Large-Scale Analysis of Docker Images and Performance Implications for Container Storage Systems

Nannan Zhao[1], Vasily Tarasov[2], Hadeel Albahar[1], Ali Anwar[2], Lukas Rupprecht[2], Dimitrios Skourtis[2], Arnab K. Paul[1], Keren Chen[1], and Ali R. Butt[1]

Abstract—Docker containers have become a prominent solution for supporting modern enterprise applications due to the highly desirable features of isolation, low overhead, and efficient packaging of the application's execution environment. Containers are created from images which are shared between users via a registry. The amount of data registries store is massive. For example, Docker Hub, a popular public registry, stores at least half a million public images. In this paper, we analyze over 167 TB of uncompressed Docker Hub images, characterize them using multiple metrics and evaluate the potential of file-level deduplication. Our analysis helps to make conscious decisions when designing storage for containers in general and Docker registries in particular. For example, only 3% of the files in images are unique while others are redundant file copies, which means file-level deduplication has a great potential to save storage space. Furthermore, we carry out a comprehensive analysis of both small I/O request performance and copy-on-write performance for multiple popular container storage drivers. Our findings can motivate and help improve the design of data reduction and caching methods for images, pulling optimizations for registries, and storage drivers.

Index Terms—Containers, Docker, Container images, Container registry, Deduplication, Docker Hub, Container storage drivers.

1 Introduction

Recently, containers [1], [2] have gained significant traction as an alternative to virtual machines [3] for virtualization both on premises and in the cloud. Moreover, containers are increasingly used in software development and test automation [4], and are becoming a key part in the release life cycle of an application [5]. In contrast to Virtual Machines (VMs), containers share the operating system kernel but are isolated in terms of process visibility (via namespaces [6]) and resource usage (via control groups [7]). As a result, containers require fewer memory and storage, start faster, and typically incur less execution overhead than VMs [8]–[10].

A driving force of the fast container adoption is the popular Docker [11] container management framework. Docker combines process containerization with convenient packaging of an application's complete runtime environment in *images*. For storage and network efficiency, images are composed of independent, shareable *layers* of files. Images and their corresponding layers are stored in a centralized *registry* and accessed by clients as needed. Docker Hub [12] is the most popular registry, currently storing almost 4 million public image repositories comprising approximately 20 million layers.

While the massive image dataset presents challenges to the registry and client storage infrastructure, storage for containers has remained a largely unexplored area. We believe one of the prime reasons is the limited understanding of what data is stored inside containers. This knowledge can help improve the container storage infrastructure and ensure scalability of and fast accesses to the registry service. Existing work has focused on various aspects of containerization [13]–[18]. However, the registry and its contents have yet to be studied in detail.

In this paper, we perform the first, comprehensive, large-scale characterization and redundancy analysis of the images and layers stored in the Docker Hub registry (§2). We download all `latest` versions of publicly accessible images from Docker Hub (as of May 2017), which amount to 47 TB of compressed image data (§3). Based on that dataset, we analyze the storage properties of images, such as the size and compression ratio distributions of layers and images, directory and file distributions, as well as Docker-specific properties, e.g., the number of layers per image, image popularity, and the effectiveness of layer sharing (§4).

We investigate the potential for data reduction in the Docker registry by using file-level deduplication (§5). We observe that 97% of the files are file duplicates and removing these duplicates can reduce storage space utilization by 50%. This suggests that file-level deduplication has a great potential to save storage space for large-scale registries. To understand why there are so many file duplicates, we further analyze the files stored in images and observe that the majority of files are executables, object files, libraries, source code, and scripts. The deduplication ratio among these file types is high, suggesting that there are a high number of similar images and layers, built for similar applications.

Once retrieved from the registry, images and layers are locally stored and managed by container storage drivers. As the performance of these drivers can largely impact container startup, build, and execution time, we perform a comprehensive analysis on popular storage drivers based on the key insights from our image characterization. First, we analyze the layer `pulling` latency distribution for the images from the Docker Hub dataset and propose various optimizations for speeding up container startup times (§6).

- *N. Zhao, H. Albahar, A. Paul, K. Chen, and A. Butt are with the Department of Computer Science, Virginia Tech, Virginia, VA, 24061.*
- *V. Tarasov, A. Anwar, L. Rupprecht, and D. Skourtis are with IBM Research-Almaden, CA 95120.*

Manuscript received April 19, 2005; revised August 26, 2015.

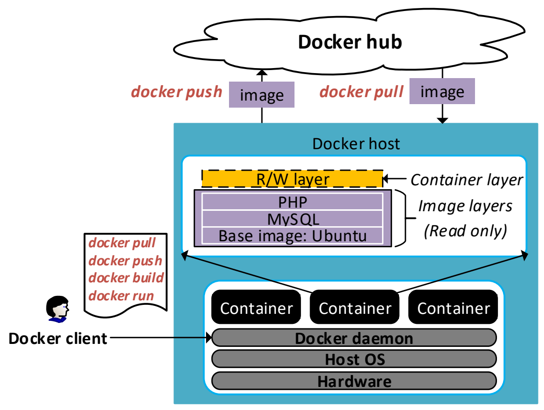


Fig. 1: Docker ecosystem

Second, we observe that files are small while layers and images contain many files. For example, 90% and 50% of files are smaller than 26 KB and 4 KB respectively while 80% of images have more than 15 K files, which implies that the *small I/O request performance* is important for container storage drivers. Therefore, we study the small I/O performance for different popular container storage drivers (§7). Based on our analysis, we derive several design implications for storage drivers, e.g., storage drivers should be optimized for both small reads and small writes, especially overwrites of the files that already exist in read-only layers.

2 Background and Related Work

In this section, we introduce the background on Docker and discuss related work on Docker container analysis.

2.1 Docker

Docker is a popular containerization framework. It automates the creation and deployment of application containers by providing the capability to package an application with its runtime dependencies into a container image and then running this images on a host machine [11]

As shown in Figure 1, the Docker ecosystem consists of several components. Users interact with Docker via the Docker *client*, which sends commands to the Docker *daemon*. The daemon is responsible for *running* containers from locally available images. Additionally, the daemon supports *building* new images and *pushing* them to a Docker *registry*. When a user wants to launch a container from an image that is not available locally, the daemon *pulls* the required image from the registry.

Images and layers. An image is represented by a *manifest* file, which contains a list of layer identifiers (digests) for all layers required by the image. Docker images consist of a series of individual *layers*. A layer contains a subset of the files in the image and often represents a specific component/dependency of the image, e.g., a library. This modular design allows layers to be shared between two images if both images depend on the same component.

Docker registry. The Docker registry is a platform for storing and sharing container images. It stores images in *repositories*, each containing different versions of the same image. Image layers are stored as Gzip compressed archival files and image manifests as JSON files [12].

Container storage drivers. When a user starts a container, the *storage driver* is responsible for creating the container root file system from the layers, e.g., by union mounting the layers into a single mount point. As image layers are read-only to allow sharing across containers, the driver also creates a new writable layer on top of the underlying read-only layers (see Figure 1). Any changes made to files in the image will be reflected inside the writable layer via a copy-on-write (CoW) mechanism.

Docker supports multiple storage drivers, e.g., overlay2 [19], devicemapper [20], zfs [21], and btrfs [22], which efficiently manage read-only and writable layers in a single file-system namespace using CoW [23]. According to the copy-on-write granularity of different storage drivers, we split the storage drivers into two main groups: File-level CoW and Block-level CoW. File-level CoW incurs a performance and space penalty as it copies the entire file from a lower layer when the container modifies the file, even if the modification is small [24], [25]. Compared to file-level drivers, block-level drivers perform CoW at block granularity. Consequently, these drivers provide more efficient usage of disk space and incur less overhead during writes [24].

2.2 Related Work on Docker images and containers.

We structure the related work into two main categories: 1) analysis of container images content, and 2) container performance analysis.

Analysis of container images content. Several works have characterized different numbers of Docker images to understand various properties [14], [16], [17], [26] focusing on quality, vulnerabilities, and security issues. For example, Cito et al. [15] conducted an empirical study to characterize the Docker ecosystem with a focus on prevalent quality issues, and the evolution of Docker files based on a dataset of 70,000 Docker files. Shu et al. [17] studied the security vulnerabilities in Docker Hub images based on a dataset of 356,218 images and found there is a strong need for more automated and systematic methods of applying security updates to Docker images. Compared to the above research, we focus on the storage properties of Docker images (Section 4).

Slacker [16] studied 57 images from Docker Hub for a variety of metrics. The authors calculated the deduplication ratio of these 57 images and also found that layer pulls can increase container startup time. Compared to Slacker, we did a deduplication analysis on the entire Docker Hub dataset (Section 5) and also inspect the content of Docker images to understand the corresponding applications. Moreover, we analyzed layer pulling performance by using different storage options and compression methods in Section 6.

Container performance analysis. Many researchers started to analyze the performance of Docker containers by comparing it to VMs. For example, Seo et al. [27] compared the performance of Docker with KVM in terms of metrics such as boot time and CPU performance while Scheepers et al. [28] compared LXC and Xen virtualization technologies for distributing resources. Felter et al. [29] evaluated and compared three different environments, Native, Docker, and KVM for throughput, CPU overhead, etc. They found that container performance is better than KVM in terms of boot time and calculation speed [27] while KVM and Xen take less time than Docker and LXC to accomplish tasks [29] [28].

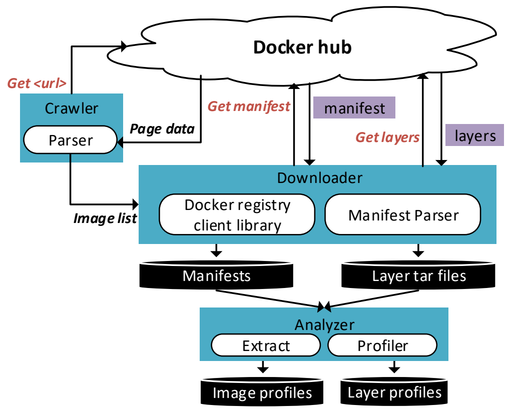


Fig. 2: Crawler, Downloader, and Analyzer

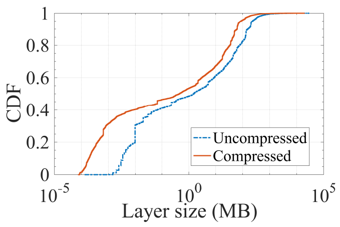


Fig. 3: CDF of layer size

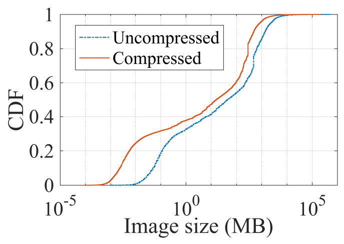


Fig. 4: CDF of image size

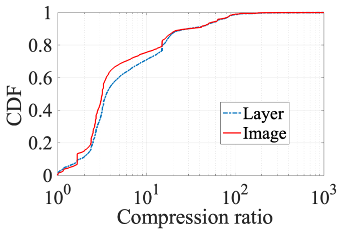


Fig. 5: CDF of compression ratios

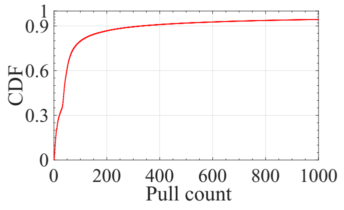


Fig. 6: CDF of image pull count

Moreover, KVM and Xen are better for equally distributing resources and have better performance/resource isolation ability than containers [30]. Compared to the above research, we focus on container storage drivers which is one of the key components for efficient execution of a container [24], [25], [31]–[34]. These works are orthogonal and complementary to our work.

A number of research papers analyzed the performance of container storage drivers. Xu et al. [35] presented a characterization of the performance impact among various storage and storage driver options and provided configuration settings for better utilizing high speed SSDs. Bhimani et al. [13] characterized the performance of persistent storage options for I/O intensive containerized applications with NVMe SSDs. Tarasov et al. [36] discussed the influence of different storage solutions on different IO workloads, running inside containers. Compared with the above studies, our analysis on container storage drivers is based on the real Docker image dataset and we consider the content and storage properties of real images. For instance, we observed that the majority of files in layers are small while images and layers have many files. Hence we focus on the small read and write performance in Section 7. However, the above work mostly measured I/O performance by reading/writing few large files, or using larger I/O requests [13], [35], [36].

3 Methodology and Dataset

Our image analysis methodology consists of three steps (see Figure 2): (1) *crawler* crawls Docker Hub to list all repositories; (2) *downloader* downloads the latest version of an image and all referenced layers from each repository based on the crawler results; (3) *analyzer* decompresses and analyzes images and layers.

We ran the crawler on May 30th, 2017 and it delivered a list of 457,627 distinct repositories (with 200 official repositories). The whole downloading process took around 30 days. Overall, we downloaded 355,319 images, resulting in 1,792,609 compressed layers and 5,278,465,130 files, with a total compressed dataset size of 47 TB. In the paper, we analyzed the compressed image dataset and studied a variety of storage metrics on layers, images, and files.

4 Image Dataset Characterization

We center our image analysis around the following three questions:

1) What are the properties of layers and images in terms of size, compression ratio, popularity, and directory and file distribution?
2) What files do developers store in images and what is the ratio between different types of files?
3) Does redundant data exist inside compressed Docker registry image dataset and what is the deduplication ratio?

4.1 Layer and image analysis

We start with the analysis of size, popularity, and structure for both layers and images to understand the basic properties of the registry dataset.

4.1.1 Size distribution for layers and images

Figure 3 shows the layer size distribution. As expected, layers are small with 90% of the compressed layers being smaller than 63 MB. There are only few large layers. For example, 0.2% of compressed layers are larger than 1 GB. Downloading and decompressing such large layers can significantly slow down container startup times. Once uncompressed, 90% of the layer tar archives are smaller than 177 MB.

To get the image size distribution (shown in Figure 4), we sum up the sizes of all layers for each image. Images are significantly bigger than layers. 90% of the images have compressed sizes less than 0.48 GB and uncompressed sizes less than 1.3 GB. However, we observe that there is a considerable amount of small images. 50% of images are smaller than 17 MB compressed and 94 MB uncompressed.

Figure 5 shows the compression ratio distribution. The compression ratio is high for both layers and images. We observe that 50% of layer compression ratios and image compression ratios are greater than 3.5 and 3.2, respectively. The high compressibility means that layer archives have a great potential for compression to reduce layer transfer latencies.

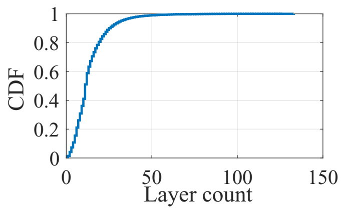


Fig. 7: CDF of layer count in images

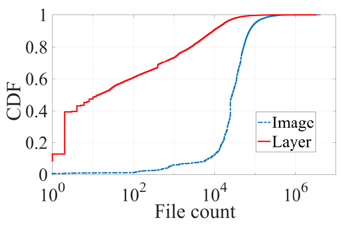


Fig. 9: File count

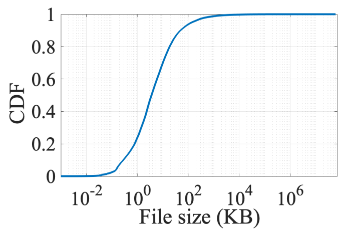
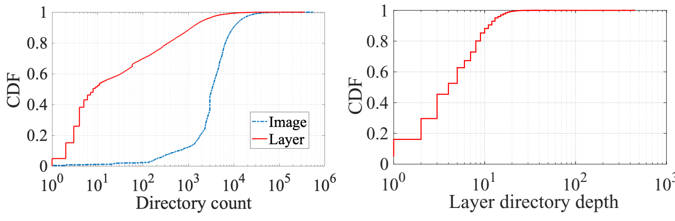
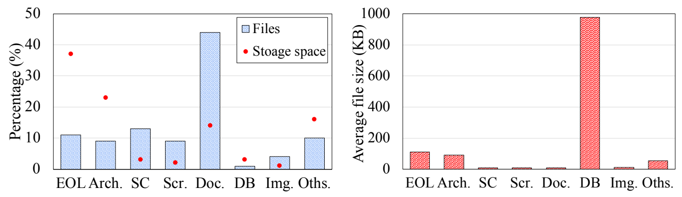


Fig. 10: CDF of file size



(a) Directory count (b) Directory depth

Fig. 8: CDF of directory count and directory depth



(a) File count and Space occupied (b) Average file size

Fig. 11: Commonly used file types

4.1.2 Image popularity

The assess image popularity, we analyze the pull counts reported by Docker Hub web page (see Figure 6). The results demonstrate that there image accesses are heavily skewed. Only 10% of images are pulled more than 333 times while 50% of images are pulled less than 40 times. This confirms that there are indeed popular images which can benefit from caching.

4.1.3 Layer and directory hierarchy

Figure 7 shows the layer count distribution in images. The majority of images have more than one layer and we observe that half of the images have more than 8 layers while 90% of the images have less than 18 layers.

Figure 8(a) shows the directory count distribution for layers and images. The majority of layers and images have more than one directory. Half of the layers consist of more than 11 directories while 90% of the layers have less than 826 directories. For images, 90% have more than 679 directories.

Figure 8(b) shows the maximum directory depth in each layer. The majority of layers have a directory depth greater than 2. 50% of the layers have directory depth greater than 4 and 90% of the layers have directory depth less than 10.

4.1.4 File count and file size distribution

Figure 9 shows the file counts in each layer and image. First, we observe that 77% of the layers have more than one file and half of the layers have more than 30 files. Second, 28% of the layers have more than 1,000 files. This large amount of files will put I/O pressures on the container storage drivers during image building and container execution. Figure 9 shows the file counts for images. Half of the images have more than 1,090 files and 10% of images have more than 64,780 files.

Figure 10 shows file size distribution. As expected, files are small with 90% and 50% of files being smaller than 26 KB and 4 KB, respectively.

4.2 File types

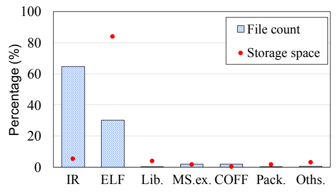
To understand what kind of files are stored in images, we first group files based on file types. Overall, our image dataset consists of 1,500 file types but we observe that only 133 file types make up 98.4% of the total dataset size and each one of them types take up more than 7 GB. We classify these 133 file types as *commonly used file types* and the remaining ones as *non-commonly used file types* as shown in Figure 13. Our further classification expands on the commonly used file types. We split the commonly used file types into 8 type groups based on their usages or functions as follows: `EOL (executables, object code, and libraries)`, `source code`, `scripts`, `documents`, `archives`, `images`, `databases`, and `others`.

4.2.1 Commonly used file type distribution

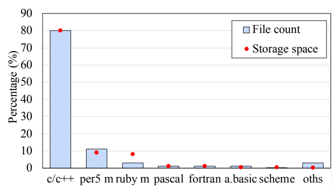
Figure 11(a) shows the total number of files in each type group (shown as bars) and the space occupied by each group (shown as dots). We observe that 44% of files are document files such as HTML, ASCII/UTF files, etc. Note that the type of a file is determined by using the Linux command `file`. ASCII/UTF files are plain text files encoded with UTF or ASCII, which do not match any well-known file types.

As shown in Figure 11(a), 13%, 11%, and 9% of files are source code, EOL, and scripts, respectively. Source code and scripts are encapsulated in Docker images for building dependencies, libraries, and executables (EOL). The EOL files include the dependencies, libraries, and executables that an application needs to run. Only 4% of files are image data files, e.g., PNG, JPEG, etc. Additionally, there are a small amount of video files, such as AVI, MPEG, etc.

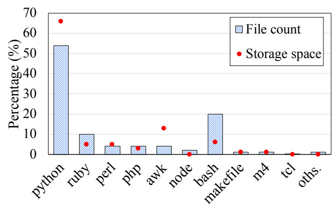
EOL files occupy most of the space, up to 37%, while archives and documents take 23% and 14% of the space, respectively. To find how file types relate to file sizes, we calculate the average file size for each file type group as shown in Figure 11(b). We observe that the average size of a database file is significantly larger (978.8 KB) compared to files within other type groups. The average size of EOL files is only around 100 KB while scripts and source code files are



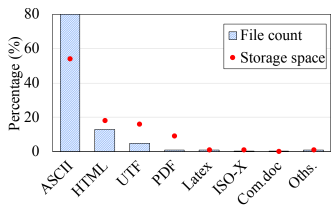
(a) EOL



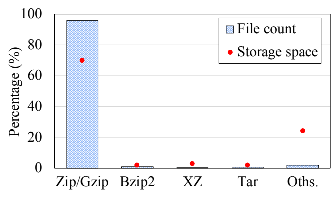
(b) Source code



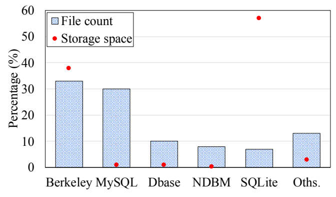
(c) Scripts



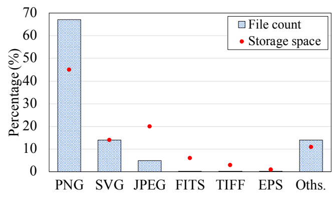
(d) Documents



(e) Archives



(f) Databases



(g) Images

Fig. 12: File count and space occupied by the commonly used file types

only 9 KB on average.

4.2.2 Executables, object code, and libraries (EOL)

EOL files make up the core of a Docker image for the correct execution of an application. The EOL group contains the following types: ELF files, COFF files, intermediate representations that can be executed by a virtual machine, Microsoft executables, Debian/RPM binary packages, libraries, and other EOL files.

Figure 12(a) shows the total number of files for each EOL type. The majority of EOL files are ELF and intermediate representations (IR). ELF files mainly contain ELF relocatables, shared objects, and executables. Intermediate representations mainly contain Python byte-compiled files, compiled java class files, and terminfo compiled files. The majority of intermediate representations are Python byte-compiled files. Although intermediate representations constitute up to 64% of EOL files, 30% of ELF files occupy 84% of the storage space consumed by the EOL type group as shown in Figure 12(a). This is because the average size of ELF files is larger than intermediate representations. The average ELF file size is 312 KB while the average file size for intermediate representations is only 9 KB.

4.2.3 Source code

Figure 12(b) shows 7 commonly used programming languages in our dataset: C/C++, Perl5 module, Ruby module, Pascal, Fortran, Applesoft basic, and Lisp/Scheme. 80.3% of source files are C/C++ sources, which take about 80% of storage space consumed by the source code group. Perl5 module and Ruby module source code represents 11% and 3% of source files, respectively, and make up 9% and 8% of the space occupied by source code files, respectively.

4.2.4 Scripts

Figure 12(c) shows the script file distribution. The script group includes: Python scripts, AWK, Ruby, Perl, PHP, make, M4 macro processor, node, Tcl, Bash/shell, and others. 54% of the scripts are Python scripts, which take 66% of storage space occupied by all scripts. 20% and 10% of scripts are Bash/shell and Ruby scripts, which only occupy 6% and 5% of storage space, respectively.

4.2.5 Documents (Doc.)

Figure 12(d) shows the documents file distribution. The majority of the documents are text files including ASCII text (80%), UTF8/16 text (5%), and ISO-8859 text (0.4%), which take up to 70% of the storage space occupied by all documents. Note that these text files are *raw text files* as we have already filtered the text based well-known file types, such as scripts and source codes. 13% of the documents are XML/HTML/XHTML files, which are related to web service applications. This is expected as current web applications are often implemented as microservices, e.g., Amazon [37] or Netflix [37].

4.2.6 Archives, Databases, and Images

Figure 12(e) shows the distribution of archive files. 96.3% are Zip/gzip files which take up to 70% of storage space consumed by all archive files.

Figure 12(f) shows the commonly used databases in containerized applications. Berkely DB and MySQL files are the most prominent, accounting for 33% and 30% of database files, respectively. In total, Berkely DB and MySQL files take up to 40% of space occupied by database related files. While only 7% of database related files are SQLite DB files, these files occupy over 57% of space among the database files.

The image data file distribution is shown in Figure 12(g). 67% of image files are PNG files, which take about 45% of the space occupied by all image files. We also observe other image data files such as JPEG, SVG, etc., in the images, which are mostly used by XML/HTML/XHTML documents for web applications.

5 Deduplication Analysis

In this section, we investigate the potential for data reduction in the Docker registry.

5.1 Layer sharing

Compared to other existing containerization frameworks [38], [39], Docker supports the sharing of layers among different images to reduce storage utilization for both local images and images in the registry. To study the effectiveness of this approach, we compute how many times each layer is referenced by images.

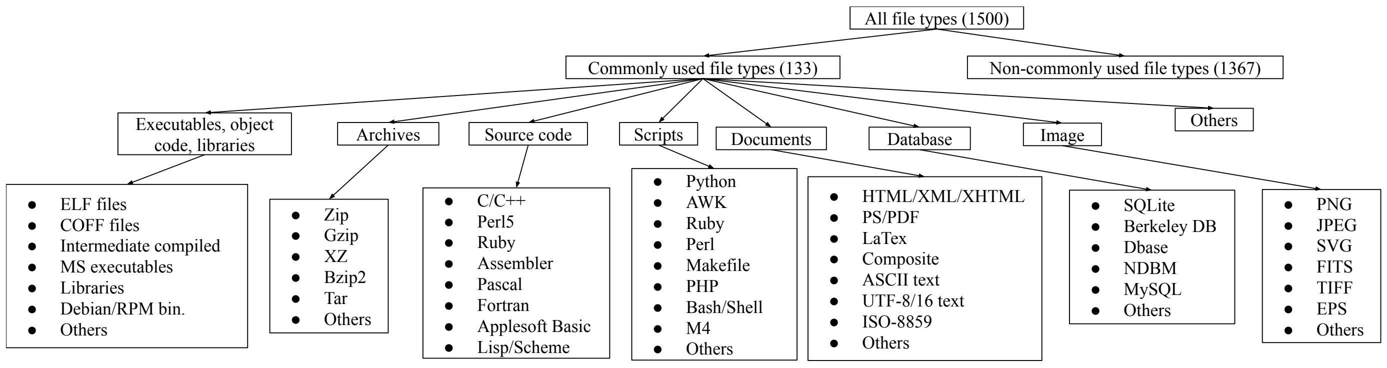


Fig. 13: Taxonomy of file types

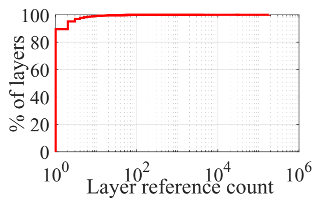


Fig. 14: CDF of layer reference count

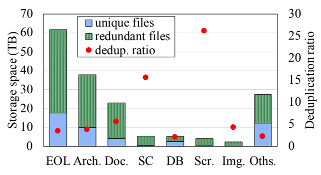


Fig. 17: Overall dedup. ratios

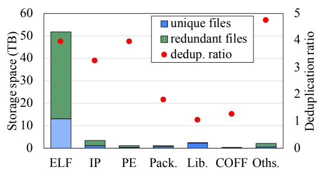


Fig. 18: EOL dedup. ratios

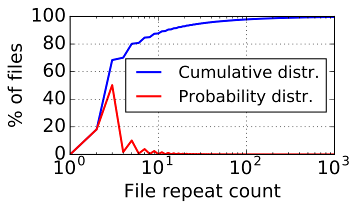


Fig. 15: File repeat count

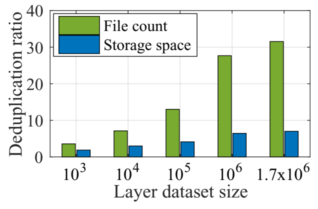


Fig. 16: Deduplication ratios

Specifically, we analyze all image manifests and count for each layer, how many times it is referenced by an image. Figure 14 shows that around 90% of layers are only referenced by a single image, an additional 5% are referenced by 2 images, and less than 1% of the layers are shared by more than 25 images. From the above data we can estimate that without layer sharing, the Docker Hub dataset would grow from 47 TB to 85 TB, implying a **1.8**× redundancy ratio provided by layer sharing.

5.2 File-level deduplication

Next, we calculate the redundancy ratio at file granularity both in terms of file count and space occupied. After removing the redundant files, only 3.2% of the files are left, which in total occupy 24 TB. This results in redundancy ratios of **31.5**× for file count and **6.9**× for space occupied compared to the uncompressed image dataset. Compared to the compressed image dataset, the file-level deduplication ratio is almost 2×.

We further analyze the number of file copies (i.e., duplicates) for every file (see Figure 15). We observe that over 99.4% of files have more than one copy. Around 50% of files have exactly 4 copies and 90% of files have 10 or less copies.

This shows that there is a high file-level redundancy in Docker images which cannot be addressed by the existing layer sharing mechanism. Hence, there is a large potential for file-level deduplication in the Docker registry.

5.3 Deduplication ratio growth

To further study the potential of file-level deduplication, we analyze the redundancy for an increasing number of layers stored in the registry (see Figure 16). We randomly sample 4 layer subsets from the whole dataset. The x-axis values correspond to the number of layers in each sample. We observe that the deduplication ratio increases almost linearly with the layer dataset size. In terms of file count, it increases from **3.6**× to **31.5**× while in terms of space occupancy, it increases from **1.9**× to **6.9**× as the layer dataset grows from 1000 to 1.7 million layers. This confirms the high benefit file-level deduplication can provide for large-scale registry deployments.

5.4 Deduplication analysis by file type

In this section, we present the deduplication ratios for the commonly used file types to understand the distribution of redundant files in terms of their types and quantity. Figure 17 shows the deduplication results for the following type groups: EOL, archives, documents, source code, scripts, images, and databases. Note that the y-axis shows the space occupied by different file type groups (shown as bars) and their corresponding redundant data ratios (shown as dots). For space occupancy, we separately list space occupied by unique files and by redundant files.

The deduplication ratios are between 2 and 26, and most of the type groups have a high deduplication ratio. For example, the deduplication ratio of EOL files, which include executables, object files, and libraries, is 4×. Source code and scripts have the highest deduplication ratio of 16× and 26×. We believe this is because Docker image developers often use version control systems, such as `git`, to fetch similar source code for building similar microservices, and the code might only differ slightly according to when it was retrieved.

As expected, database related files have the lowest deduplication ratio of 2× because they store customized data.

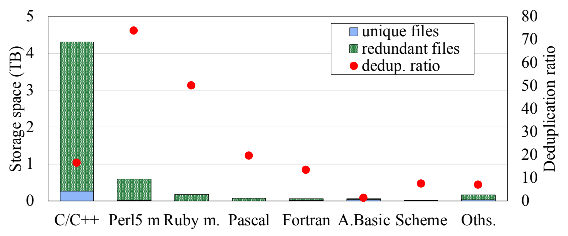


Fig. 19: Source code deduplication ratios

5.4.1 Executable, object code, and libraries (EOL)

We further calculate the deduplication ratios for specific file types in each common type group. We start with the EOL group since it occupies most of the space and significantly contributes to the overall space savings after deduplication.

Figure 18 shows the deduplication results for EOL files. We observe that ELF files, intermediate representations, and PE files have the highest deduplication ratio of 3.3×-4×. Libraries and COFF files have the lowest deduplication ratios: 1.1× and 1.3×, respectively.

5.4.2 Source code

To find out which kind of source code is commonly replicated, we study the deduplication ratios of 7 common languages as shown in Figure 19. We observe that all the languages have a high redundancy ratio of 7.3×- 73× except for A.Basic (1.2×). In particular, C/C++ source files have a deduplication ratio of 16×. To find out why there are so many duplicate C/C++ source files, we inspect those files and find frequently reused source files related to Google Test [40], a cross-platform C++ test framework available on GitHub [40]. We suspect that many developers replicate open source code from external public repositories, such as GitHub [41], and build it inside their container images. This could also explain the significant number of shared source code files across different images. Considering that Docker Hub allows developers to automatically build images from source code in external public repositories and automatically push the built image to their Docker repositories, we believe that replicated source code in different images is a common case in the Docker Hub registry.

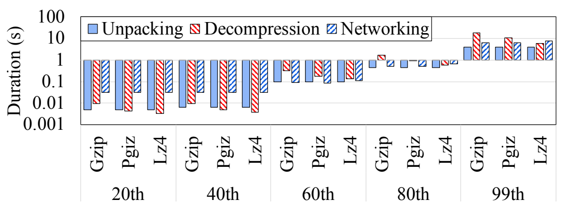
6 Layer Pulling Latency Analysis

Layer `pulling` performance is critical as it largely affects the container *startup time*. The startup time can dominate short-lived jobs/containers especially for the newly emerging serverless computing model [42].

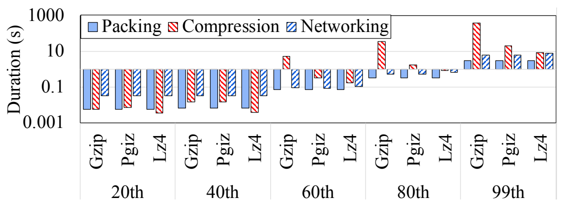
In this section, we measure the layer pulling latency distribution, identify performance bottlenecks, and propose different methods to reduce the latency. In particular, we aim to address the following questions.

1) What is the overall layer pull latency distribution for Docker registries?
2) What is the bottleneck during layer pulling?
3) How do different compression methods impact layer pulling latency?
4) How do different storage options affect layer pulling latency?

Testbed. Our testbed consists of two servers, one running a Docker client and the other running a Docker registry. Each



(a) Average pulling latency breakdown



(b) Average pushing latency breakdown

Fig. 20: Average pulling/pushing latency breakdown. X-axis shows the layer size percentiles. Y-axis is log-scaled.

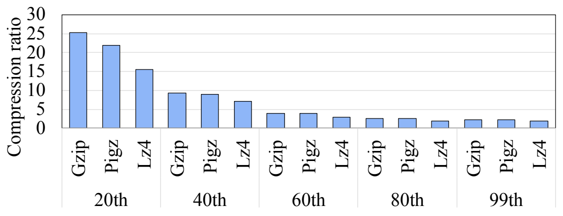


Fig. 21: Average compression ratios

server is equipped with 32 cores, 64 GB RAM, a 500 GB SSD, a 1 TB HDD, and a 10 Gbps NIC.

Layer dataset. To measure the overall layer pulling performance distribution, we first group the layer dataset by layer size. To capture the overall layer pulling performance distribution, we select five groups at the 20^{th}, 40^{th}, 60^{th}, 80^{th}, and 99^{th} percentiles of the layer size distribution (see Table 1). Then, we randomly sample 1000 layers from each group to study pulling performance.

6.1 Performance breakdown

Pulling a layer includes *layer transfer*, *layer decompression*, and *layer unpacking* while layer pushing includes *layer packing*, *layer compression*, and *layer transfer*. Layer compression can significantly reduce layer size and network transfer time. We first study the impact of the following three popular compression methods on layer pulling and pushing latency: (1) `Gzip`, the default compression method for Docker. (2) `Pigz`, a parallel gzip compression method, and (3) `Lz4`, a fast byte-oriented compression scheme. Note that we choose a compression level of 9 for all three methods, which is the high compression mode.

Figure 20(a) shows the average layer pulling latency breakdown for layers at different size percentiles with different compression algorithms. In this experiment, the Docker client stores layers and their uncompressed content on a `tmpfs` disk to reduce packing and unpacking time. As shown in Figure 20(a), the layer pulling latency increases dramatically with layer size. Layers at the 20^{th} (3 KB) and 40^{th} percentile (14 KB) show a similar layer pulling latency of 0.03 s for all three compression algorithms. When the

TABLE 1: Docker image distribution

	20th percentile	40th percentile	60th percentile	80th percentile	99th percentile
File size	1 KB	2 KB	6KB	18 KB	1 MB
File cnt. / layer	1	2	90	2.6K	50K
File cnt. / Img.	15 K	20 K	35K	50K	230K
Layer cnt. / Img.	6	10	12	19	50
Dir depth / layer	1	3	5	8	18
Dir cnt. / layer	3	4	34	400	7.5K
Dir cnt. / Img.	1.8K	3K	4.2K	6.6K	33K
Layer size	3 KB	14 KB	7.8 MB	53.7 MB	878.9 MB
Img. size	190 MB	280 MB	530 MB	800 MB	4.9 GB

layer size increases from 7.8 MB (60th percentile) to 878.9 MB (99th percentile), layer pulling latency increases from 0.36 s to 29 s for Gzip.

Among the three compression algorithms, Gzip has the lowest decompression speed, especially for large layers. For example, it takes 18.7 s for Gzip to decompress a 878.9 MB layer on average. While for Pigz and Lz4, decompression time is reduced to 10.6 s and 6 s, respectively. Although Lz4 is up to 3.1× and 1.8× faster than Gzip and Pigz, respectively, we observe that the network transfer time for Lz4 is slightly higher (1.2×) than both Gzip and Pigz. This is because the average compression ratio for Lz4 is slightly lower (1.2×) compared to Gzip and Pigz as shown in Figure 21. Consequently, there is a trade-off between compression ratio (network transfer time) and compression/decompression speed.

Both Gzip and Pigz show a similar compression ratio because both are based on the DEFLATE algorithm [43] and Pigz only parallelizes Gzip compression to achieve faster compression. We also observe that the compression ratios decrease with layer sizes (see Figure 21). For example, when a layer size increases from 3 KB (20th percentile) to 898.9 MB (99th percentile), the compression ratio decreases from 25.3 to 2.3 for both Gzip and Pigz.

As shown in Figure 20(a), when a tmpfs file system is used for storing unpacked layer contents, the network transfer time is the bottleneck for layer pulling performance for 40% of the layers (layers smaller than 14 KB). For larger layers, decompression time becomes the bottleneck. Although network transfer time also increases with layer size, decompression time increases much faster. This is because, in our setup, the Docker registry and Docker client are located in the same fast LAN. In this case, a fast compression algorithm, such as Lz4 or Pigz, is crucial for speeding up layer pulling.However, if images are accessed from a remote registry over a wide area network, the bottleneck can shift again back to the network.

Figure 20(b) shows the layer pushing latency distribution. We observe that push latencies are much higher than pull latencies. This is because compression is more expensive than decompression. For example, it takes 18 s for Gzip to decompress a 878.9 MB layer while it takes 380.2 s to compress it. Lz4 has the fastest compression compared to the other two algorithms and is 43.2× and 2.3× faster than Gzip and Pigz, respectively, for layers at the 99th percentile. Consequently, a fast compression algorithm, such as Lz4 or Pigz, is important for improving both layer pulling and pushing performance.

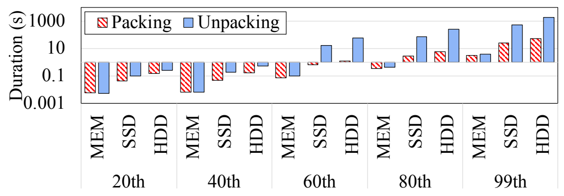


Fig. 22: Storage impact

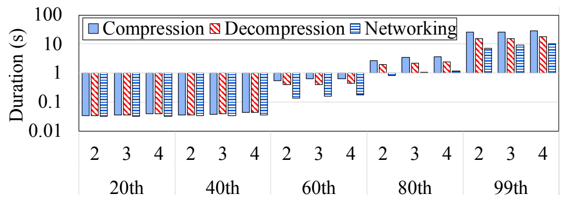
6.2 Storage impact on packing and unpacking

Next, we investigate if packing/unpacking can become a bottleneck for layer pushing/pulling. Figure 22 shows the average packing and unpacking latency distribution by using different storage media to store the uncompressed and unpacked layer content. As expected, unpacking takes much longer than packing because unpacking involves more writes. For example, when an HDD is used to store the unpacked layer content, it takes 53.5 s to pack layers at the 99th percentile while it takes, on average, 1,853.9 s to unpack the layers.

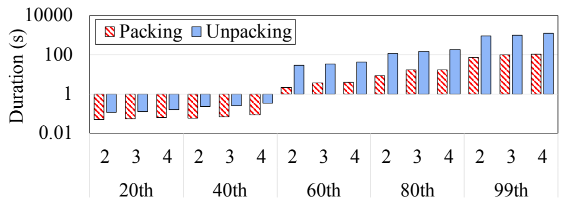
Packing/unpacking time also increases with layer size. For example, when an SSD is used to store the unpacked layer content the unpacking time increases from 0.1 s to 520.2 s as the layer size increases from 3 KB to 878.9 MB Furthermore, using memory to save the unpacked layer content can significantly reduce packing/unpacking time. Packing time decreases by up to 8.6× and 17.9× compared to using a SSD or a HDD, respectively, while for unpacking, the time reduction is up to 129.5× and 461.5×. Therefore, using memory, such as a RAM disk, to temporally host the unpacked layer content and lazily write them back to persistent storage can efficiently reduce packing/unpacking time during layer pushing/pulling. Moreover, the tar archiving process sequentially stores the files in the output tarball. Thus, for larger layers, parallelizing the archiving process can greatly reduce the packing/unpacking overhead.

6.3 Concurrent layer pulling/pushing impact

Figure 23(a) shows the impact of concurrent layer pulling/pushing on compression, decompression, and network transfer. When the concurrency level increases, layer compression time, decompression time, and network transfer time increase slightly. For example, when the number of concurrent layer pulling/pushing threads increases from 2 to 4, the decompression time increases by 1.1–1.3× across different layer sizes and the compression time increases by 1.1–1.4×. The network transfer time also increases by up to 1.4×. Moreover, we observe that decompression and compression take more time than network transfer under



(a) Compression/decompression and network latency



(b) Packing/unpacking latency

Fig. 23: Concurrent layer pulling/pushing impact (x-axis shows the layer size percentile and number of concurrent threads)

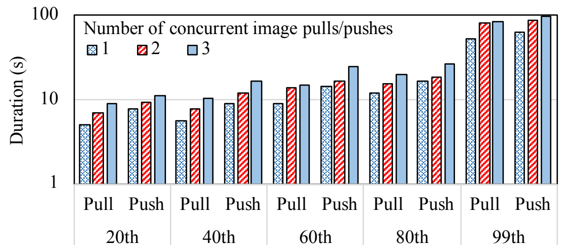


Fig. 24: Image pulling/pushing latency (x-axis shows the image size percentiles)

concurrent layer pulling/pushing threads due to IO contention.

Figure 23(b) shows the concurrent layer pulling/pushing impact on packing and unpacking when an SSD is used to store uncompressed and unpacked layer contents. When the concurrency level increases from 2 to 4, the time to pack and unpack a layer increases by 1.3–2× and 1.3–1.6× respectively. Consequently, using SSDs or HDDs to store unpacked layer contents under concurrent layer pulling/pushing will incur a considerable overhead on packing/unpacking. However, using a RAM disk to host unpacked layer contents under concurrent layer pulling/pushing will surely consume a large amount of memory space. Therefore, using a small memory as a packing and unpacking cache and gradually writing the unpacked layer contents to SSDs/HDDs can significantly reduce packing/unpacking time.

6.4 Pulling and pushing images

Next, we investigate how long it takes to pull or push an image. Pulling an image includes pulling all its containing layers in parallel while pushing an image includes pushing all its containing layers in parallel. In this experiment, we pull/push three layers of an image at a time, as it is Docker's default.

Figure 24 shows the distribution of the average latency to pull and push images of different size percentiles under different image pulling/pushing concurrency. First, pulling is faster than pushing. For example, pushing images at the 60^{th} percentile (530 MB) is 1.6× slower compared to pulling. This is because layer pulling is faster than layer pushing as shown in Section 6.1.

Second, the duration of both pulling and pushing increases with image sizes. For instance, pushing and pulling images at 99^{th} percentile (4.9 GB) is 8× and 10.6× slower compared to pushing and pulling images at 20^{th} percentile (190 MB) as large images usually contain more large layers and layer pulling/pushing latency increases with layer size (see Section 6.1). Third, the duration increases with the number of concurrent images pulled/pushed. For example, when the number of concurrent image pulling/pushing threads increases from 1 to 3, the duration of both image pulling and pushing doubles.

Overall, we find that breaking large images into small, evenly sized layers is beneficial for achieving better pull/push performance as large layers take longer to decompress and unpack and limit pulling parallelism.

7 Container Storage Driver Performance Analysis

As a major component of containers, the I/O performance of container storage drivers is critical to the *image build time*, *container startup time*, and *container execution time*. Therefore, in this section, we evaluate the I/O performance of multiple widely used container storage drivers. We mainly focus on small block sizes for several reasons: First, during image building, commands like `COPY`, `RUN apt install`, or `RUN git clone` can write files into layers inside building containers. Since there are many small-sized files stored in layers and therefore images (i.e., 50% of the files are smaller than 4 KB as detailed in §4 and Table 1), the I/O performance for small I/O requests (i.e., small block sizes) is important for the image building latency.

Second, commands like `RUN make` build (i.e., compiles, archives, or links) and write executables from source code into layers inside building containers. The block sizes involved in executable building usually range from few bytes to 1 MB, e.g., half of I/O requests' block sizes are smaller than 4 KB [44].

We aim to answer the following questions:

1) What is the I/O performance distribution across various block sizes? How does block size impact the read/write performance of containers?
2) What is the overhead caused by container storage drivers? How does the performance of storage drivers compare?
3) What is the impact of concurrency on the I/O performance of container storage drivers?

Testbed. Our testbed is detailed in Section 6. We use SSDs for testing container drivers. To get a stable SSD read/write performance, we first use the `dd` [45] command to write a large amount of data to SSDs with a block size of 1 GB sequentially until the write performance stabilizes at 230 MB/s. Moreover, we use flexible I/O tester `fio` [46] to measure the performance of Docker container storage drivers. To study the raw impact of graph drivers, we clear the page cache [47], delete the data written by the previous test, set the `O_DIRECT` flag, use the asynchronous I/O engine `libaio` for `fio`, and disable write buffers on storage drives before each test.

TABLE 2: Schemes.

Raw file system	Container storage drivers	
	Drivers	**Backends**
Xfs	Overlay2	Xfs
Btrfs	Btrfs	Btrfs
Device Mapper + Xfs	Devicemapper	Device Mapper + Xfs

Drivers. We study three popular container storage drivers: `Overlay2` [19], `Btrfs` [22], and `Devicemapper` [20]. The backend file systems for the above drivers are `Xfs`, `Btrfs`, and `Device Mapper (DM)` thin pools in `direct-lvm mode` with `Xfs`, respectively as shown in Table 2.

To evaluate the overhead caused by the storage drivers, we compare each storage driver with its corresponding backend file system without using a container, denoted as *RAW* in Table 2. Each backend file system and raw file system is created on a separate physical partition on the SSD drive.

7.1 Small I/O requests

To evaluate the I/O performance of different storage drivers for varying block sizes, we launch a `fio` container [46]. We first create 1000 128 KB files and measure the random I/O performance using different block sizes.

Figure 25 shows IOPS, bandwidth, and average I/O completion latency for random reads. We observe that the raw file systems have higher IOPS and bandwidth and lower latencies compared to their corresponding container storage drivers. For example, the latencies of `Overlay2`, `DM`, and `Btrfs` are 1.01×, 1.2×, and 3.7× higher than their corresponding raw file systems, `Xfs-RAW`, `DM-RAW`, and `Btrfs-RAW`, when the block size is 2 KB.

When block sizes are smaller than 4 KB, `Btrfs` exhibits the highest IOPS, bandwidth, and lowest latency for both the storage driver and the raw file system. For example, when the block size is 2 KB, the IOPS for `Btrfs` and `Btrfs-RAW` are 23.1 K and 25.4 K, respectively, as shown in Figure 25(a). The bandwidth of `Btrfs` is 1.5× and 1.9× higher than `Overlay2` and `DM`, respectively (see Figure 25(b)) while latencies are 20.8× and 25× lower than the latencies of `Overlay2` and `DM` (see Figure 25(c)). For `Btrfs-RAW`, the bandwidth is 1.5× and 1.6× higher than `Xfs-RAW` and `DM-RAW` and its latencies are 76.6× and 79.2× lower than `Xfs-RAW` and `DM-RAW` (see Figure 25(c)). However, when the block size increases, both `Overlay2` and `Xfs-RAW` show slightly higher IOPS and bandwidth and lower latencies compared to the other four configurations as shown in Figure 25.

Overall, Figure 25 indicates that during image building or container execution, if the I/O requests' block sizes are smaller than 4 KB, `Btrfs` has the best small file read performance compared to `Overlay2` and `DM`. Otherwise, `Overlay2` can provide a better read performance for containerized applications.

As for write performance, `Btrfs` performs slightly better than `overlay2` and `DM` when the block size is smaller than 4 KB as shown in Figure 26. For example, the bandwidth of `Btrfs` is 1.1× and 1.3× higher than the bandwidth of `Overlay2` and `DM`, respectively. When the block size increases, `Overlay2` outperforms both `Btrfs` and `DM`.

7.2 CoW performance

To evaluate the overhead caused by the CoW mechanism of container storage drivers, we launch a `fio` container, create a new layer by randomly creating and writing 1000 128 KB files, and measure random write performance, denoted as *creation performance*. After that, we `commit` the container as a new image and run the newly created image as a container instance. We then randomly rewrite 6 KB of data in each file in the preceding read-only layer, and measure the rewrite performance.

We observe that when the block size is larger than 4 KB container storage drivers and their corresponding raw file systems exhibit similar performance. Therefore, in this experiment, we set a smaller block size for I/O requests to 2 KB.

Figure 27 shows the IOPS and bandwidth for both layer file creation write requests and rewrite requests to the read-only layer files. Overall, the creation write performance is slightly higher than the rewrite performance for all the container drivers. For example, IOPS and bandwidth degrade 4%-23%, for rewrites compared to creation writes. `Btrfs` degrades the most for rewrites compared to `Overlay2` and `DM`, though the performance of `Btrfs` is best in both creation and rewriting. For raw file systems the rewrite performance is slightly higher than creation performance, with the exception of `Btrfs-RAW` as shown in Figure 27.

Moreover, we observe that raw file systems outperform container storage drivers. For example, with the `Btrfs` driver, the IOPS degrade by 11.3% compared to `Btrfs-RAW` for creation. This indicates that container storage drivers incur rewrite overhead for smaller block sizes.

7.3 Concurrency impact

To study the concurrency impact on the performance of container storage drivers, we increase the number of read/write threads for `fio` and show their I/O latency in Figure 28. Note that the block size is set to 2 KB, as in §7.2.

Read latency increases with concurrency as shown in Figure 28. The I/O latency increases 1.02–1.5× as the number of read threads increases from 2 to 5. We observe that `Xfs-RAW` and `Overlay2` are more sensitive to concurrency. They increase by up to 1.5× and 1.14× as concurrency increases from 2 to 5. The other four schemes only increase by 1.02-1.06× and remain almost stable with increasing read threads.

The write completion latency is more stable with concurrency. The I/O latency only increases 1.04–1.2× as the number of write threads increases from 2 to 5. For example, the I/O latency of `DM-RAW` increases by up to 1.2× while for the remaining configurations, it increases by less than 1.09×.

8 Conclusion

We carried out the first comprehensive analysis of container images stored in Docker Hub. Our findings reveal that there is room for optimizing how images are stored and used. We summarize the dataset (§4) and deduplication (§5) analysis in the following key observations: **(1) Images and layers contain many files, many of which are small.** For example, half of layers and images have more than 30 and 1,090 files, respectively. 50% and 90% of files are

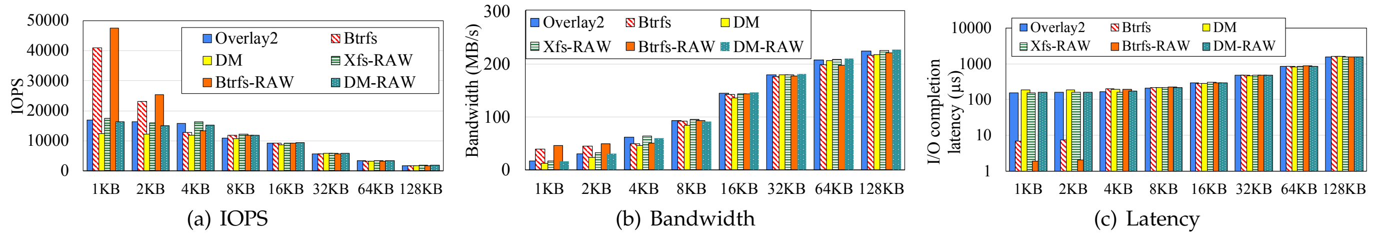


Fig. 25: I/O size impact on reads. X-axes represents I/O size

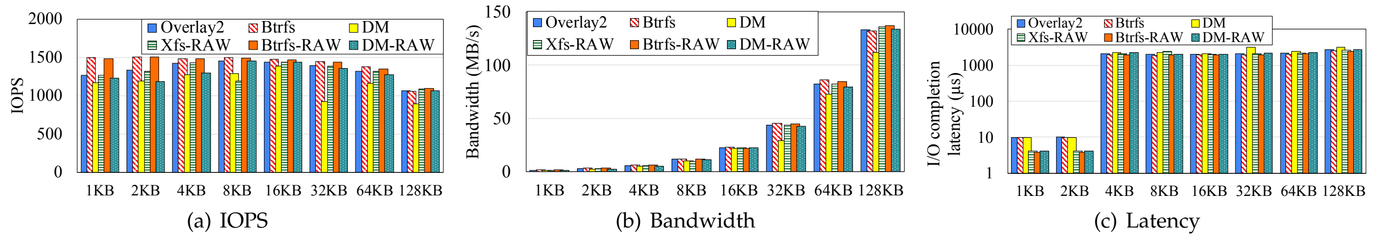


Fig. 26: I/O size impact on writes. X-axes represents I/O size

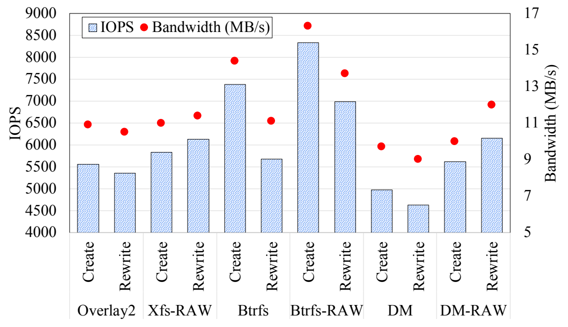


Fig. 27: CoW performance of container storage drivers and backend file systems

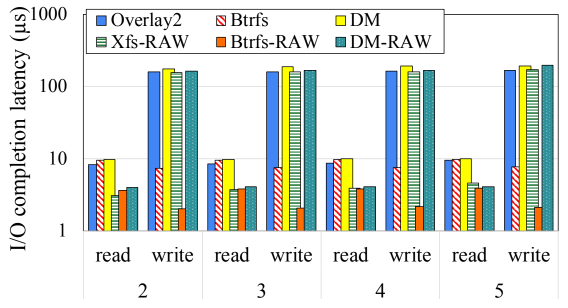


Fig. 28: I/O latency versus the number of read/write threads

smaller than 4 KB and 26 KB (§4). This requires the underlying container storage drivers to efficiently manage small files. **(2) There is significant potential for deduplication on large-scale Docker registries** (as shown in our recent paper on a deduplication system for Docker registries [48]). Only 3% of files in the uncompressed image dataset are unique, resulting in a deduplication ratio of 2×. (§5). **(3) A large potion of redundant files is introduced into images by different users using slightly different source code versions or package managers, such as `apt` or `pip`, to build or install similar software (§5).** For example, we observed that the deduplication ratio for source code files in the uncompressed image dataset can be as high as 73× (see §5.4.2). Hence, we believe our deduplication analysis is representative, even for more recent images, which may utilize slimmer base images like `alpine` Linux to reduce the size of a single image, as redundancy is often caused by user-specific data.

Based on our image characterizations, we further investigated the I/O performance of image retrievals (§6) and storage drivers (§7) and drew several design implications. **(1) parallel compression algorithms such as `Pigz` and `Lz4` can significantly reduce compression/decompression time.** Otherwise, compression/decompression can become a bottleneck for layer `pulling/pushing` (§6). **(2) Using memory as a cache to temporally store unpacked layer content can significantly reduce the latency**. Otherwise, packing/unpacking on HDD can become a bottleneck for layer `pulling/pushing` (§6). **(3) Container storage drivers add an additional overhead to their corresponding backend file systems, especially when block sizes are small.** The performance of container storage drivers is lower compared to their corresponding raw backend file system (without using containers) (§7). **(4) Rewrites to files in preceding read-only layers is slower than writing new files, for all container storage drivers evaluated (i.e., `Overlay2`, `Btrfs`, and `DM`) (§7).**

As future work, we plan to extend our performance analysis of containers to different kinds of workloads, such as HPC applications, and extend our image analysis to more recent datasets from different Docker registries, such as JFrog Artifactory [49].

References

[1] P. Menage, "Adding Generic Process Containers to the Linux Kernel," in *Linux Symposium*, 2007.

[2] 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 *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 1–10, 2019.

[3] M. Rosenblum and T. Garfinkel, "Virtual Machine Monitors: Current Technology and Future Trends," *Computer*, vol. 38, no. 5, 2005.

[4] J. Cito, V. Ferme, and H. C. Gall, "Using docker containers to improve reproducibility in software and web engineering research," in *Web Engineering*, Springer International Publishing, 2016.

[5] K. Matthias and S. P. Kane, *Docker: Up & Running Shipping Reliable Containers in Production*. 2018.

[6] "Namespaces(7) ← linux programmer's manual." http://man7.org/linux/man-pages/man7/namespaces.7.html.

[7] "Control Group v2." https://www.kernel.org/doc/Documentation/cgroup-v2.txt.

[8] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum, "Disco: Running Commodity Operating Systems on Scalable Multiprocessors," *TOCS*, vol. 15, no. 4, 1997.

[9] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An Updated Performance Comparison of Virtual Machines and Linux Containers," in *ISPASS*, 2015.

[10] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. Lightweight Virtualization: A Performance Comparison," in *IC2E*, 2015.

[11] "Docker." https://www.docker.com/.

[12] "Docker Hub." https://hub.docker.com/.

[13] J. Bhimani, J. Yang, Z. Yang, N. Mi, Q. Xu, M. Awasthi, R. Pandurangan, and V. Balakrishnan, "Understanding performance of I/O intensive containerized applications for NVMe SSDs," in *IPCCC*, 2016.

[14] A. Brogi, D. Neri, and J. Soldani, "DockerFinder: Multi-attribute Search of Docker Images," in *IC2E*, 2017.

[15] J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, and H. C. Gall, "An Empirical Analysis of the Docker Container Ecosystem on GitHub," in *MSR*, 2017.

[16] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast Distribution with Lazy Docker Containers," in *FAST*, 2016.

[17] R. Shu, X. Gu, and W. Enck, "A Study of Security Vulnerabilities on Docker Hub," in *CODASPY*, 2017.

[18] N. Zhao, V. Tarasov, A. Anwar, L. Rupprecht, D. Skourtis, A. S. Warke, M. Mohamed, and A. R. Butt, "Slimmer: Weight loss secrets for docker registries," in *IEEE Cloud*, 2019.

[19] "Use the overlayfs storage driver-how the overlay2 driver works." https://docs.docker.com/storage/storagedriver/overlayfs-driver/#how-the-overlay2-driver-works.

[20] "Use the device mapper storage driver." https://docs.docker.com/storage/storagedriver/device-mapper-driver/.

[21] "Use the zfs storage driver." https://docs.docker.com/storage/storagedriver/zfs-driver/.

[22] "Use the btrfs storage driver." https://docs.docker.com/storage/storagedriver/btrfs-driver/.

[23] V. Tarasov, L. Rupprecht, D. Skourtis, A. Warke, D. Hildebrand, M. Mohamed, N. Mandagere, W. Li, R. Rangaswami, and M. Zhao, "In Search of the Ideal Storage Configuration for Docker Containers," in *AMLCS*, 2017.

[24] F. Guo, Y. Li, M. Lv, Y. Xu, and J. C. S. Lui, "Hp-mapper: A high performance storage driver for docker containers," in *SoCC*, 2019.

[25] S. P. R. Dua, V. Kohli and S. Patil, "Performance analysis of union and cow file systems with docker," pp. 550–555, IEEE Press, 2016.

[26] D. Skourtis, L. Rupprecht, V. Tarasov, and N. Megiddo, "Carving perfect layers out of docker images," in *HotCloud*, 2019.

[27] K.-T. Seo, H.-S. Hwang, I.-Y. Moon, O.-Y. Kwon, and B.-J. Kim, "Performance comparison analysis of linux container and virtual machine for building cloud," pp. 105–111, 12 2014.

[28] M. J. Scheepers, "Virtualization and containerization of application infrastructure : A comparison," 2014.

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

[30] M. G. Xavier, I. C. D. Oliveira, F. D. Rossi, R. D. D. Passos, K. J. Matteussi, and C. A. F. D. Rose, "A performance isolation analysis of disk-intensive workloads on container-based clouds," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 253–260, 2015.

[31] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *OSDI*, 2018.

[32] T. Harter, *Emergent Properties in Modular Storage: a Study of Apple Desktop Applications, Facebook Messages, and Docker Containers*. PhD thesis, Madison, WI, USA, 2016.

[33] "Improving copy-on-write performance in container storage drivers." https://www.snia.org/sites/default/files/SDC/2016/presentations/capacity_optimization/FrankZaho_Improving_COW_Performance_ContainerStorage_Drivers-Final-2.pdf.

[34] S. Talluri, A. undefineduszczak, C. L. Abad, and A. Iosup, "Characterization of a big data storage workload in the cloud," in *ICPE*, 2019.

[35] Q. Xu, M. Awasthi, K. T. Malladi, J. Bhimani, J. Yang, and M. Annavaram, "Performance analysis of containerized applications on local and remote storage," in *MSST*, 2017.

[36] V. Tarasov, L. Rupprecht, D. Skourtis, A. Warke, D. Hildebrand, M. Mohamed, N. Mandagere, W. Li, R. Rangaswami, and M. Zhao, "In search of the ideal storage configuration for docker containers," in *FAS*W*, 2017.

[37] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, and et al., "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *ASPLOS*, 2019.

[38] "OpenVZ Linux Containers Wiki." http://openvz.org/.

[39] "singularity." http://singularity.lbl.gov/.

[40] "Google test - google testing and mocking framework." https://github.com/google/googletest.

[41] "GitHub." https://github.com/.

[42] X. Yi, F. Liu, D. Niu, H. Jin, and J. C. Lui, "Cocoa: Dynamic container-based group buying strategies for cloud computing," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, vol. 2, pp. 1–31, 02 2017.

[43] S. Oswal, A. Singh, and K. Kumari, "DEFLATE COMPRESSION ALGORITHM," *International Journal of Engineering Research and General Science*, vol. 4, no. 1, 2016.

[44] A. Klimovic, Y. Wang, C. Kozyrakis, P. Stuedi, J. Pfefferle, and A. Trivedi, "Understanding ephemeral storage for serverless analytics," in *ATC*, 2018.

[45] "dd." https://linux.die.net/man/1/dd.

[46] "fio(1) - linux man page." https://linux.die.net/man/1/fio.

[47] D. P. Bovet and M. Cesati., *Understanding the Linux Kernel.* 2018.

[48] 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 *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pp. 769–783, 2020.

[49] J. Artifactory. https://jfrog.com/.

Nannan Zhao is a Ph.D. candidate in the Department of Computer Science at Virginia Tech. Her research areas include distributed storage systems, key-value stores, flash memory, and Docker container.

Vasily Tarasov is a data storage researcher at IBM Almaden Research Center. His research areas include performance analysis of file systems and I/O stack design.

Hadeel Albahar is a Ph.D. candidate in the Department of Electrical and Computer Engineering at Virginia Tech. Her research interests include distributed storage systems, cloud computing, containerization, and serverless computing.

Ali Anwar is a distributed systems researcher at IBM Almaden Research Center. His research areas include distributed machine/federated learning and serverless.

Lukas Rupprecht is a data storage researcher at IBM Almaden Research Center. His research areas are related to scalability and performance of distributed systems.

Dimitrios Skourtis is a data storage researcher at IBM Almaden Research Center. His research areas are related to resource management and scheduling.

Arnab K. Paul is a Ph.D. candidate in the Department of Computer Science at Virginia Tech. His research areas include parallel I/O systems, storage systems, high-performance and distributed computing.

Keren Chen is an undergraduate in the Department of Computer Science at Virginia Tech. His research areas include distributed storage systems and Docker container.

Ali R. Butt is a professor in the Department of Computer Science at Virginia Tech. His research areas include distributed systems and parallel computing.