Slimmer: Weight Loss Secrets for Docker Registries

***Abstract*—Due to their tight isolation, low overhead, and efficient packaging of the execution environment, Docker containers have become a prominent solution for deploying modern applications. Containers are created from *images* which are stored in a Docker *registry*. An image consists of a list of *layers* which can be shared among images. Docker registries store a large amount of images and with the increasing popularity of Docker, they continue to grow. For example, Docker Hub—a popular public registry—stores more than half a million public images. In this paper, we analyze over 167 TB of uncompressed Docker images and evaluate the potential of file-level deduplication in the registry. Our analysis reveals that only 3% of the files in images are unique and Docker's existing layer sharing mechanism is not sufficient to eliminate this profound redundancy. We then present the design of Slimmer—a Docker registry with file deduplication support—and conduct a simulation-based analysis of its performance implications.**

***Keywords*-Docker; Deduplication; Docker registry; Distributed storage systems;**

I. Introduction

Containers [1] have recently gained significant traction due to their low overhead, fast deployment, and the rise of container management frameworks such as Docker [2]. Polls suggest that 87% of enterprises are at various stages of adopting containers, and they are expected to constitute a $2.7 billion market by 2020 [3].

Docker combines process containerization with efficient and effective packaging of complete runtime environments in so called *images*. Images are composed of shareable and content addressable *layers*. A layer is a set of files, which are compressed in a single archive. Both images and layers are stored in a Docker *registry* and accessed by clients as needed. Since layers are uniquely identified by a collision-resistant hash of their content, no duplicate layers are stored in the registry.

Registries are growing rapidly. For example, Docker Hub [4], the most widely used registry, stores more than 500,000 public image repositories comprising over 2 million layers and it keeps growing. Over a period from June to September 2017, we observed a linear growth of the number of images in Docker Hub with an average creation rate of 1,241 repositories per day. We expect this trend to continue as containers gain more popularity. This massive image dataset presents challenges to the registry storage infrastructure and so far has remained largely unexplored.

In this paper, we perform the first large-scale redundancy analysis of the images and layers stored in Docker Hub. We downloaded 47 TB (167 TB uncompressed) worth of Docker Hub images, which in total contain over 5 billion files. Surprisingly, we found that only around 3% of the files are unique while others are redundant copies. This suggests that current layer sharing cannot efficiently remove data duplicates. We further analyzed the reasons for the high number of redundant files and found, for example, that different Docker images often contain the same source code from external public repositories (e.g., GitHub [5]). Container images are similar with virtual machine images in the sense that they all serve as OS snapshots. Difference users might choose similar OS and run similar applications, which incurs a considerable redundancy across different container images.

Deduplication technique is a good solution to remove the redundancy from container image storage system. However, current deduplication techniques cannot be directly applied on container image storage system because container images are stored as a number of gzip compressed tar files, known as layers. Gzip compressed files have very low deduplication ratio.

Given our findings, we propose Slimmer, a file-level content addressable storage model for the Docker registry. Slimmer unpacks layer tarballs into individual files and deduplicates them. When a Docker client requests a layer, Slimmer dynamically reconstructs the layer from its constituent files. To assess the feasibility of our design, we conduct a simulation-based evaluation of Slimmer. The simulation results show that Slimmer improves the deduplication ratio from 1.8×, provided by layer sharing, to 6.9×. While Slimmer comes with some overhead caused by the need to decompress and reconstruct layers, we found, for example, that for layers less than 10 MB (around 60% of all layers) the overhead of retrieving a layer is less than 1 s. For larger layers, we propose several optimizations to reduce overhead.

II. Background

Docker [2] is a popular virtualization technology that extends traditional OS containers with higher level functionalities. It allows to efficiently package an application and its runtime dependencies in a container image to simplify and automate application deployment [6]. As shown in Figure 1, a typical Docker setup consists of three main components: *client*, *host*, and *registry*.

Users interact with Docker using the Docker client which, in turn, sends commands to the Docker host. The client can be co-located on the host machine. The Docker host runs a daemon process that implements the core logic

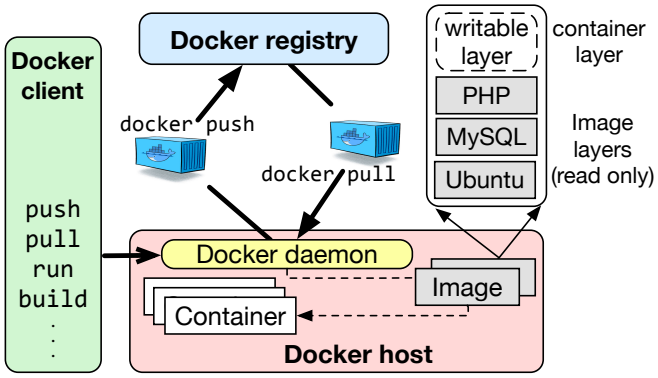


Figure 1. Docker ecosystem.

of Docker and is responsible for *running* containers from locally available images. If a user tries to launch a container from an image that is not available locally, the daemon *pull*s the required image from the Docker registry. Additionally, the daemon supports *building* new images and *pushing* them to the registry.

Images and layers: At the center of Docker is the concept of container images for packaging, distributing, and running applications. A Docker image consists of an ordered series of *layers*. Each Docker layer contains a subset of the files in the image and often represents a specific component/dependency of the image, e.g., a shared library. Layers can be shared between two or more images if the images depend on the same layer. Image layers are read-only. When users start a container, Docker creates a new *writable layer* on top of the underlying read-only layers (Figure 1). Any changes made to files in the image will be reflected inside the writable layer via a copy-on-write mechanism.

Registry: The Docker registry is a platform for storing and distributing container images. It stores images in *repositories*, each containing different versions (*tags*) of the same image, identified as `<repo-name:tag>`. For each image, the Docker registry stores a *manifest* that describes, among other things, which layers constitute the image. Layers are identified via a digest that is computed as a hash (SHA-256) over the uncompressed content of the layer and stored as compressed archival files. Identifying layers by their content allows the registry to store only one instance of a layer even if it is referenced by multiple images. Docker Hub is one of the most popular public registries, supporting both public and private repositories, via which users can upload, search, and download images [4].

III. Deduplication Analysis

In this section, we investigate the potential for data reduction in the Docker registry by estimating the efficacy of layer sharing, compression, and the proposed file-level deduplication. We also analyze the root causes of the high file redundancy across container images.

A. Methodology

To analyze the benefits of different data reduction techniques for the Docker registry, we downloaded a large number of Docker images. We chose the Docker Hub registry [4] as our source of images due to its popularity and its significant number of public repositories. We expect that because of these reasons, our findings are applicable to other registry deployments.

Docker Hub does not provide an API to retrieve all repository names. Hence, we crawled the registry's website to obtain a list of all available repositories, and then downloaded the *latest* version of an image and its corresponding layers for each repository. We plan to extend our analysis to other image tags in the future. 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. Table I summarizes the properties of the downloaded dataset. Specifically, there are two steps: i) crawl Docker Hub web pages to list all repositories; ii) download the *latest* image (and referenced layers) from each repository based on the crawler results. Here, we chose latest image from each repository because (1) the "latest" version is usually the newest, stable, and commonly pulled by developers; (2) downloading only the "latest" version can shorten our downloading process since the latest images already took about 30 days to download. To store and analyze the data, we deployed an 8-node Spark [7] cluster with HDFS [8] as the backend storage.

B. Data reduction analysis

Layer sharing: Compared to other existing containerization frameworks [9] [10], Docker supports the sharing of layers among different images. To analyze the effectiveness of this approach, we compute how many times each layer is referenced by images.

Figure 2 shows that around 90% of layers are 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. Interestingly, there is one layer that is referenced by 184,171 images. Further analysis reveals that this is an empty layer. The presence of an empty layer in an image can be explained by the fact that during the image build, Docker creates a new layer for every `RUN <cmd>` instruction in the Dockerfile [11]. If the `<cmd>`, which can be an arbitrary shell command, does not modify any files in the file system, an empty layer is created.

The next 5 top-ranked layers by reference count are included in 29,200 – 33,413 images. Specifically, one layer contains a whole `Ubuntu 14.04.2 LTS` distribution, one layer contains a `sources.list` file for `apt`, and one layer contains binaries and libraries needed for `dpkg`. The other two layers are related to cowsay, a program that can generate ASCII pictures of a cow with a message [12]. One layer contains a whole installation package for `cowsay`

Table I
DATASET SUMMARY

# of repos crawled	# of unique repos	# of images downloaded	# of layers downloaded
634,412	457,627	355,319	1,792,609
# of images analyzed	**# of layers analyzed**	**Compressed dataset size**	**Total # of files**
355,319	1,792,609	47TB	1,792,609

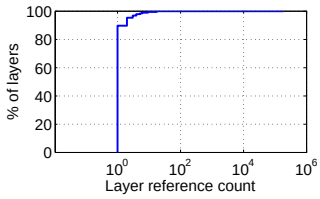


Figure 2. CDF of layer reference count.

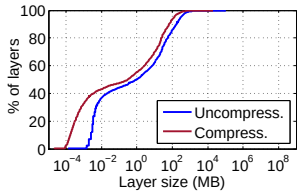


Figure 3. CDF of compress. and uncompress. layer size.

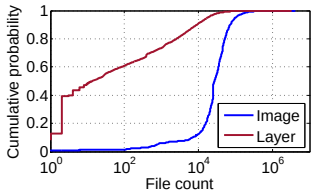


Figure 4. CDF of file count.

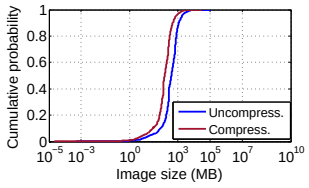


Figure 5. CDF of image size (MB)

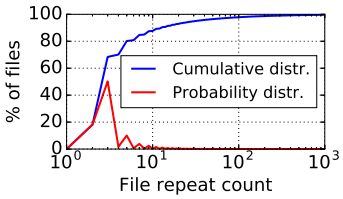


Figure 6. File repeat count distribution.

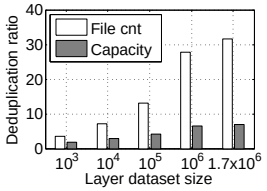


Figure 7. Deduplication ratio growth.

`3.03` while the other layer only contains the binaries for cowsay.

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**× deduplication ratio provided by layer sharing.

Compression: Figure 3 presents compressed and uncompressed layer size distributions. We found that 50% of the layers are smaller than 1 MB and 90% of the layers are smaller than 64 MB in compressed format. If uncompressed, 50% of the layers are smaller than 2 MB and 90% of the layers are smaller than 170 MB. Moreover, the total compressed layer dataset grows from 47 TB to 167 TB after decompression, resulting in a compression ratio of **3.6**×. Compression is complementary to layer sharing and therefore, the current registry reduces the dataset size by a factor of **3.6**×**1.8 = 6.5**.

Figure 5 show the image size distributions. We find that 80% of the images have an uncompressed size less than 794 MB and compressed size of 312 MB. In the median, this decreases to 406 MB and 157 MB, respectively. The largest uncompressed image is 498 GB which is a Ubuntu-based image. As shown in Figure 4, 50% of images have less than 27,194 files and 90% of images have less than 74,266 files.

File-level deduplication: Next, we calculate the deduplication ratio in terms of file count and capacity for the complete dataset. After removing redundant files, there are only 3.2% of files left that in total occupy 24 TB, resulting in deduplication ratios of **31.5**× and **6.9**× in terms of file count and capacity, respectively. With compression applied, file-level deduplication can hence reduce storage utilization by a factor of **6.9**×**3.6 = 24.8**.

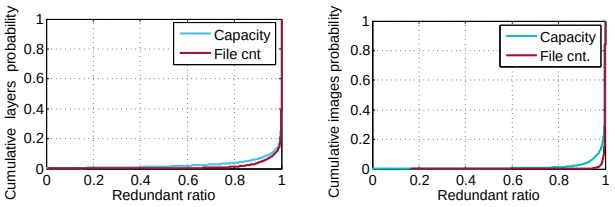
We further analyze the repeat count for every file (see Figure 6). We observe that over 99.4% of files have more than one copy. Around 50% of files have exactly 4 copies and 90% of files have 10 or less copies. The file that has the maximum repeat count of 53,654,306 is an empty file. Around 4% of empty files are `__init__.py`, which make Python treat a directory as containing packages and are usually empty. Other frequent empty files include `lock` or `.gitkeep` files.

We also analyze 5 frequently repeated files which repeat 3,338,145 – 11,847,356 times. Specifically, two files `libkrb5-3:amd64.postrm` and `libkrb5-3:amd64.postinst` are two Kerberos runtime libraries for `dpkg`. Another two files are related to the `npm` package manager (`license` and `.npmignore`) and the last file, `dependency_links.txt`, contains a list of dependency URLs for Python.

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 deduplication in the Docker registry.

Deduplication ratio growth: To further study the potential of file-level deduplication, we analyze the deduplication for an increasing number of files stored in the registry (see Figure 7). The x-axis values correspond to the sizes of 4 random samples drawn from the whole dataset and the size of the whole dataset.

We see 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 capacity, it increases from **1.9**× to **6.9**× as the layer dataset grows from 1000 to 1.7 million layers. This confirms the high



(a) CDF of cross-layer shared file ratio.

(b) CDF of cross-image shared file ratio.

Figure 8. Cross layer sharing and cross image sharing

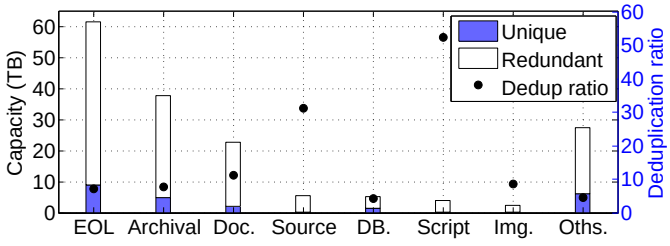


Figure 9. Deduplication ratio for seven file classes—EOL, archival, documents, source code, database, scripts, images—and other files. Light bars indicate the original storage utilization while dark bars represent the capacity after removing redundant files. Dots shows the corresponding deduplication ratio.

potential for file-level deduplication in large-scale Docker registry deployments.

Cross layer sharing: Based on the high deduplication ratio, we guess that a large amount of files are shared between layers and the large potential from deduplication is due to files sharing between layers. Cross layer shared files are files that stored in more than one layers. There could be a high probability for Docker registry. For example, different developers work on same source codes and build same executables in their layers.

Figure 8(a) shows percentage of cross layer shared files for each layer. We find that 90% of layers contain more than 97.6% of files that are shared across layers. We also calculate the percentage of files that are shared across images. As shown in Figure 8(a), 90% of images contain more than 99.4% of files that are shared across images, indicating that majority of files are replicated across different images and layers.

C. Deduplication by file types

To understand the sources of high data redundancy among Docker images, we investigate deduplication for common file types. We identify 133 file types (e.g., JPG, C/C++, and Java files) using the file type library [13] and then group file types into 7 classes by their high-level use cases: 1) executable, object, and library files (EOL) (such as .o or .pyc) 2) archival (such as .gz or .tar), 3) documents (such as .txt or .tex), 4) source code (such as .c or .java), 5) scripts (such as .py or .js), 6) images (such as .png or .eps), and 7) database files (such as .sqlite or .frm). These classes cover about 88% of the whole dataset (by capacity); we group the remaining 12% in the Others class.

Figure 9 illustrates the deduplication results for these classes. The overall deduplication ratio is **6.9**× and 4 classes have a comparable ratio (indicated by the dots). For example, the deduplication ratio for EOL files is **7.1**×. However, for the other 3 classes, the deduplication ratio is significantly higher—**31.25**× for source code, **50**× for scripts, and **12.5**× for documents. This indicates that users frequently replicate source code, scripts, and documents in their images.

We found that redundant C/C++ source code takes up over 77% of the capacity occupied by source code files. Looking closer we found that, for example, Google Test [14]—a cross-platform C++ test framework available on GitHub [5]—is frequently replicated. Interestingly, we found there are multiple Docker repositories related to Google Test but there is no single *official* repository. We think that many developers replicate open source code from external public repositories, such as GitHub, and store it in container images but do not specifically encapsulate it in a separate layer. Source code also frequently changes so if different versions are put into a layer in different images, a large file base may overlap while few files are different. This can also result in different layers with high redundancy. In addition to Google Test, we also found many replicas of the source code of go-ethereum [15], Android Native Development Kit (NDK) [16], xnu-chroot-environment [17] among others.

Next, we observe that redundant EOL and archival files occupy over half of the total dataset capacity (51.4%). We analyze the top-5 most frequent archival files. We found that `NEWS.Debian.gz`, which stores news about package changes, has 521,611 copies and `pwunconv.8.gz`, which is used to create passwords has 358,374 copies. There are two files that have around 87,000 copies: `ubuntu_dists_trusty_universe_Sources.gz` and `ubuntu_dists_trusty_universe_binary-amd64_Packages.gz`, which contain metadata of the installed packages for the distributions. The last file, `gcc.log.xz` has 13,384 copies and belongs to the GNU C++ compiler.

IV. Registry with deduplication support

As the number of images stored in Docker registry is increasing dramatically, storing and managing the large proportion of redundant files requires both storage equipment and administration investment. In this section, as an alternative to layer-level content addressable storage, a file-level content addressable storage model (Slimmer) is suggested for Docker registry to remove redundant files and save space while maintaing good performance.

In this section, we first describe a high-level design of *Slimmer*— a Docker registry that supports file-level deduplication. We then proceed with a simulation-based evaluation

of the expected performance implications. We also provide suggestions on how one can efficiently server push/pull requests.

A. Design

We designed Slimmer so that the interface between the Docker clients and the registry remains unchanged. As such, no modifications to the Docker clients are needed. Below we describe how Slimmer handles layer pushes and pulls at the registry side.

Push: Slimmer handles push requests asynchronously. After receiving a layer from a client, it does not immediately unpack the layer. Instead, it reliably stores the layer's compressed tarball in a persistent *staging area*. A separate *off-line* deduplication process iterates over the layers in the staging area and performs the following steps for every layer:

1) decompress and unpack the layer's tarball into individual files;
2) compute a *fingerprint* for every file in the layer;
3) check all file fingerprints against the *file index* to identify if identical files are already stored in Slimmer;
4) store non-deduplicated files in Slimmer's storage;
5) create and store a *layer recipe* that includes the path, metadata, and fingerprint of every file in the layer;
6) remove the layer's tarball from the staging area.

Layer recipes are identified by layer digests (see Section II) and files are identified by their fingerprints. These identifiers are used to address corresponding objects in the underlying storage. For example, if a file system is used as a backend storage, Slimmer creates a single file for every layer recipe (named by the digest) and a single file for every in-layer file (named by the fingerprint).

Pull: When a layer is pulled, Slimmer has to *reconstruct* the layer based on the layer recipe. A pull request cannot be postponed to an off-line process as the pulling client is actively waiting for the layer. Slimmer performs the following steps *inline* during the pull request:

1) check if the requested layer is still in the staging area and if so, service it directly from there;
2) otherwise, find the layer recipe by the layer digest provided by the client;
3) prepare a directory structure for the layer based on the layer recipe;
4) pack and compress the layer's directory tree into a temporary tarball;
5) send the layer tarball back to the client and then discard the layer tarball.

B. Performance evaluation

While Slimmer can effectively eliminate redundant files in the Docker registry, it introduces overhead which can reduce the registry's performance.

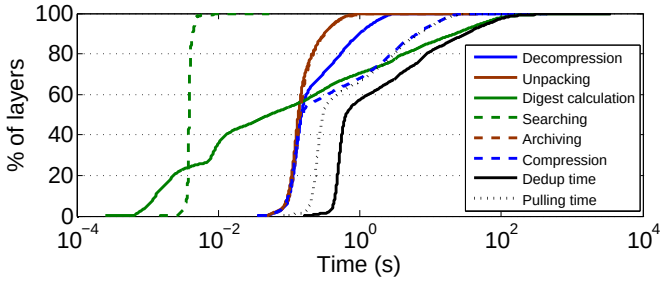


Figure 10. Off-line file-level deduplication run time.

Simulation: To analyze the impact of file-level deduplication on performance, we conduct a preliminary simulation-based study of Slimmer. Our simulation approximates several of Slimmer's steps as described in Section IV-A. First, a layer from our dataset is copied to a RAM disk. The layer is then decompressed, unpacked, and the fingerprints of all files are computed using the MD5 hash function [18]. The simulation searches the fingerprint index for duplicates, and, if the file has not been stored previously, it records the file's fingerprint in the index. At this point our simulation does not include the latency of storing unique files. To simulate the layer reconstruction during a `pull` request, we archive and compress the corresponding files.

The simulator is implemented in 600 lines of Python code and our setup is a one-node Docker registry on a machine with 32 cores and 64 GB of RAM. To speed up the experiments and fit the required data in RAM we use 50% of all layers and exclude the ones larger than 50 MB. We process 60 layers in parallel using 60 threads. The entire simulation took 3.5 days to finish.

Figure 10 shows the CDF for each sub-operation of Slimmer. Unpacking, Decompression, Digest Calculation, and Searching are part of the deduplication process and together make up the Dedup time. Searching, Archiving, and Compression simulate the processing for a `pull` request and form the Pulling time.

Push: Slimmer does not directly impact the latency of `push` requests because deduplication is performed asynchronously. The appropriate performance metric for `push` is the time it takes to deduplicate a single layer. Looking at the breakdown of the deduplication time in Figure 10, we make several observations.

First, the searching time is the smallest among all operations with 90% of the searches completing in less than 4 ms and a median of 3.9 ms. Second, the calculation of digests spans a wide range from 5 μs to almost 125 s. 90% of digest calculation times are less than 27 s while 50% are less than 0.05 s. The diversity in the timing is caused by a high variety of layer sizes both in terms of storage space and file counts. Third, the run time for decompression and unpacking follows an identical distribution for around 60%

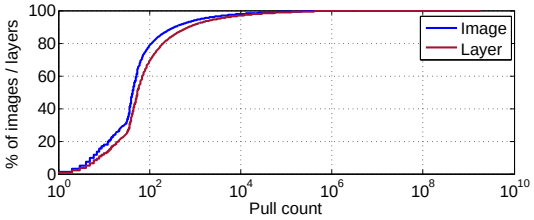


Figure 11. CDF of layer & image pull count.

of the layers and is less than 150 ms. However, after that, the times diverge and decompression times increase faster compared to unpacking times. 90% of decompressions take less than 950 ms while 90% of packing time is less than 350ms.

Overall, we see that 90% of file-level deduplication time is less than 35 s per layer, while the average processing time for a single layer is 13.5 s. This means that our single-node deployment can process about 4.4 layers/s on average (using 60 threads). In the future we will work on further improving Slimmer's deduplication throughput.

Pull: From Figure 10 we can see that 55% of the layers have close compression and archiving times ranging from from 40 ms to 150 ms and both operations contribute equally to pulling latency. After that, the times diverge and compression times increase faster with an 90th percentile of 8 s. This is because compression times increase for larger layers and follow the distribution of layer sizes (see Figure 3). Compression time makes up the major portion of the pull latency and is a bottleneck. Overall, the average pull time is 2.3 s.

C. Enhancements

To reduce deduplication overhead, we propose additional optimizations that can help to speed up Slimmer:

1) As the majority of the pull time is caused by compression, we propose to cache hot layers as precompressed tar files in the staging area. We observed that only a small proportion of images and layers are frequently requested. A majority of images and layers are *cold*. As shown in Figure 11, x-axis shows the total number of pulls since the layers/images are stored in Docker Hub to May 30, 2017.
   According to our statistics, only 10% of all images were pulled from Docker Hub more than 360 times from the time the image was first pushed to Docker Hub until May 30, 2017. Moreover, we found that 90% of pulls went to only 0.25% of images based on image pull counts. This suggests the existence of both cold and hot images and layers.
2) As deduplication provides significant storage savings, Slimmer can use faster but less effective local compression methods than gzip [19].
3) Deduplication can be expensive in terms of performance overhead, file-level dedupilication can be triggered for removing the redundant files for cold layers when the workload is low and storage utilization is high. The registries often experience fluctuation in load with peaks and troughs [20]. 80% of time, there were only 100 requests [20]. Thus, file-level deduplication can be triggered when the load is low to prevent interference with client `pull` and `push` requests.
4) To improve performance, we also suggest to use RAM to temporarily store *small* layers and directly process them in RAM to perform decompression, unpacking, file content digest calculation. According to our findings majority (87.3%) of layers that are less than 50M as shown in Figure 3. So majority of layers can be stored and processed in RAM to speed up file-level deduplication.

V. Related work

Due to its increasing popularity, Docker has recently received increased attention from the research community.

A number of studies investigated various dimensions of Docker storage performance [21], [22], [6], [23], [24], [25], [26]. Harter et al. [6] studied 57 images from Docker Hub for a variety of metrics but not for data redundancy. The authors used the results from their study to derive a benchmark to evaluate the push, pull, and run performance of Docker graph drivers based on the studied images. Compared to Slacker, our analysis focuses on the entire Docker Hub dataset. Cito et al. [27] conducted an empirical study of 70,000 Dockerfiles, focusing on the image build process but not image contents. However, their study did not focus on actual image data. Shu et al. [28] studied the security vulnerabilities in Docker Hub images based on 356,218 images. Anwar et al. [20] performed a detailed analysis of an IBM Docker registry workload but not the dataset. Dockerfinder [29] is a microservice-based prototype that allows searching for images based on multiple attributes, e.g., image name, image size, or supported software distributions. It also crawls images from remote Docker registry but the authors do not provide a detailed description of their crawling mechanism. Bhimani [21] et al. characterized the performance of persistent storage options for I/O intensive containerized applications with NVMe SSDs.

Data deduplication is a well explored and widely applied technique [30], [31], [32], [33], [34]. A number of studies characterized deduplication ratios of real-world datasets [35], [36], [37], [38], [39], [40], [41] but to the best of our knowledge, we are the first to analyze a large-scale Docker registry dataset for its deduplication properties.

VI. Conclusion

Data deduplication has proven itself as a highly effective technique for eliminating data redundancy. In spite of being

successfully applied to numerous real datasets, deduplication bypassed the promising area of Docker images. In this paper, we propose to fix this striking omission. We analyzed over 1.7 million real-world Docker image layers and identified that file-level deduplication can eliminate 96.8% of the files resulting in a capacity-wise deduplication ratio of 6.9×. We proceeded with a simulation-based evaluation of the impact of deduplication on the Docker registry performance. We found that restoring large layers from registry can slow down `pull` performance due to compression overhead. To speed up Slimmer, we suggested several optimizations. Our findings justify and lay way for integrating deduplication in the Docker registry.

Future work: In the future, we plan to investigate the effectiveness of sub-file deduplication for Docker images and to extend our analysis to more image tags rather than just the `latest` tag. We also plan to proceed with a complete implementation of Slimmer.

REFERENCES

[1] P. Menage, "Adding Generic Process Containers to the Linux Kernel," in *Linux Symposium'07*.

[2] "Docker," https://www.docker.com/.

[3] 451 Research, "Application Containers Will Be a $2.7Bn Market by 2020," https://tinyurl.com/ya358jbn.

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

[5] "GitHub," https://github.com/.

[6] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast Distribution with Lazy Docker Containers," in *USENIX FAST'16*.

[7] "spark," https://spark.apache.org/.

[8] "hdfs," https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

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

[10] "singularity," http://singularity.lbl.gov/.

[11] "Dockerfile," https://docs.docker.com/engine/reference/builder/.

[12] "cowsay," https://github.com/piuccio/cowsay.

[13] "python-magic," https://github.com/ahupp/python-magic.

[14] "Google test," https://github.com/google/googletest.

[15] "go-ethereum," https://github.com/ethereum/go-ethereum.

[16] "Android Native Development Kit (NDK)," https://github.com/android-ndk/ndk.

[17] "xnu-chroot-environment," https://github.com/winocm/xnu-chroot-environment.

[18] R. Rivest, "The md5 message-digest algorithm," 1992.

[19] "Extremely Fast Compression algorithm," https://github.com/lz4/lz4.

[20] A. Anwar, M. Mohamed, V. Tarasov, M. Littley, L. Rupprecht, Y. Cheng, N. Zhao, D. Skourtis, A. S. Warke, H. Ludwig, D. Hildebrand, and A. R. Butt, "Improving docker registry design based on production workload analysis," in *USENIX FAST'18*.

[21] 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 *IEEE IPCCC'16*.

[22] R. S. Canon and D. Jacobsen, "Shifter: Containers for HPC," in *Cray User Group'16*.

[23] R. P. Spillane, W. Wang, L. Lu, M. Austruy, R. Rivera, and C. Karamanolis, "Exo-clones: Better Container Runtime Image Management Across the Clouds," in *USENIX HotStorage'16*.

[24] 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 *IEEE AMLCS'17*.

[25] F. Zhao, K. Xu, and R. Shain, "Improving Copy-on-Write Performance in Container Storage Drivers," in *SNIA SDC'16*.

[26] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci, "Cntr: Lightweight OS containers," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 199–212. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/thalheim

[27] 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 *IEEE MSR'17*.

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

[29] A. Brogi, D. Neri, and J. Soldani, "DockerFinder: Multi-attribute Search of Docker Images," in *IEEE IC2E'17*.

[30] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble, "Sparse indexing: Large scale, inline deduplication using sampling and locality," in *USENIX FAST '09*.

[31] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *ACM SOSP'01*.

[32] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti, "iDedup: Latency-aware, inline data deduplication for primary storage," in *USENIX FAST'12*.

[33] V. Tarasov, D. Jain, G. Kuenning, S. Mandal, K. Palanisami, P. Shilane, S. Trehan, and E. Zadok, "Dmdedup: Device mapper target for data deduplication," in *Ottawa Linux Symposium'14*.

[34] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the Data Domain deduplication file system," in *USENIX FAST '08*.

[35] K. Jin and E. Miller, "The effectiveness of deduplication on virtual machine disk images," in *ACM SYSTOR'09*.

[36] M. Lu, D. Chambliss, J. Glider, and C. Constantinescu, "Insights for data reduction in primary storage: A practical analysis," in *ACM SYSTOR'12*.

[37] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, and J. Kunkel, "A study on data deduplication in HPC storage systems," in *ACM/IEEE SC'12*.

[38] H. Shim, P. Shilane, and W. Hsu, "Characterization of incremental data changes for efficient data protection," in *USENIX ATC'13*.

[39] Z. Sun, G. Kuenning, S. Mandal, P. Shilane, V. Tarasov, N. Xiao, and E. Zadok, "A long-term user-centric analysis of deduplication patterns," in *IEEE MSST'16*.

[40] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu, "Characteristics of backup workloads in production systems," in *USENIX FAST'12*.

[41] R. Zhou, M. Liu, and T. Li, "Characterizing the efficiency of data deduplication for big data storage management," in *IEEE IISWC'13*.