instance. To this end, we design a three-level storage structure and a Gear File Viewer.

1) Three-level Storage Structure: Docker containers only support the sharing of image layers at a local server. An image's top writable layer is to store modifications to an existing image. In contrast, a Gear image adopts a three-level storage structure to enable the sharing of Gear files and the

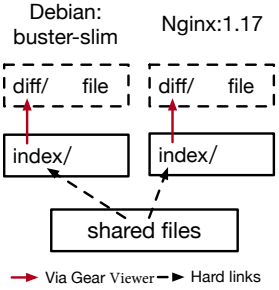


Fig. 5. Storage of Gear

storage of all modifications. Figure 5 shows the three-level storage structure stored on the local file system. The first level is a shared cache of Gear files that belong to different Gear images at a deployment client. Files are deduplicated based on their fingerprints of their contents. The second level stores Gear indexes. The third level (i.e., "*diff*" directories) collects all modifications to the Gear indexes. To deploy a Gear container, the Gear index is first retrieved and the second level is set up (an empty "*index*" directory is created and the Gear index is saved to this directory). Then, the Gear container is launched through the Gear index.

This design has two advantages. First, when searching for local Gear files that can be shared, Gear only needs to conduct the search at the first level, instead of traversing all the image layers. Second, it decouples life cycles of container instances, images, and Gear files. When a container instance is deleted, only the corresponding third level is deleted, and the new container instance can still be launched from the second level. When an image is deleted, its Gear files remain at the first level and can still be shared by other images. As for the file cache at the first level, users can decide how much storage it can occupy and can apply replacement algorithms on it, such as *first-in-first-out* (FIFO) or *least recently used* (LRU). Files that are not linked to Gear indexes are candidates for replacement.

2) Gear File Viewer: We develop Gear File Viewer based on Overlay2 to provide the root file system views for containers. When a container is started, Gear File Viewer union mounts "*index*" and "*diff*" directory as shown in Figure 5. "*Index*" directory is a read-only layer and responsible for providing directory structure of a container. "*Diff*" directory is a writable layer and responsible for recording modifications to container.

A container can access Gear files through Gear File Viewer. If the accessed file is an irregular file, Gear File Viewer returns the result directly from the Gear index. For example, for a request to obtain a symbolic link target ("*path/to/symlink*"), Gear File Viewer first looks for the target symbolic link ("*index/path/to/symlink*") matching the path in the request, and then obtains and returns the target. If the accessed file is a regular file, Gear File Viewer will find the file in the shared cache or Gear Registry based on the fingerprint in "*index*" directory. If the corresponding Gear file is in the shared cache, it is hard linked to the index and its corresponding fingerprint file in the second level is updated, accordingly. This ensures that the Gear container can serve the following requests for the same file from the index without searching the first layer again. Otherwise, the Gear file is in Gear Registry. It is downloaded on demand, stored at the first level, and hard linked to the index. In this way, the Gear container does not need to download files that exist locally or are not used. "*Index*" directory is read-only so that it can start multiple container instances concurrently. When we want to modify or delete a Gear file, Gear File Viewer creates the file or a *whiteout* file in "*diff*" directory (i.e., writable layer) so that the newly created file can be operated directly. If we want to commit the container as an image, Gear File Viewer first extracts the files' contents in "*diff*" directory to construct Gear files. Then, Gear File Viewer combines the metadata of newly added files with the Gear index of current image to build a new Gear index. Finally, Gear pushes the new Gear index and newly added Gear files belonging to the new image to Docker Registry and Gear Registry, respectively.

IV. The Implementation

The prototype system has been implemented based on Docker 18.09.6 and Linux Kernel 4.15.18 using about 2500 lines of Go code and 200 lines of C code. Among the Go codes, about 300 LoC are in Gear Registry to serve requests for storing and retrieving the Gear files. Gear Registry and Docker Registry are deployed on the same node. The managers can store a regular image in Docker Registry. If they want to enable on-demand downloading, they can convert the image to Gear index and Gear files. The Gear index is stored in Docker Registry and the Gear files are stored in Gear Registry. The original Docker image can be removed if the managers want to save storage space. Gear Registry is based on MinIO [31], a high performance object-based storage server. We provide three HTTP interfaces: query, upload, and download. Through these interfaces, users can query whether the target Gear file already exists in Gear Registry, and upload/download the target Gear file to/from Gear Registry. About 1200 LoC are used to implement the Gear Converter for converting Docker images into Gear images. The conversion is carried out through the Docker API. The Gear Driver takes approximately 1000 LoC based on Docker graph driver plugin SDK and Docker Overlay2 driver source code. We directly reuse the interfaces of Overlay2, because the Gear index can be regarded as a normal layered image. All components in the system communicate with each other via HTTP.

All C codes are used to modify Overlay2. Gear File Viewer is implemented based on Overlay2. When the container accesses a file, the *ovl_lookup_single()* function is called to get the *dentry* of the target file. Through the *dentry*, it can actually read file's content. But the file in the Gear index cannot be directly read. Therefore, we modify the *ovl_lookup_single()* function so that the function can pause itself when detecting a fingerprint file and wait for a response after sending a request to a user-mode process (which is responsible for making the target file readable whether linking Gear file from local cache or downloading through network). After receiving the response, it resumes its execution of subsequent codes.

V. The Evaluation

In this section, we evaluate the efficiency of storage, bandwidth, and deployment of Gear containers.

A. Experimental Setup

Hardware. We run our experiments on two identical servers, one running the registries and the other running the Docker daemon. Each server is configured with 2.3GHz Xeon

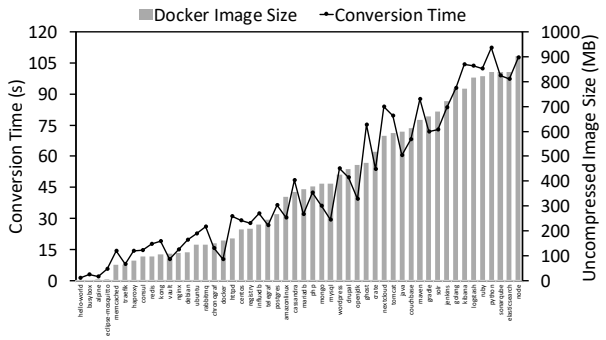


Fig. 6. Average conversion time of each image series

CPUs (E5-2620), 64GB RAM, an Intel Gigabit CT PCIe Network Adapter (EXPI9301CTBLK), and a hard disk (Western Digital WD60PURX). The network bandwidth between the two servers is measured to be 904 Mbps.

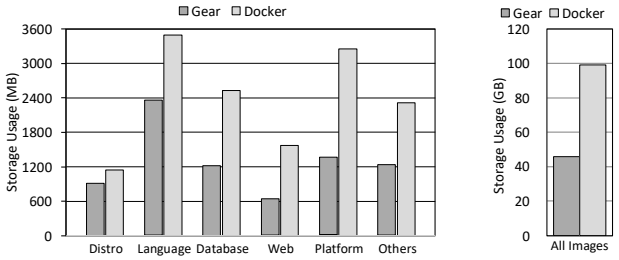
Workloads. Table I lists the top 50 most popular official image series, each consisting of a number of images of different versions, at Docker Hub on November 11, 2020. For each image series, we collected its most recent 20 versions, except for *hello-world*, *centos*, and *eclipse-mosquitto*, which have fewer than 20 versions. For them, we collected all the versions they had. In total, these 50 image series contain 971 images for this investigation.

B. The Image Conversion Time

Before storing and deploying Gear containers, we need to construct Gear images from the corresponding Docker images. Figure 6 shows the conversion time for the Docker images in each image series to be converted into the corresponding Gear images and the relationship between the conversion times and the Docker images' sizes. In the figure we present the results in ascending order of the average uncompressed size of each image series. On average, the conversion time is about 46 seconds. This time is acceptable and will not affect the pulling of the corresponding Gear image as the conversion operation is performed in advance and only once for a given image. Furthermore, if using a faster disk, such as a SSD, or using multiple threads for conversion, the time can be shorter. For example, the conversion time of the node image series can be reduced by 65.7% when using SSDs (from 105s to 36s). Besides, we can see that the conversion time is proportional to the image size, as it includes the file system traversal time and the Gear image build time. In Docker images, files are usually small (less than 1MB). Therefore, the larger the image size, the more files in the image, which will result in longer file system traversal time and longer Gear image build time, and accordingly a longer conversion time.

C. Storage Saving

We build private Gear registries and Docker registries, and evaluate their respective storage demands. First, we compare the storage footprints for Gear registries, including both Gear indexes and Gear files, and Docker registries for each image series. Figure 7(a) shows the results. Docker images are



(a) By Category (b) All Images

Fig. 7. Storage saving of Gear images

compressed and shared at the layer-level while Gear images are compressed and shared at the file-level. We can see that the Database, Web Component, Application Platform, and Others images have storage savings of 52.2%, 60.9%, 58.6%, and 46.7%, respectively. The storage savings of the Linux Distro and Language images are only 20.5% and 32.8%, respectively. This is because the former four types of images are application images. Only the application data may be different among different versions of the same image. The data, such as environments, can be of the same in multiple versions. The Linux Distro images and Language images are usually used as the base images. When they are updated, most of the data in the images may be changed.

Second, we compare the storage footprints for registries that store the top 50 most popular images series. As shown in Figure 7(b), when storing all images together, not only the redundant files between different images of the same image series but also the redundant files between different image series can be deduplicated. Gear Registry can save 53.7% of storage space compared to Docker Registry's layer-level compressing and sharing. For Docker Hub and other public/private large-scale registries, where hundreds of terabytes of images are stored, more space can be saved [11]. Notably, across all image series, the Gear indexes are very small (i.e., only about 0.53MB on average). The total size of all Gear indexes is 515MB, which only occupies 1.1% of total Gear images.

D. Bandwidth Saving

To compare the bandwidth efficiency of Gear containers and Docker containers, we deployed different types of images to perform different types of tasks. Specifically, the Linux Distro containers execute the "echo hello" commands. The Language

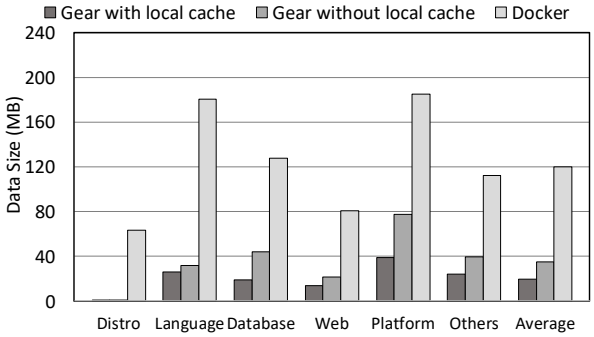


Fig. 8. Bandwidth usage during container deployments

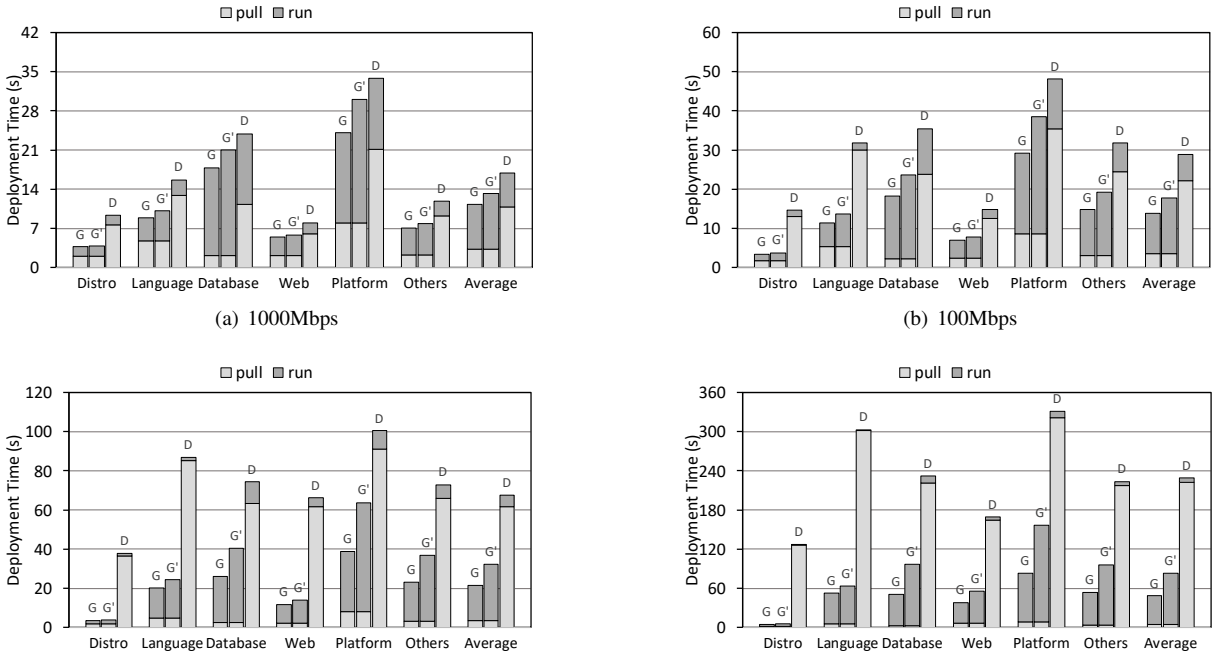


Fig. 9. Deployment time under different network bandwidth. Average pull and run times are shown for each set of images. Bars are labeled with a "*G*" for Gear, "*G'*" for Gear with no local cache and "*D*" for Docker.

containers compile and run a "hello world" program written in the corresponding languages. The Database containers perform additions, deletions, updates, and queries on a database. The Web Component containers start a web server and respond to a request. The Application Platform and Others containers complete their specific tasks. We evaluated the deployment time of Gear in two scenarios. In the first scenario, the Gear maintains and uses its locally cached files. In the second scenario, the Gear's local cache is emptied before each deployment, so that all Gear files have to be remotely downloaded.

We break down the results by category in Figure 8. Compared to Docker, Gear without a local cache reduces data transmission by 70.9%. We can also see that different containers in a common image series access some common files during deployment and the proportion of the common files reaches 44.4% of the total accessed files. By setting up a local cache, only 16.2% of the data has to be remotely retrieved to deploy a container as compared to Docker.

E. Deployment Time

In this section, we evaluate deployment time in different scenarios. First, we evaluate the deployment time of different types of containers. Second, we evaluate the deployment time of different versions of containers.

1) Deployment Time of Different Types of Containers: Containers can be launched while downloading the required Gear files lazily. As shown in Figure 9(a), the deployment speeds of Gear both with and without a local cache are faster than that of Docker. Specifically, Gear without a local cache improves the deployment time by 1.4× on average. And Gear with a local cache improves the deployment time by 1.64× on average as it can access some of its required files locally. The process of deploying a container has two phases: pull (i.e., downloading the Docker images or Gear indexes) and run (i.e., running the container). In the figure, the pull phase of Gear is shorter than that of Docker. But its run time is longer. This is expected, because in the pull phase Gear only needs to download a small Gear index. But Docker needs to download the entire Docker image. Gear takes more time in the run phase, because they need to get the required Gear files on demand during the run phase.

We also evaluate the deployment performance under various bandwidth conditions. Figures 9(b), 9(c), and 9(d) show results under 100Mbps, 20Mbps, and 5Mpbs, respectively. With a lower bandwidth, containers spend more time downloading the images. Gear accordingly achieves better performance improvements. For example, in terms of the time averaged over all images, Gears with and without a local cache are 2.61× and 1.92× faster than Docker, respectively, on the 100Mbps network. Their speedups reach 3.45× and 2.23× under 20Mbps, respectively. With an even lower bandwidth (5Mbps), the improvements reach up to 5.01× and 2.95×, respectively. This means Gear can significantly improve container deployment under bandwidth limited scenarios such as edge/fog computing and IoT. For high bandwidth, pulling phase is not the major bottleneck of deployment. For example, an 1GB image (i.e., the biggest image of top-50 images in Docker Hub) can be transferred in 1 second when bandwidth is 10Gbps. Gear and other on-demand retrieval image formats show no obvious advantage compared to original Docker in terms of pulling time in high bandwidth. Nevertheless, Gear

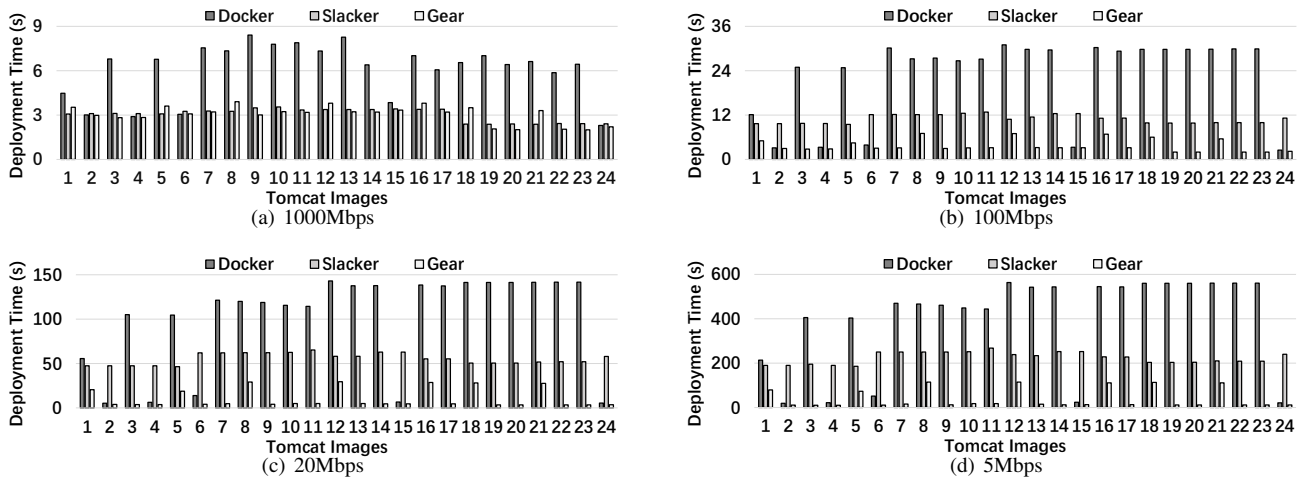


Fig. 10. Deployment time when different versions of Tomcat are deployed one by one. The *x*-axis indicates the deployment sequence of Tomcat containers of different versions.

can still reduce data transmission and save storage space in clients.

2) Deployment Time of Different Versions of Containers: In practice, a client needs to quickly update images to enhance security, add new features, repair bugs, or improve performance. As shown in Figure 10(a), different versions of *Tomcat*, one of the most popular Web server images [27], are deployed under Docker, Slacker, and Gear, separately. Slacker is a block-based remote image and realized based on LVM and NFS [14]. When bandwidth is 1000Mbps, the average deployment time of Docker, Slacker, and Gear are 6.08s, 3.03s, and 3.04s, respectively. This means Slacker and Gear show similar deployment time. Due to sharing mechanism, some of the later images show decreased deployment time for Docker and Gear. For example, compared to the *1st* container, the deployment time of the *2nd* container is reduced by 31.0% and 15.5% for Docker and Gear, respectively. Even if efficient image sharing allows Docker to achieve similar deployment time compared to Gear and Slacker in some cases, file-level sharing is more efficient. For example, the *20th* container shows slight change in deployment time compared to the *18th* container for Docker, while Gear reduces the deployment time by 42.9%. With the number of deployed containers increasing, Gear's deployment becomes more efficient due to the use of file-level sharing, while Slacker's time shows little change due to the absence of sharing mechanism. For example, for the *1st* container, the deployment time under Gear is longer than Slacker by 15.3%, because Gear is based on Overlay2 which introduces additional layer to I/O stack. While for the *3rd* container, the deployment time under Gear becomes shorter than Slacker by 8.8%.

When network bandwidth decreases, the deployment time of Slacker and Docker becomes worse. For example, when the bandwidth decreases from 1000Mbps to 100Mbps, the average deployment times of Docker and Slacker increase by 2.7× and 2.6×, respectively. While for Gear, the average deployment time only increases by 1.2×. Docker shows long deployment time under low bandwidth, because it needs to download the entire image. Slacker pulls data on demand like Gear. However, the deployment time of Slacker increases drastically. The reason is that the number of blocks to be pulled by Slacker is much more than the number of files to be pulled by Gear as we mentioned in Section II-D. As a result, the speed of block transfer is limited by the bandwidth.

In summary, file-level sharing and pulling make Gear achieve improved deployment time when the number of deployed containers increases, especially in bandwidth-limited environments.

F. Long-running and Short-running Containers

After a container is successfully deployed, it is ready to provide services. We evaluate the service performance of Gear containers and Docker containers for long-running and short-running workloads. For the long-running workloads, we selected two widely used databases (*Memcached* and *Redis*), and two popular web servers (*Nginx* and *Httpd*) to evaluate the throughput. For Memcached and Redis, we use the memtier_benchmark [32], which simulates multiple client-generated requests to the database with a 1:10 SET-GET ratio. For *Nginx* and *Httpd*, we use the Apache ab benchmark [33] which benchmarks web server throughput by sending concurrent requests. Figure 11(a) shows the results. For Redis,

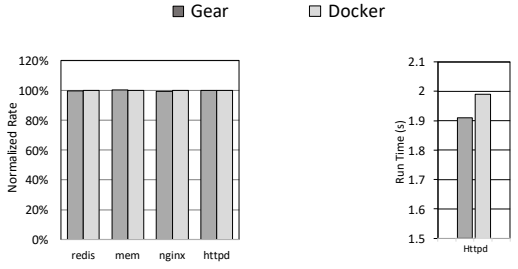


Fig. 11. Long-running and Short-running workloads. Normalized rate refers to the ratio of the rate to that of Docker.

Memcached, Nginx, and Httpd, Gear and Docker have similar performance.

For the short-running workloads, we still choose *Httpd* as the test system. We use a custom benchmark that repeats the process of launching, requesting, and destroying the container 100 times. Figure 11(b) shows the time of each process. It shows that Gear containers have a slight advantage over Docker containers. The reason is that Gear spends less time unmounting the file system, because it only needs to destroy the inode caches of required files.

VI. Related Work

There have been many efforts on improvement of containers' storage and deployment efficiency. However, these two efficiency issues have been addressed separately, missing opportunities for a simple and lightweight solution to both issues at the same time.

A. Efforts on Container Storage

Skourtis et al. [21] argue that Docker image layers in the registry should be reorganized to maximize their overlap and reduce storage consumption. They present a mathematical formulation of the problem, and develop a greedy algorithm to this end. Zhao et al. [11] make a detailed analysis of more than 47 TB compressed Docker images in the Docker Hub. Their results reveal that using file-level deduplication on unpacked images (167 TB in total) can remove 96.8% of duplicate files of the unpacked images, which occupy 24 TB, showing that file-level deduplication has a great potential to save storage space for the registry. This is consistent with our experimental results. However, layers stored in the registry are compressed, which means even very similar layers will become fundamentally distinct after compressing, making deduplication not a good choice in current registry [16, 17, 34]. Accordingly, Zhao et al. [10] propose to do file-level deduplication after decompressing the layers and hide the overhead caused by reconstructing the compressed layers via a content-aware cache. However, existing deduplication methods neither reduce bandwidth demands nor accelerate the deployment of a container, because these methods cannot support on-demand image retrieval and an entire image still has to be reconstructed and downloaded when deploying a container. In contrast, Gear not only effectively reduces image storage space in the registry but also accelerates container deployment.

B. Efforts on Container Deployment

Acceleration of image downloading is one of the key methods to optimize the deployment of containers.

Obtaining an image in an on-demand method. Data to be downloaded before starting a container should be reduced. Accordingly, CNTR [35] extracts the application from the image as a slim image and leaves the remaining data as a fat image. When deploying a container, only the slim image is downloaded, and the fat image is downloaded on demand. Furthermore, remote image formats [12–15] are proposed to launch containers without downloading images in advance. Slacker [12] and Wharf [13] leverage NFS to support downloading required data on demand when containers are launching. Nydus [15] employs a user-level file system, which may degrade native I/O performance, to pull required data in chunk granularity. DADI [14] realizes lazy data retrieval at the block level via virtual block device for high performance and compatibility. Existing remote image formats lack flexibility, because they must be bound to a specific file system or virtual block devices. Besides, they require significant modifications to the container I/O stack which in turn prevents these methods from being widely applied. In contrast, Gear image is based on the existing container I/O stack, which makes Gear compatible with the current Docker framework.

Obtaining an image in a decentralized method. Besides the registry, other servers that can provide image data may be used for higher parallelism. CoMICon [36] and Wharf [13] found that it is time-consuming if each node in a cluster pulls its own copy of the same image from the registry. They set up a cache in the cluster to store the layers that have been pulled, reducing the subsequent access to the remote registry and accelerating the container deployment. Specifically, CoMICon sets up a cooperative cache by placing a cooperative registry on each node in the cluster. Wharf utilizes the distributed file system to store the image layers. Dragonfly [37], FID [38], and DADI [14] introduce *peer-to-peer* (P2P) technology to accelerate deploying containers. These works are orthogonal to the Gear work as using P2P and designing a tightly connected cluster also help speed up the distribution of Gear files.

VII. Conclusion and Future Work

In this paper, we propose Gear, a new image format that supports lazy retrieval of image data and file-level sharing. While many efforts have been made to reduce container storage demand and accelerate container deployment, Gear addresses both challenges in one solution. It transforms a Docker image to its Gear image, which consists of two components (a Gear index and a set of Gear files). In the design, actual files in the Docker images are taken out and stored to enable file-level sharing. Containers can be launched fast with only Gear index downloaded and the Gear files can be retrieved lazily. In addition, a local shared cache can reduce the remote access so that containers can be further accelerated. We implemented a prototype of Gear supporting the new format and extensively evaluated it. Our evaluation shows that Gear can save around 54% in the registry, speed up the container deployment by up to 5×, and reduce bandwidth demands by 84%.

In the future, we plan to enable Gear to read big files on demand in chunks to better accelerate containers that need to download big files, such as AI containers with big models.

Acknowledgments

This work is supported by National Key Research and Development Program under grant 2018YFB1003600, National Science Foundation of China under grants No.62032008 and 61872155. The corresponding author is Song Wu. We also thank Antonio Barbalace for shepherding our manuscript.

REFERENCES

[1] U. Deshpande, "Caravel: Burst Tolerant Scheduling for Containerized Stateful Applications," in *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 1432–1442.

[2] S. Nadgowda, S. Suneja, N. Bila, and C. Isci, "Voyager: Complete Container State Migration," in *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 2137–2142.

[3] F. Guo, Y. Li, M. Lv, Y. Xu, and J. C. S. Lui, "HP-Mapper: A High Performance Storage Driver for Docker Containers," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2019, pp. 325–336.

[4] P. Sharma, L. Chaufournier, P. J. Shenoy, and Y. C. Tay, "Containers and Virtual Machines at Scale: A Comparative Study," in *Proceedings of the International Middleware Conference (Middleware)*, 2016, pp. 1–13.

[5] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. Lightweight Virtualization: A Performance Comparison," in *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*, 2015, pp. 386–393.

[6] C. Boettiger, "An Introduction to Docker for Reproducible Research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[7] "Docker Hub," https://hub.docker.com, 2019.

[8] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, Analysis, and Optimization of Serverless Function Snapshots," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 559–572.

[9] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale Cluster Management at Google with Borg," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2015, pp. 1–17.

[10] N. Zhao, H. Albahar, S. Abraham, K. Chen, V. Tarasov, D. Skourtis, L. Rupprecht, A. Anwar, and A. R. Butt, "DupHunter: Flexible High-Performance Deduplication for Docker Registries," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2020, pp. 769–783.

[11] N. Zhao, V. Tarasov, H. Albahar, A. Anwar, L. Rupprecht, D. Skourtis, A. S. Warke, M. Mohamed, and A. R. Butt, "Large-Scale Analysis of the Docker Hub Dataset," in *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*, 2019, pp. 1–10.

[12] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast Distribution with Lazy Docker Containers," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2016, pp. 181–195.

[13] C. Zheng, L. Rupprecht, V. Tarasov, D. Thain, M. Mohamed, D. Skourtis, A. Warke, and D. Hildebrand, "Wharf: Sharing Docker Images in a Distributed File System," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2018, pp. 174–185.

[14] H. Li, Y. Yuan, R. Du, K. Ma, L. Liu, and W. Hsu, "DADI: Block-level image service for agile and elastic application deployment," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2020, pp. 727–740.

[15] "Nydus," https://github.com/dragonflyoss/image-service, 2020.

[16] Z. Yan, H. Jiang, Y. Tan, and H. Luo, "Deduplicating Compressed Contents in Cloud Storage Environment," in *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2016, pp. 1–5.

[17] Y. Tan, H. Jiang, D. Feng, L. Tian, Z. Yan, and G. Zhou, "SAM: A Semantic-Aware Multi-tiered Source De-duplication Framework for Cloud Backup," in *Proceedings of the International Conference on Parallel Processing (ICPP)*, 2010, pp. 614–623.

[18] B. Tak, C. Isci, S. S. Duri, N. Bila, S. Nadgowda, and J. Doran, "Understanding Security Implications of Using Containers in the Cloud," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017, pp. 313–319.

[19] C. Peng, M. Kim, Z. Zhang, and H. Lei, "VDN: Virtual machine image distribution network for cloud data centers," in *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2012, pp. 181–189.

[20] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2018, pp. 57–70.

[21] D. Skourtis, L. Rupprecht, V. Tarasov, and N. Megiddo, "Carving Perfect Layers out of Docker Images," in *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2019.

[22] Y. Sun, J. Lei, S. Shin, and H. Lu, "Baoverlay: A Block-Accessible Overlay File System for Fast and Efficient Container Storage," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2020, pp. 90–104.

[23] S. Garg and S. Garg, "Automated Cloud Infrastructure, Continuous Integration and Continuous Delivery using Docker with Robust Container Security," in *Proceedings of the IEEE Conference on Multimedia Information Processing and Retrieval (MIPR)*, 2019, pp. 467–470.

[24] J. Henkel, C. Bird, S. Lahiri, and T. Reps, "Learning from, Understanding, and Supporting DevOps Artifacts for Docker," in *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 38–49.

[25] Z. Lu, Y. Wu, J. Xu, and T. Wang, "An Acceleration Method for Docker Image Update," in *Proceedings of the IEEE International Conference on Fog Computing (ICFC)*, 2019, pp. 15–23.

[26] F. Hassan, R. Rodriguez, and X. Wang, "RUDSEA: Recommending Updates of Dockerfiles via Software Environment Analysis," in *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018, pp. 796–801.

[27] "Docker Hub Image Index," https://hub.docker.com/search?type=image, 2019.

[28] "Amazon container registry," https://aws.amazon.com/ecr/, 2020.

[29] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, and J. Kunkel, "A study on data deduplication in hpc storage systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012, pp. 1–11.

[30] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou, "A Comprehensive Study of the Past, Present, and Future of Data Deduplication," *Proceedings of the IEEE*, vol. 104, no. 9, pp. 1681–1710, 2016.

[31] "MinIO," https://docs.min.io/, 2021.

[32] "Memtier Benchmark," https://github.com/RedisLabs, 2020.

[33] "Apache Ab," https://httpd.apache.org/docs/2.4/programs/ab.html, 2020.

[34] "What Is ZFS?" https://docs.oracle.com/cd/E19253-01/819-5461/zfsover-2, 2019.

[35] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci, "Cntr: Lightweight OS Containers," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2018, pp. 199–212.

[36] S. Nathan, R. Ghosh, T. Mukherjee, and K. Narayanan, "Comicon: A co-operative management system for docker container images," in *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*, 2017, pp. 116–126.

[37] "Dragonfly," https://d7y.io, 2020.

[38] K. Wang, Y. Yang, Y. Li, H. Luo, and L. Ma, "FID: A faster image distribution system for docker platform," in *Proceedings of the IEEE International Workshops on Foundations and Applications of Self* Systems, (FAS*W)*, 2017, pp. 191–198.