An End-to-End High-Performance Deduplication Scheme for Docker Registries and Docker Container Storage Systems

NANNAN ZHAO, Research & Development Institute of Northwestern Polytechnical University in Shenzhen, Shenzhen, China
MUHUI LIN, Alibaba Group, Hangzhou, China
HADEEL ALBAHAR, Kuwait University, Kuwait
ARNAB K. PAUL, BITS Pilani, K K Birla Goa Campus, Goa, India
ZHIJIE HUANG, Northwestern Polytechnical University, Xi'an, China
SUBIL ABRAHAM, Oak Ridge National Laboratory, Oak Ridge, USA
KEREN CHEN, Virginia Tech, Blacksburg, USA
VASILY TARASOV, IBM Research-Almaden, San Jose, USA
DIMITRIOS SKOURTIS, IBM Research-Almaden, San Jose, USA
ALI ANWAR, University of Minnesota, Twin Cities Campus, Minneapolis, USA
ALI R. BUTT, Virginia Tech., Blacksburg, USA

The wide adoption of Docker containers for supporting agile and elastic enterprise applications has led to a broad proliferation of container images. The associated storage performance and capacity requirements place a high pressure on the infrastructure of **container registries** that store and distribute images and **container storage systems** on the Docker client side that manage image layers and store ephemeral data generated at container runtime. The storage demand is worsened by the large amount of duplicate data in images. Moreover, container storage systems that use Copy-on-Write (CoW) file systems as storage drivers exacerbate the redundancy. Exploiting the high file redundancy in real-world images is a promising approach to drastically reduce the growing storage requirements of container registries and improve the space efficiency of container storage systems. However, existing deduplication techniques significantly degrade the performance of both registries and container storage systems because of data reconstruction overhead as well as the deduplication cost.

We propose DupHunter, an end-to-end deduplication scheme that deduplicates layers for both Docker registries and container storage systems while maintaining a high image distribution speed and container I/O performance. DupHunter is divided into 3 tiers: registry tier, middle tier, and client tier. Specifically, we first build a high-performance deduplication engine at the registry tier that not only natively deduplicates layers for space savings but also reduces layer restore overhead.

Authors' addresses: Nannan Zhao, nannanzhao@nwpu.edu.cn, Research & Development Institute of Northwestern Polytechnical University in Shenzhen, Shenzhen, No. 45th Gaoxin South 9th Road, Guangdong, China, 518063; Muhui Lin, muhui.lmh@alibaba-inc.com, Alibaba Group, Hangzhou, China, 310000; Hadeel Albahar, hadeel.albahar@ku.edu.kw, Kuwait University, Sabah Al-Salem University City, Kuwait; Arnab K. Paul, arnabp@goa.bits-pilani.ac.in, BITS Pilani, K K Birla Goa Campus, Goa, India, 403726; Zhijie Huang, jayzy.huang@nwpu.edu.cn, Northwestern Polytechnical University, Xi'an, Shaanxi, China, 710129; Subil Abraham, abrahams@ornl.gov, Oak Ridge National Laboratory, Oak Ridge, TN, USA, 37830; Keren Chen, keren9@vt.edu, Virginia Tech, Blacksburg, VA, USA, 24061; Vasily Tarasov, vtarasov@us.ibm.com, IBM Research-Almaden, San Jose, CA, USA, 95120; Dimitrios Skourtis, skourtis@soe.ucsc.edu, IBM Research-Almaden, San Jose, CA, USA, 95120; Ali Anwar, aanwar@umn.edu, University of Minnesota, Twin Cities Campus, Minneapolis, MN, USA, 55455; Ali R. Butt, butta@cs.vt.edu, Virginia Tech., Blacksburg, VA, USA, 24061.

ACM 1553-3077/2024/1-ART
https://doi.org/10.1145/3643819

Then, we use deduplication offloading at the middle tier to eliminate the redundant files from the client tier and avoid bringing deduplication overhead to the clients. To further reduce the data duplicates caused by CoWs and improve the container I/O performance, we utilize a container-aware storage system at the client tier that reserves space for each container and arranges the placement of files and their modifications on the disk to preserve locality. Under real workloads, DupHunter reduces storage space by up to 6.9× and reduces the GET layer latency up to 2.8× compared to the state-of-the-art. Moreover, DupHunter can improve the container I/O performance by up to 93% for reads and 64% for writes.

CCS Concepts: • **Information systems → Distributed storage**; **Deduplication**; • **Software and its engineering → File systems management**.

Additional Key Words and Phrases: Docker registry, Docker storage driver, Linux file system, Deduplication

1 INTRODUCTION

Containerization frameworks such as Docker [16] have been widely adopted in modern cloud environments. This is due to their lower overhead compared to virtual machines [1, 43], a rich ecosystem that eases application development, deployment, and management [13], and the growing popularity of microservices [83]. By now, all major cloud platforms endorse containers as a core deployment technology [5, 31, 35, 55]. For example, Datadog reports that in 2018, about 21% of its customers' monitored hosts ran Docker and that this trend continues to grow by about 5% annually [15].

As the container market continues to expand, the amount of container images grows rapidly. For example, Docker Hub alone stored over 9 million public images [17] and the image dataset continues to grow. We believe that this is just the tip of the iceberg and the number of *private* images is significantly higher. Other popular cloud public registries [4, 30, 39, 54], as well as on-premises registry deployments in large organizations, experience a similar surge in the number of images. As a result, organizations spend an increasing amount of their storage and networking infrastructure on operating image registries. The storage demand for container images is worsened by the large amount of duplicate data in images. As Docker images must be self-contained by definition, different images frequently include the same, common dependencies (e.g., libraries). Consequently, different images are prone to contain a high number of duplicate files as shared components exist in more than one image. To reduce this redundancy, Docker employs a layer sharing technique since images are structured in read-only *layers*. However, this is insufficient as layers are coarse and rarely identical because they are built by developers independently and without coordination. Indeed, a recent analysis of the Docker Hub image dataset showed that about 97% of files across layers are duplicates [90].

The redundant data across different images not only wastes storage resources and slows down image distribution for Docker registries, but also causes storage inefficiency for container storage systems on the Docker client side. Moreover, Docker container storage systems that use the Copy-on-Write (CoW) mechanism further exacerbate this redundancy and impose significant I/O overhead to containerized applications. The Docker container storage typically leverages union file systems, such as Overlay2 file system [76] which is a shim layer sitting atop backing file systems (e.g., Ext4 file system) to merge all layers of an image and presents a single file system tree to the container instance. Since layers are immutable and shared across both images and containers, Overlay2 simply uses the CoW mechanism, wherein writing to a file in a read-only layer causes a copy of the whole file being written to a separate, per-container writable layer. The CoW mechanism negatively impacts the I/O performance of containerized applications [73]. Furthermore, the backing file systems, such as Ext4 file system [44, 50] which places and manages all layers of a container instance unaware of the container instance. Therefore the backing file systems cannot differentiate allocations for the same container. Our experiments show that Ext4 places the writable layer of a container instance far away from the read-only layers, which negatively impacts both locality and I/O performance, especially for HDD-based storage systems (detailed in §3.2).

Deduplication is an effective method to reduce capacity demands of intrinsically redundant datasets [60]. However, applying deduplication to a Docker registry or container storage system is challenging. First, layers are stored in the registry as **compressed** tarballs that do not deduplicate well [52]. And decompressing layers first and storing individual files incurs high reconstruction overhead and slows down image pulls. The slowdowns during image pulls are especially harmful because they contribute directly to the startup times of containers. Our experiments show that, on average, naive deduplication increases layer pull latencies by up to 98× compared to a registry without deduplication. Second, applying deduplication on the container storage system incurs substantial CPU and memory overheads, such as data fingerprint calculations and index searches, consequently slowing down image pulls and impairing the performance of container runtime operations.

In this paper, we propose DupHunter, an end-to-end deduplication framework to increase storage efficiency for both Docker registry and container storage system while reducing the corresponding overhead. DupHunter is composed of three tiers: registry tier, middle tier, and client tier. DupHunter first utilizes domain-specific knowledge about the stored data on registry storage system and user access pattern to reduce the impact of layer deduplication on image distribution performance at the Docker registry tier. Then, it exploits the registry's global knowledge about layer deduplication to eliminate the redundant files from container storage system by using deduplication offloading at the middle tier. Moreover, to further mitigate redundant data and eliminate CoW overhead from container storage system at the client tier, DupHunter uses a novel container-aware storage system that places all files of a container instance closer and redirects updates instead of using CoW to preserve locality and speed up I/O performance.

For this purpose, DupHunter offers three key contributions:

(1) DupHunter exploits existing redundancy policies and user access patterns to improve image distribution performance for Docker registry. It keeps a specified number of layer replicas as-is, without decompressing and deduplicating them. Accesses to these replicas do not experience layer restoring overhead. Any additional layer replicas needed to guarantee the desired availability are decompressed and deduplicated. Moreover, DupHunter *monitors user access patterns* and proactively restores layers before layer download requests arrive. This allows it to avoid reconstruction latency during pulls.
(2) DupHunter deduplicates the container storage system by offloading deduplication to registries to avoid imposing deduplication overhead on the client side. DupHunter registry only sends *deduplicated layers* to clients to eliminate redundant files on the client side, reduce network load, and consequently speed up image pulls and container startups.
(3) DupHunter uses a container-aware storage system to make sure that files of a container are placed closer on the disk to maintain good locality and speed up container I/O performance. The container-aware storage system reserves space for each container during image unpacking. Changes to the files in the read-only layers are redirected to the preallocated space instead of performing CoW operations.

We evaluate DupHunter on a 6-node cluster using real-world workloads and layers. DupHunter outperforms the state-of-the-art Docker registry, Bolt [48], by reducing layer pull latencies by up to 2.8×. On the Docker client side, DupHunter can improve the container I/O performance by up to 93% for reads and 64% for writes. Moreover, DupHunter reduces storage consumption by up to 6.9×.

2 BACKGROUND AND RELATED WORK

We first provide the background on the Docker registry and container storage system including storage drivers and backing file systems, then discuss existing deduplication works.

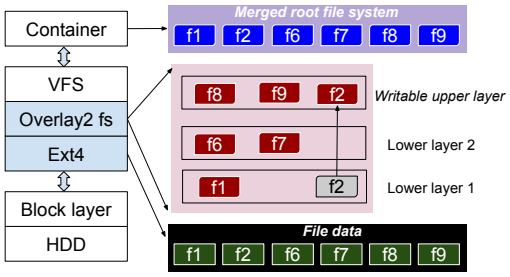


Fig. 1. Container storage system.

2.1 Docker Registry

The main purpose of a Docker registry is to store and distribute container images to Docker clients. A registry provides a REST API for Docker clients to *push* images to and *pull* images from the registry [18, 19]. Docker registries group images into *repositories*, each containing versions (*tags*) of the same image, identified as `<repo-name:tag>`. For each tagged image in a repository, the Docker registry stores a *manifest*, i.e., a JSON file that contains the runtime configuration for a container image (e.g., environment variables) and the list of layers that make up the image. A layer is stored as a compressed archival file and identified using a digest (SHA-256) computed over the uncompressed contents of the layer. When pulling an image, a Docker client first downloads the manifest and then the referenced layers (that are not already present on the client). When pushing an image, a Docker client first uploads the layers (if not already present in the registry) and then the manifest.

The current Docker registry software is a single-node application with a RESTful API. The registry delegates storage to a backend storage system through corresponding storage drivers. The backend storage can range from local file systems to distributed object storage systems such as Swift [59] or others [3, 36, 59, 72]. To scale the registry, organizations typically deploy a load balancer or proxy in front of several independent registry instances [6]. In this case, client requests are forwarded to the destination registries through a proxy, then served by the registries' backend storage system. To reduce the communication overhead between the proxy, registry, and backend storage system, Bolt[48] proposes to use a consistent hashing function instead of a proxy, distribute requests to registries, and utilize the local file system on each registry node to store data instead of using a remote distributed object storage system. Multiple layer replicas are stored on Bolt registries for high availability and reliability. DupHunter is implemented based on the architecture of Bolt registry for high scalability.

Registry performance is critical to Docker clients. In particular, the layer `pulling` performance (i.e., `GET` layer performance) impacts container startup times significantly [34]. A number of works have studied various dimensions of registry performance for a Docker image dataset [6, 9, 34, 70, 76, 89, 90]. However, such works do not provide deduplication for the registry. A community proposal exists to add file-level deduplication to container images [2], but as of now lacks even a detailed design, let alone performance analysis. Skourtis et al. [69] propose restructuring layers to optimize for various dimensions, including registry storage utilization. Their approach does not remove all duplicates, whereas DupHunter leaves images unchanged and can eliminate all duplicates in the registry. Finally, a lot of works aim to reduce the size of a single container image [20, 32, 63, 78], and are complementary to DupHunter.

2.2 Container storage system on the Docker client side

A Container storage system on the Docker client side is composed of a storage driver and a backing file system. After an image is downloaded from a Docker registry, Docker utilizes a storage driver (e.g., Overlay2 file system [73]) to store image layers on top of a backing file system (e.g., Ext4 [44]). Figure 1 shows an example of a

container storage stack. Applications inside a container instance access to a file in a specific layer are sent to the backing file system–Ext4 file system via the storage driver–Overlay2 file system.

The purpose of storage drivers is to create a pluggable architecture. During the creation of a container instance, the storage driver organizes the image layers into a particular stacking order: when files with the same name exist in multiple layers, only the one in the topmost layer is visible. In addition, a writable layer is created and stacked at the top as shown in Figure 1. Then, the storage driver presents the immutable image layers and the writable layer as a single root file system to the container instance.

Overlay2 file system. Overlay2 file system is a popular modern union file system and the default storage driver due to its good performance and simple implementation [16, 79]. To create a container from an image, Overlay2 combines a list of image layers (referred to `lowerdir`) and the container's writable layer (referred to `upperdir`) and presents them as a single directory called `merged`. `merged` is the container mount point, which presents a unified file system view to the container instance. When the `lowerdir` and the `upperdir` contain the same files, the uppdir obscures the existence of the same files in the `lowerdir`. Any modifications made to the container are written to the `upperdir`. For instance, if a file in the `lowerdir` is opened with a write-mode flag, the whole file will be copied up to the `upperdir` in spite of small changes or no updates at all.

Ext4 file system. For a container instance, the image layers and the data written inside the container are actually stored and managed by the backing file systems. Ext4 is the default file system for a majority of modern Linux distributions and a commonly used backing file system for Overlay2 due to its reliability, stability, scalability, and high performance[16, 44]. To improve the performance of sequential file read/writes and reduce metadata overhead, Ext4 stores the file content as `extents` instead of fixed blocks by allocating contiguous blocks for files, wherein each extent represents a set of logically contiguous blocks within the file and also contiguous on-disk[44]. To reduce performance degradation due to fragmentation and maintain locality, Ext4 tries to co-locate small related files and reserve space for big files that grow. Moreover, Ext4 places all the files in a directory together to maintain locality. This is because all the files in a directory may be related, therefore it is useful to place them together.

Related work on container storage system optimizations. Many storage drivers leverage CoW operations to create layers efficiently. A disk block can be shared by multiple files due to block-level CoWs. Since file access and caching are based on logic file address, reading a block from the disk multiple times brings multiple redundant copies to the cache. TotalCoW [85], HP-Mapper [33], and [89] are proposed to reduce redundant cached file blocks due to CoWs. Their methods include identifying if the requested blocks exist in the page cache, mapping the file (logical) block to the disk block in the page cache, etc. These methods focus on improving cache efficiency for CoWs while our container-aware storage system (detailed in §4.4) targets optimizing storage efficiency for disks. Therefore, these methods are complementary to our container-aware storage system.

CAST [84] enables simultaneous CoW operations to avoid the overhead of searching and modifying files. This is due to the fact that when a container modifies a file in the lower layer, Overlay2 will first copy the file into the upper layer and rename the file to the upper layer to hide the corresponding file in the lower layer. During the rename operation, Overlay2 uses a global lock to prevent deadlock. CAST disables the global lock for CoW operations since the rename operations do not cause deadlock, consequently improving concurrent CoW performance. Similarly, CAST is also complementary to our container-aware storage system.

Baoverlay [73] is a block-accessible Overlay file system that provides fine-grained accessibility to files so that CoW to a file only involves copying a number of blocks rather than the whole file. For a partial-block write, Baoverlay first writes the new data into the upper layer. Baoverlay defers the copying of the remaining portions to the future read requests. In this case, Baoverlay can significantly degrade read performance. Similarly to

Overlay2 file system, Baoverlay ignores the data placement and locality for container instances, thus performing inadequately for HDD-based storage systems.

Related work on container storage system analysis. Many researchers have conducted a comprehensive analysis of container storage drivers in different aspects to present the challenges and projections for future container storage system design. For example, Qiumin Xu et al. [86] compare different container storage drivers and analyze the performance impact among different storage options (e.g., direct attached SSD and NVMf storage) for deploying Docker containers. Vasily Tarasov et al. [77] aim to provide the choice of an appropriate storage solution for Docker containers by analyzing the performance impact of different storage drivers, layers, and devices under different concurrency levels.

2.3 Deduplication

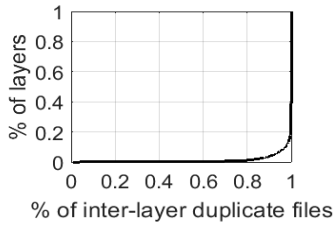
Data deduplication has received considerable attention, particularly for virtual machine images [37, 40, 71, 91]. Many deduplication studies focus on primary and backup data deduplication [26–28, 46, 47, 49, 56, 68, 75, 82, 92] and show the effectiveness of file- and block-level deduplication [53, 74]. To further reduce storage space, integrating block-level deduplication with compression has been proposed [80]. In addition to local deduplication schemes, a global deduplication method [57] has also been proposed to improve the deduplication ratio and provide high scalability for distributed storage systems.

Data restoring latency is an important factor for storage systems with deduplication support. Efficient chunk caching algorithms and forward assembly are proposed to accelerate data restore performance [10]. At first glance, one could apply existing deduplication techniques to solve the issue of high data redundancy among container images. However, as we demonstrate in detail in §3.3, such a naive approach leads to slow reconstruction of layers on image pulls, which severely degrades container startup times. DupHunter is specifically designed for Docker registries, which allows it to leverage image and workload information to reduce deduplication and layer restore overhead.

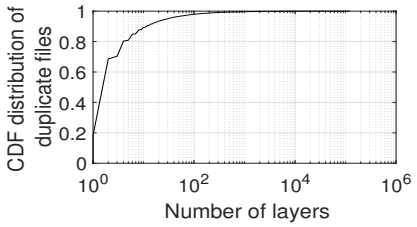
Deduplication related work on Docker images. BED [88] provides block-level deduplication for both the registry and client side. During image pulling, BED client first pulls the image's block fingerprint list and calculates the unique block fingerprints based on local block fingerprints, then downloads the unique block fingerprints. Afterwards, BED client reconstructs the image. The calculation of unique block fingerprints and image reconstruction incur considerable overhead (e.g., CPU and memory) on the client side, and consequently slow down image pulling especially when only a small number of images are been pulled by the client [88].

Gear [22] is a file-level deduplication method for both the registry and client side by using a special image format – Gear image. Gear image is one of the on-demand remote image formats that are proposed in [34, 45] to speedup container deployment. Since Gear images are not compatible with Docker images, Docker images have to be converted into Gear images and then pulled by Gear clients. As the amount of Docker images stored in Docker Hub is continuously increasing, it is infeasible to convert all Docker Hub images to Gear images in advance. If the requested image has not been converted into a Gear image, the image has to be first converted into a Gear image and then pulled by the Gear client. However, the conversion process to the Gear image format takes considerable time because it involves file system traversal, file fingerprint calculations, and Gear image build time, leading to longer container deployment time. Another drawback of Gear is that it still needs to download the required files during the run phase, which degrades container running performance.

Our proposed scheme, DupHunter, avoids introducing data deduplication overhead to the client side as opposed to BED. Also, compared to Gear, DupHunter uses Docker images and does not degrade container running performance. Moreover, the DupHunter registry provides multiple different deduplication mode options and better space efficiency than Gear. In addition, Gear and BED focus more on reducing the amount of data transferred



(a) CDF of inter-layer duplicate file ratio.



(b) CDF of duplicate files across layers(X-axis is log-scale)

Fig. 2. Inter-layer duplicate files.

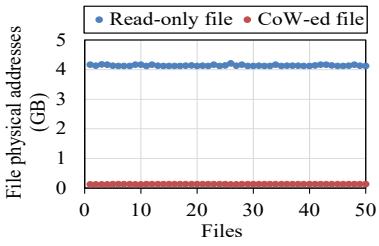


Fig. 3. Ext4 allocation for CoWs.

during pulling operations. DupHunter delves deeper into optimizing the efficacy for both Docker registries and Docker container storage systems. For example, the DupHunter registry uses prefetching/preconstruction to speed up layer pullings and provides various deduplication modes to support a range of uses (detailed in §4.2). The DupHunter client utilizes a container-aware storage system that not only avoids bringing the redundant data caused by CoWs but also improves data locality for good I/O performance (detailed in §4.4).

3 MOTIVATING OBSERVATIONS

The need and feasibility of DupHunter is based on four key observations: 1) container images have a lot of redundancy; 2) container storage system has bottlenecks; 3) existing scalable deduplication technologies significantly increase image pull latencies; and 4) image access patterns can be predicted reliably.

3.1 Redundant files across layers

Container image layers exhibit a large degree of redundancy in terms of duplicate files. Although Docker supports the sharing of layers among different images to remove some redundant data in the Docker registry, this is not sufficient to effectively eliminate duplicates. According to the deduplication analysis of the Docker Hub dataset [90], 97% of files have more than one file duplicate, resulting in a deduplication ratio of 2× in terms of capacity. We believe that the deduplication ratio is much higher when private repositories are taken into account.

The duplicate files are executables, object code, libraries, and source code, and are likely imported by different image developers using package installers or version control systems such as `apt`, `pip`, or `git` to install similar dependencies. However, as layers often share many but not all files, this redundancy cannot be eliminated by Docker's current layer sharing approach. *R*-way replication for reliability further fuels the high storage demands of Docker registries. Hence, satisfying demand by adding more disks and scaling out storage systems quickly becomes expensive.

Cross-layer duplicate files. To further study the deduplication characteristics for layers, we analyze a 47 TB image dataset [12] and show the CDF distribution of inter-layer duplicate files (Figure 2(a)) and CDF of duplicate files across layers (Figure 2(b)). Note that inter-layer duplicate files are files that are duplicated in more than one layer. As shown in Figure 2(a), less than 0.006% of layers do not have any inter-layer redundant files, meaning that the majority of layers contain cross-layer duplicate files. As shown in Figure 2(b), we see that 19% of files and their redundant copies (if they have) are stored in the same layers. These redundant copies are called intra-layer duplicate files. The remaining 81% of the files and their redundant copies are scattered in multiple layers, indicating that the majority of redundant files are cross-layer duplicate files. Although the Docker client

can be configured to avoid downloading redundant layers, it cannot guarantee that cross-layer redundant files are not repeatedly downloaded, resulting in wasted network bandwidth and slow layer pulling operations.

3.2 Bottlenecks in the container storage system

While containerized applications run independently and are isolated in the containers, they lack isolation from the operating system. For example, containers share the same backing file system, therefore files from different containers are interleaved because the backing file systems does not differentiate allocations for the same container. Specifically, when Ext4 is used as the backing file system for Docker, all the layers of an image are first downloaded from Docker registry, then decompressed and unpacked to the underlying Ext4 file system. During layer unpacking, Ext4 places all the files of each layer directory close to each other to maintain locality. Moreover, Ext4 stores each file contiguously by allocating a single extent for each file. When a container is instantiated later, an `upperdir` is then created and stored on Ext4. Since Ext4 lacks the knowledge that the previously unpacked image layers (i.e., lower layers) and the newly created `upperdir` belong to the same container instance, the `upperdir` and its future containing files are not placed close to the lower layers.

We used a test case to illustrate the fragmentation of the container storage system using Overlay2 as the storage driver and Ext4 as the backing file system (see §6 for a more detailed description of the system setup), as shown in Figure 3. We first launched a container from a fio image [25] with a `lower` layer made up of 50 16KB read-only files, then wrote 4KB data to each file, which resulted in 50 files first copied to the container's `upper` layer and then modified correspondingly. Figure 3 shows how far these two layers are placed on disk. Note that the physical addresses, measured by using `filefrag` [24], are the first physical block numbers allocated to the files. Ext4 is not aware that the two layers belong to the same container and should be placed close to each other, resulting in poor locality. As depicted in Figure 3, the distance between the two layers is 4 GB.

Additionally, exacerbating the fragmentation issue described above, storage drivers that use CoW file systems suffer from significant CoW overhead. For example, modifying a file in a read-only layer results in the entire file being copied to the upper layer, leading to high data redundancy and write penalties [33].

3.3 Drawbacks of Existing Technologies

A naive approach to eliminating duplicates in container images could be to apply an existing deduplication technique. To experimentally demonstrate that such a strategy has significant shortcomings, we try four popular local deduplication technologies, `VDO` [81], `Btrfs` [8], `ZFS` [87], `Jdupes` [38], in a single-node setup and on one distributed solution, `Ceph` [11], on a 3-node cluster. The deduplication block sizes are set to 4KB for both `VDO` and `Ceph`, and 128KB for both `Btrfs` [8] and `ZFS` [87] by default. Table 1 presents the deduplication ratio and pull latency overhead for each technology in two cases: 1) when layers are stored compressed (as-is); and 2) when layers are uncompressed and unpacked into their individual files. Note that the deduplication ratios are calculated against the case when all layers are compressed (the details of the dataset and testbed are presented in §6).

Deduplication ratios. Putting the original compressed layer tarballs in any of the deduplication systems results, unsuprisingly, in a deduplication ratio of 1. This is because even a single byte change in any file in a tarball scrambles the content of the compressed tarball entirely[14, 52]. Hence, to expose the redundancy to the deduplication systems, we decompress every layer before storing it.

After decompression, all deduplication schemes yield significant deduplication ratios. `Jdupes`, `Btrfs`, and `ZFS` reduce the dataset to about half and achieve deduplication ratios of 2.1, 2.3, and 2.3, respectively. `Ceph` has a higher deduplication ratio since it uses a smaller deduplication block size, while `VDO` shows the highest deduplication ratio as it also compresses deduplicated data.

It is important to note that for an enterprise-scale registry, a large number of storage servers need to be deployed and single-node deduplication systems (`Jdupes`, `Btrfs`, `ZFS`, and `VDO`) can only deduplicate data within

Table 1. Dedup. ratio vs. increase in GET layer latency.

Technology	Dedup ratio, compressed layers	Dedup ratio, uncompressed layers	GET latency increase wrt. uncompressed layers
Jdupes	1	2.1	36 ×
VDO	1	4	60 ×
Btrfs	1	2.3	51 ×
ZFS	1	2.3	50 ×
Ceph	1	3.1	98 ×

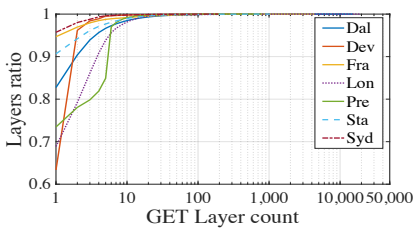


Fig. 4. CDF of GET layer request count.

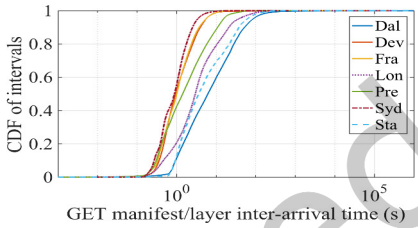


Fig. 5. CDF of GET manifest/layer inter-arrival time.

a single node. Therefore, in a multi-node setup, such solutions can never achieve optimal *global* deduplication, i.e., duplication across nodes.

Pull latencies. To analyze layer pull latencies, we implement a layer restoring process for each technology. Restoring includes fetching files, creating a layer tarball, and compressing it. We measure the average GET layer latency and calculate the restore overhead compared to GET requests without layer deduplication.

As shown in Table 1, the restoration overhead is high. The file-level deduplication scheme Jdupes increases the GET layer latency by 36×. This is caused by the expensive restoring process. Btrfs, ZFS, and VDO show an increase of more than 50×, as they are block-level deduplication systems, and hence they also add file restoring overhead. The overhead for Ceph is the highest because restoration is distributed and incurs network communication.

In summary, our analysis shows that while existing technologies can provide storage space savings for container images (after decompression), they incur high cost during image pulls due to slow layer reconstruction. At the same time, pull latency constitutes the major portion of container startup times even without deduplication. According to [34], pulling images accounts for 76% of container startup times. This means that, for example, for Btrfs the increase of layer GET latency by 51× would prolong container startup times by 38×. Hence, deduplication has a major negative impact on the startup times of containerized applications.

3.4 Predictable User Access Patterns

A promising approach to mitigate layer restoring overhead is predicting which layers will be accessed and preconstruct them. In DupHunter, we can exploit the fact that when a Docker client pulls an image from the registry, it first retrieves the image manifest, which includes references to the image layers.

User pulling patterns. Typically, if a layer is already stored locally, then the client will not fetch this layer again. We use the IBM Cloud registry workload [6] to analyze the user pulling patterns. The traces span ~80 days for 7 registry clusters: Dallas, Frankfurt, London, Sydney, Development, Prestaging, and Staging. Figure 4 shows the CDF of layer GET counts by the same clients. The analysis shows that the majority of layers are only fetched once by the same clients. For example, 97% of layers from Syd are only fetched once by the same clients. This suggests that, by observing access patterns, we are able to predict, whether they will pull a layer or not by keeping track of user access history.

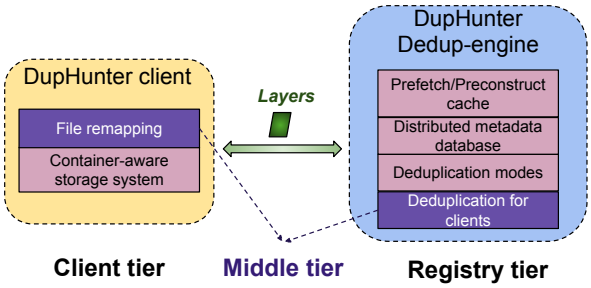


Fig. 6. DupHunter architecture.

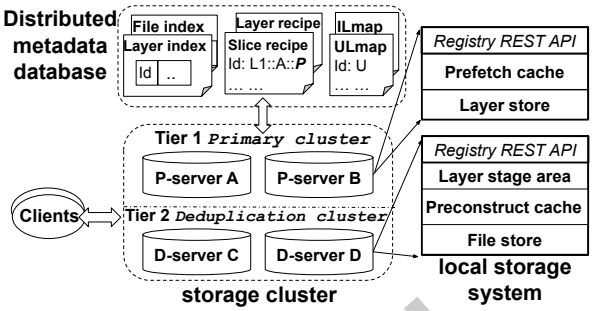


Fig. 7. DupHunter registry architecture.

Layer preconstruction. We analyze the inter-arrival time between a GET manifest request and the subsequent GET layer request. As shown in Figure 5, the majority of intervals are greater than 1 second. For example, 80% of intervals from London are greater than 1 second, and 60% of the intervals from Sydney are greater than 5 seconds. There are several reasons for this long gap. First, when fetching an image from a registry, the Docker client fetches a fixed number of layers in parallel (three by default) starting from the lowest layer. In the case where an image contains more than three layers, the upper layers have to wait until the lower layers are downloaded, which delays the GET layer request for these layers. Second, network delay between clients and registry often accounts for a large portion of the GET latency in cloud environments. As we show in §6, layer preconstruction can significantly reduce layer restoring overhead. In the case of a shorter duration between a GET manifest request and its subsequent GET layer requests, layer preconstruction can still be beneficial because the layer construction starts prior to the arrival of the GET request.

4 DUPHUNTER DESIGN

In this section, we first provide an overview of the 3-tier DupHunter deduplication model (§4.1). We then describe in detail how the end-to-end deduplication is performed and improves performance across the following three tiers: DupHunter registry tier (§4.2), Middle tier (§4.3), DupHunter client[1] tier (§4.4).

4.1 Overview

Figure 6 shows the 3-tier architecture of DupHunter that comprises a client tier, a middle tier, and a registry tier. First, DupHunter registry tier is responsible for storing and distributing Docker images to clients. DupHunter uses a Dedup-engine to mitigate redundant data(§4.2). Second, the middle tier is a logical tier that offloads deduplication from the client tier to the registry tier to eliminate deduplication overhead for clients(§4.3). Third, the client tier is responsible for fetching images from the registry tier, storing them on the local storage system, and running Docker containers.

Dedup-engine at registry tier. As shown in Figure 6, Dedup-engine provides different deduplication modes by performing deduplication and uses prefetch/preconstruct cache to speedup image distribution(detailed in §4.2). The deduplication metadata and user access information are kept in the database.

Deduplication offloading at middle tier. Deduplication offloading involves both the registry tier and the client tier as shown in Figure 6 (detailed in §4.3). Upon a GET layer request, Dedup-engine will first *deduplicate* the requested layer for clients by utilizing the layer deduplication-related information and user access history, and

[1]The client refers to Docker's container-based platform that includes the Docker client and the Docker daemon.

then send a *deduplicated layer* to the client. After receiving the deduplicated layer, the client uses a file remapping technique to share files among different images and containers.

Container-aware storage system at client tier. DupHunter uses a container-aware storage system to optimize data layout, reduce the redundant data, and speed up container I/O requests as shown in Figure 6 (§4.4). Upon receiving an image, the container-aware storage system reserves space for the container instance and places the container data closer for good locality.

4.2 Dedup-engine at registry tier

As shown in Figure 7, Dedup-engine consists of two parts: 1) a cluster of *storage servers*, each exposing the registry REST API for DupHunter clients; and 2) a distributed *metadata database* for storing deduplication metadata. Dedup-engine uses three techniques to reduce deduplication and restore overhead for registries: 1) replica deduplication modes; 2) parallel layer reconstruction; and 3) proactive layer prefetching/preconstruction.

Replica deduplication modes. For higher fault tolerance and availability, existing registry setups replicate layers. Dedup-engine also performs layer replication, but additionally deduplicates files inside the replicas.

A *basic deduplication mode n* (B-mode *n*) defines that Dedup-engine should only keep *n* layer replicas intact and deduplicate the remaining $R - n$ layer replicas, where R is the layer replication level. At one extreme, B-mode R means that no replicas should be deduplicated, and hence provides the best performance but no data reduction. At the other end, B-mode 0 deduplicates all layer replicas, i.e., it provides the highest deduplication ratio but adds restoration overhead for `GET` requests. The remaining in-between B-modes allow to trade off performance for data reduction.

For heavily skewed workloads, Dedup-engine also provides a *selective deduplication mode* (S-mode). The S-mode utilizes the skewness in layer popularity, observed in [6], to decide how many replicas should be deduplicated for each layer. As there are hot layers that are pulled frequently, S-mode sets the number of intact replicas proportional to their popularity. This means that hot layers have more intact replicas, and hence can be served faster, while cold layers are deduplicated more aggressively.

Deduplication in Dedup-engine, for the example of B-mode 1, works as follows: Dedup-engine first creates 3 layer replicas across 3 servers. It keeps a single layer replica as the *primary layer replica* on one server. Deduplication is then carried out in one of the other servers storing a replica, i.e., the layer replica is decompressed and any duplicate files are discarded while unique files are kept. The unique files are replicated and saved on different servers for fault tolerance. Once deduplication is complete, the remaining two layer replicas are removed. Any subsequent `GET` layer requests are sent to the primary replica server first since it stores the complete layer replica. If that server crashes, one of the other servers is used to rebuild the layer and serve the `GET` request.

To support different deduplication modes, Dedup-engine stores a mix of both layer tarballs and individual files. This makes data placement decision more complex with respect to fault tolerance because individual files and their corresponding layer tarballs need to be placed on different servers. As more tarballs and files are stored in the cluster, the placement problem gets more challenging.

To avoid accidentally co-locating layer tarballs and unique files, which are present in the tarball, and simplify the placement problem, Dedup-engine divides storage servers into two groups (Figure 7): a *primary cluster* consisting of *P-servers* and a *deduplication cluster* consisting of *D-servers*. P-servers are responsible for storing full layer tarball replicas and replicas of the manifest, while D-servers deduplicate, store, and replicate the unique files from the layer tarballs. The split allows Dedup-engine to treat layers and individual files separately and prevent co-location during placement. The split is based on deduplication modes. For example, B-mode 0 does not have P-servers while B-mode 3 does not have D-server. For the remaining modes, the partitioning of P-servers and D-servers is determined by the intact layer dataset size and the unique file dataset size. To determine the

intact layer size and the unique file dataset size, we can begin by estimating the rate of data growth using various prediction models [7, 21]. Once we have this information, we can then estimate the intact layer size and unique file dataset size by leveraging the deduplication ratios of different deduplication modes. If storage space becomes insufficient for incoming data, we add more servers to the pools. To balance the data among servers, we can use algorithms designed for data balancing and data distribution in distributed storage systems, such as CHT [41]. This paper focuses on the optimization of deduplication techniques, but we plan to research layer dataset estimation and the partitioning of P-servers and D-servers in the future.

P- and D-servers form a 2-level storage hierarchy. In the default case, the primary cluster serves all incoming `GET` requests. If a request cannot be served from the primary cluster (e.g., due to a node failure, or Dedup-engine operating in B-mode 0 or S-mode), it will be forwarded to the deduplication cluster and the requested layer will be reconstructed.

Parallel layer reconstruction. Dedup-engine speeds up layer reconstruction through parallelism. As shown in Figure 7, each D-server's local storage is divided into three parts: the layer staging area, preconstruction cache, and file store. The layer staging area temporarily stores newly added layer replicas. After deduplicating a replica, the resulting unique files are stored in a content addressable file store and replicated to the peer servers to provide redundancy. Once all file replicas have been stored, the layer replica is deleted from the layer staging area.

Dedup-engine distributes the layer's unique files onto several servers (see §4.2.1). All files on a single server belonging to the same layer are called a *slice*. A slice has a corresponding *slice recipe*, which defines the files that are part of this slice, and a *layer recipe* defines the slices needed to reconstruct the layer. This information is stored in Dedup-engine's metadata database. Each slice can be independently reconstructed at each server in parallel. This allows D-servers to rebuild layer slices in parallel and thereby improve reconstruction performance. DupHunter maintains layer and file fingerprint indices in the metadata database.

Predictive cache prefetch and preconstruction. To reduce the layer access latency, Dedup-engine employs a cache layer in both the primary and the deduplication clusters, respectively. Each P-server has an in-memory *user-behavior based prefetch cache* to reduce disk I/Os. When a `GET` manifest request is received from a user, Dedup-engine predicts which layers in the image will actually need to be `pulled` and prefetches them in the cache. Additionally, to reduce layer restoring overhead, each D-server maintains an on-disk *user-behavior based preconstruct cache*. As with the prefetch cache, when a `GET` manifest request is received, Dedup-engine predicts which layers in the image will be `pulled`, preconstructs the layers, and loads them in the preconstruct cache (see §4.2.4).

4.2.1 Deduplicating layers. Dedup-engine maintains a *layer index*. After receiving a `PUT` layer request, Dedup-engine first checks the layer fingerprint in the *layer index* to ensure an identical layer is not already stored. If not, Dedup-engine, replicates the layer r times across the P-servers and submits the remaining $R - r$ layer replicas to the D-servers. Those replicas are temporarily stored in the layer staging areas of the D-servers. Once the replicas have been stored successfully, Dedup-engine notifies the client of the request completion.

File-level deduplication. Once in the staging area, one of the D-servers decompresses the layer and starts the deduplication process. First, it extracts file entries from the tar archive. Each file entry is represented as a *file header* and the associated *file content* [29] as shown in Figure 8. The file header contains metadata such as file name, path, size, mode, and owner information. Dedup-engine records every file header in slice recipes (Figure 8) to be able to correctly restore the complete layer archive later (described below).

To deduplicate a file, Dedup-engine computes a file Id by hashing the file content and checks if the Id is already present in the file index. If present, the file content is discarded. Otherwise, the file content is assigned to a D-server and stored in its file store, and the file Id is recorded in the file index (Figure 8). The file index maps different file Ids to their physical replicas stored on different D-servers.

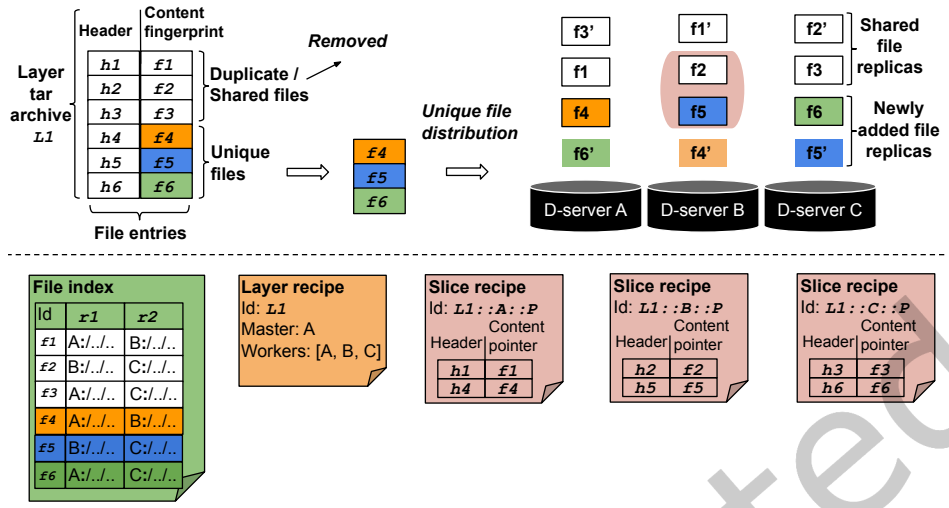


Fig. 8. Layer deduplication, replication, and partitioning.

Layer partitioning. DupHunter picks D-servers for new unique files to improve reconstruction times. For that, it is important that different layer slices are similarly sized and evenly distributed across D-servers. To achieve this, Dedup-engine employs a greedy packing algorithm. Consider first the simpler case in which each file only has a single replica. Dedup-engine first sorts the new unique files and partitions by size. Next, it assigns the largest file to the smallest partition until all the unique files are assigned. Note that during layer partitioning, Dedup-engine does not migrate existing shared files to reduce I/O overhead.

In the case where a file has more than one replica, Dedup-engine performs the above-described partitioning *per replica*. That means that it first assigns the primary replicas of the new unique files to D-servers according to the location of the primary replicas of the existing shared files. It then does the same for the secondary replicas and so on. Dedup-engine also ensures that two replicas of the same file are never placed on the same node.

Unique file replication. Next, Dedup-engine replicates and distributes the unique file replicas across D-servers based on the layer partitioning. The headers and content pointers of all files in the deduplicated layer that are assigned to a specific D-server are included in that D-server's *slice recipe* for that layer. After file replication, Dedup-engine adds the new slice recipes to the metadata database.

Dedup-engine also creates a *layer recipe* (Figure 8) for the uploaded layer and stores it in the metadata database. The layer recipe records all the D-servers that store slices for that layer and which can act as *restoring workers*. When a layer needs to be reconstructed, one worker is selected as the *restoring master*, responsible for gathering all slices and rebuilding the layer (see §4.2.2). Figure 8 shows an example deduplication process. The example assumes B-mode 1 with 3-way replication, i.e., each unique file has two replicas distributed on two different D-servers. The files $f1$, $f2$, and $f3$ are already stored in Dedup-engine, and $f1'$, $f2'$, and $f3'$ are their corresponding replicas. Layer $L1$ is being pushed and contains files $f1$–$f6$. $f1$, $f2$, and $f3$ are *shared files* between $L1$ and other layers, and hence are discarded during file-level deduplication. The unique files $f4$, $f5$ and $f6$ are added to the system and replicated to D-servers A, B, and C.

After replication, server B contains $f2$, $f5$, $f1'$, and $f4'$. Together $f2$ and $f5$ form the *primary slice* of $L1$ at server B, denoted as $L1 :: B :: P$. This slice Id contains the layer Id the slices belongs to ($L1$), the node, which stores the slice (B) and the backup level (P for primary). The two backup file replicas $f1'$ and $f4'$ on B form the

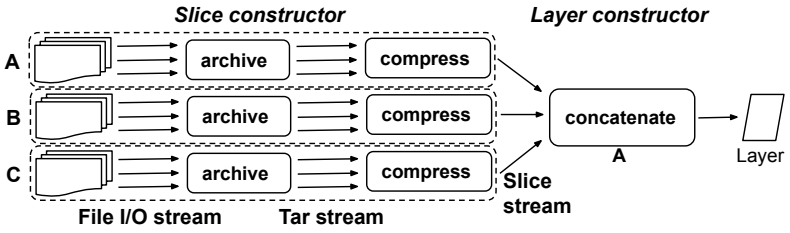


Fig. 9. Parallel streaming layer construction.

backup slice $L1 :: B :: B$. During layer restoring, $L1$ can be restored by using any combination of primary and backup slices to achieve maximum parallelism.

4.2.2 Restoring Layers. The layer restoration performance is critical to keep `pull` latencies low. Hence, Dedup-engine parallelizes slice reconstruction on multiple nodes and avoids generating intermediate files on disk to reduce disk I/O.

The restoring process works in two phases: slice reconstruction and layer reconstruction. Considering the example in Figure 8, restoring works as follows:

According to $L1$'s layer recipe, the restoring workers are three D-servers: D-servers A, B, and C. The node with the largest slice is picked as the restoring master, also called *layer constructor* (A in the example). Since A is the restoring master it sends `GET slice` requests for the primary slices to B and C. If a primary slice is missing, the master locates its corresponding backup slice by consulting the layer recipe, slice recipe, and the file index, and sends a `GET slice` request to the corresponding D-server.

After a `GET slice` request has been received, B's and C's *slice constructors* start rebuilding their primary slices in parallel and send them to A as shown in Figure 9. Meanwhile, A instructs its local slice constructor to restore its primary slice for $L1$. To construct a layer slice, a slice constructor first gets the associated slice recipe from the metadata database. The recipe is keyed by a combination of layer Id, host address and requested backup level, e.g., $L1 :: A :: P$. Based on the recipe, the slice constructor creates a slice tar file by concatenating each file header and the corresponding file contents; it then compresses the slice and passes it to the master. The master concatenates all the compressed slices into a single compressed layer tarball and sends it back to the client.

4.2.3 Integrating with different redundant policies. Note that the goal of Dedup-engine is to provide flexible deduplication modes to meet different space-saving and performance requirements and mitigate layer restore overhead. The above design of Dedup-engine mainly focuses on file-level deduplication and assumes layer replication.

To achieve a higher deduplication ratio, Dedup-engine can integrate with block-level deduplication. After removing redundant files, D-servers can further perform block-level deduplication only on unique files by using systems such as `VDO` [81] and `Ceph` [57]. However, higher deduplication ratios come with higher layer restoring overhead as the restoring latency for block-level deduplication is higher than that of file level as we show in §6. This is because to restore a layer, its associated files need to be first restored, which incurs extra overhead. Furthermore, when integrating with a global block-level deduplication scheme, the layer restoring overhead will be higher due to network communication. In this case, it is beneficial to maintain a number of layer replicas on P-servers to maintain a good performance.

While Dedup-engine exploits existing replication schemes, it is not limited to those. If the registry is using erasure coding for reliability, Dedup-engine can integrate with the erasure coding algorithm to improve space

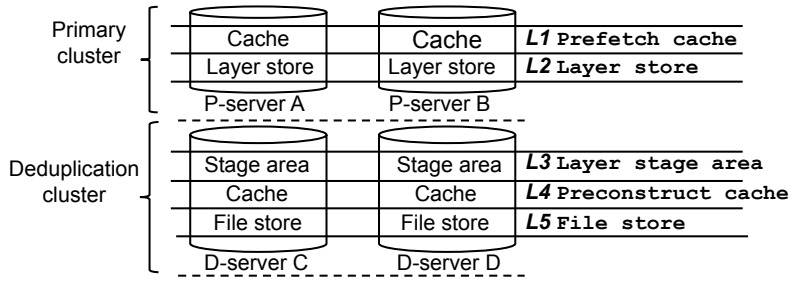


Fig. 10. Leveled storage architecture.

efficiency. Specifically, after removing redundant files from layers, Dedup-engine can store unique files as erasure-coded chunks. While Dedup-engine can not make use of existing replicas to improve pull performance in this case, its preconstruct cache remains beneficial to mitigate high restoring overheads as shown in §6.

A known side effect when performing deduplication is that the loss of a chunk has a bigger impact on fault tolerance as the chunk is referenced by several objects [67]. To provide adequate fault tolerance, Dedup-engine maintains at least three copies of a layer (either as full layer replicas or unique files that can rebuild the layer) in the cluster.

4.2.4 Caching and Preconstructing Layers. DupHunter maintains a cache layer in both the primary and deduplication clusters to speedup `pull` requests. The primary cluster cache (in-memory prefetch cache) is to avoid disk I/O during layer retrievals while the deduplication cluster on-disk cache stores preconstructed layers, which are likely to be accessed in the future. Both caches are filled based on the user access patterns seen in §3.

Request prediction. To accurately predict layers that will be accessed in the future, DupHunter keeps track of image metadata and user access patterns in two data structures: *ILmap* and *ULmap*. ILmap maps an image to its containing *layer set*. ULmap stores for each user the layers the user has accessed and the corresponding pull count. A user is uniquely identified by extracting the sender's IP address from the request. If Dedup-engine has not seen an IP address before, it assumes that the request comes from a new host, which does not store any layers yet.

When a `GET manifest` request r is received, Dedup-engine first calculates a set of image layers that have not been pulled by the user $r.addr$ by calculating the difference S_Δ between the image's layer set and the user's accessed layer set:

$$S_\Delta = ILmap[r.img] - ULmap[r.addr] \tag{1}$$

The layers in S_Δ are expected to be accessed soon. Dedup-engine then fetches the layers in S_Δ into the cache.

Cache handling in leveled storage. The introduction of the two caches results in a 5-level storage architecture of DupHunter as shown in Figure 10. Requests are passed through the tiers from top to bottom. Upon a `GET` layer request, DupHunter first determines the P-server(s) which is (are) responsible for the layer and searches the prefetch cache(s). If the layer is present, the request will be served from the cache. Otherwise, the request will be served from the layer store.

If a `GET layer` request cannot be served from the primary cluster due to a failure of the corresponding P-server(s), the request will be forwarded to the deduplication cluster. In that case, DupHunter will first lookup the layer recipe. If the recipe is not found, it means that the layer has not been fully deduplicated yet and DupHunter will serve the layer from one of the layer stage areas of the responsible D-servers. If the layer recipe is present,

DupHunter will contact the restoring master to check, whether the layer is in its preconstruct cache. Otherwise, it will instruct the restoring master to rebuild the layer.

Both the prefetch and the preconstruct caches are write-through caches. When a layer is evicted, it is simply discarded since the layers are read-only. We use an Adaptive Replacement Cache (ARC) replacement policy [51], which keeps track of both the frequently and recently used layers and adapts to changing access patterns.

4.3 Deduplication offloading at middle tier

The middle tier connects both the client side and the registry side to eliminate duplicate files from the container storage system and accelerate container deployment. When a `GET layer` request *r* is received, the Dedup-engine utilizes the layer deduplication-related metadata and user access information to remove the duplicates that have already been stored by user *r.addr* from layer *r.layer*. After that, the *deduplicated* layer *r.layer'* is sent to the user. Consequently, the duplicate files in the container storage system can be mitigated. Moreover, the amount of data that needs to be transferred to the users can be reduced.

4.3.1 Removing duplicate files from requested layer. When a `GET layer` request *r* is received, Dedup-engine first calculates a set of files in layer *r.layer* that have not been pulled by user *r.addr* by calculating the difference F_Δ between the layer's file set and the user's downloaded file set:

$$F_\Delta = LFmap[r.layer] - UFmap[r.addr] \tag{2}$$

where, *LFmap* maps a layer to its containing *file set* while *UFmap* stores for each user, the files the user has downloaded. DupHunter keeps track of layer metadata and user access patterns in *LFmap* and *UFmap* to calculate F_Δ.

Recall from §4.2.4 that DupHunter preconstructs/prefetches layers when a `GET manifest` request *r* is received. If layer *r.layer* has not been pulled by user *r.addr*, i.e., $r.layer \subset S_\Delta$ (as shown in Equation 1), which means that the files in F_Δ have not been downloaded by the user. Then, the files in F_Δ will be put into the *deduplicated* layer.

Next, for file in the layer that has already been stored by the user, i.e., $file \subset F_\cap$ ($F_\cap = LFmap[r.layer] \cap UFmap[r.addr]$), we replace it with a *reference* to the copy of the file existing in the layer pulled earlier by the user *r.addr*, and put the reference into the *deduplicated* layer. Finally, the *deduplicated* layer is compressed and sent to user *r.addr*.

4.3.2 Integrating with different deduplication modes. When deduplication offloading is integrated with B-mode 0, the overhead of layer construction can be significantly reduced, as only a small deduplicated layer needs to be created.

For the remaining deduplication modes (B-mode 1-3 and S-mode), they maintain at least one intact layer replica, which enables them to directly send the layer replica to the users without any layer restoring overhead. However, when deduplication offloading is combined with these four deduplication modes, it results in a certain amount of overhead in constructing the deduplicated layer. This is because DupHunter registry first needs to construct a deduplicated layer and then send it to the users. The process of constructing a deduplicated layer involves decompressing the requested layer, packing the necessary files, and creating the deduplicated layer. To minimize the overhead in constructing deduplicated layers, our system, DupHunter, proactively preconstructs deduplicated layers before receiving a `GET` layer request. Additionally, we utilize an in-memory buffer to store the decompressed layers, thereby avoiding disk I/O.

4.3.3 Shared file remapping. To ensure file operation correctness of the container storage system on the client side, DupHunter client utilizes an efficient shared file remapping mechanism to resolve the references in a deduplicated layer and redirect the file accesses correspondingly. During deduplicated layer unpacking, references are resolved and recorded in a shared file mapping table as shown in Figure 11. The shared file mapping table

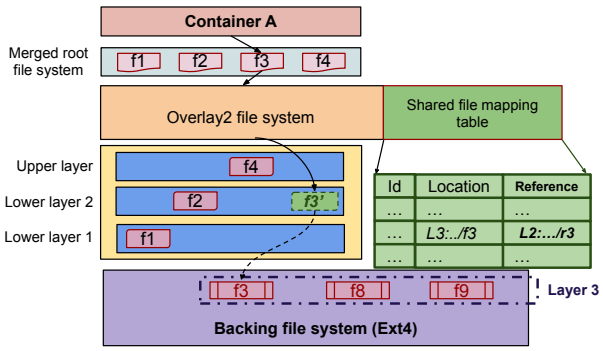


Fig. 11. File remapping.

stores each unique file's Id (fingerprint), location, and all the references to it. For instance, reference *r3* in the deduplicated layer *L2* is resolved in the file path of *f3* in layer *L3*.

In Linux file systems, files are identified and described by an index node, referred to inode, that stores the file attributes and disk block locations of the file's data. And directories are like tables that contain one entry per file (i.e., dentry), containing the file name and the pointer to the inode of that file. Therefore, before the instantiation of a container, we create dentries that point to the inodes of associated target files in the corresponding layer directories for the references.

Figure 11 shows an example of file remapping. Before the initiation of the storage driver for container *A*, we insert a dentry *f3'* in layer directory *L2* that points to the inode of file *f3* in the backing file system. This way *f3'* points to the same data blocks as *f3* as shown in Figure 11. Consequently, when an application in container *A* accesses *f3* in the merged root file system, the request will be directed to *f3'* in layer *L2* and follows to its data blocks in the backing file system.

4.4 Container-aware storage system at client tier

To further reduce the amount of redundant data and eliminate the CoW overhead caused by container storage drivers, we change the CoW mechanism in the container storage driver and use a container-aware storage system to optimize the container data layout on disk.

4.4.1 Container-aware preallocation. To maintain good locality and keep data for each container in close proximity, we utilize a container-aware allocator that is integrated with the backing file system (i.e., *Ext4*) to reserve space during image unpacking, before starting each container.

Once the manifest of an image is downloaded, the Dedup-client discloses the mapping of layers to images in hints to the backing file system so that the backing file system can use this information to exploit data locality. Disclosure hints describe the container image name and size, its containing layer names and sizes, and the upper layer name. Here, a layer name indicates the name of the path that stores the corresponding layer content. The disclosing hints are issued through an I/O-control (`ioctl`) system call in our implementation.

When unpacking an image, we allocate additional space beyond the requested amount to accommodate new files, modifications to existing files, and other potential changes once the container is launched. This extra space will be reclaimed when the associated images are deleted, ensuring efficient use of resources. The size of the preallocated space is related to the uncompressed image size indicated in the disclosure hints, with larger images receiving a correspondingly larger additional allocation. To minimize fragmentation, an in-memory preallocation range is maintained for each container. However, it is essential to note that users may launch multiple containers from a single image, which could result in the reserved space being insufficient to hold all

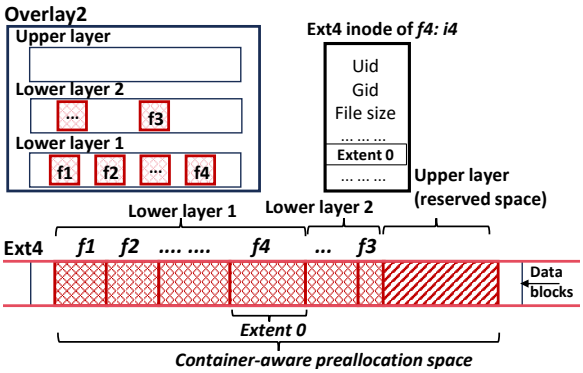


Fig. 12. Container-aware preallocation.

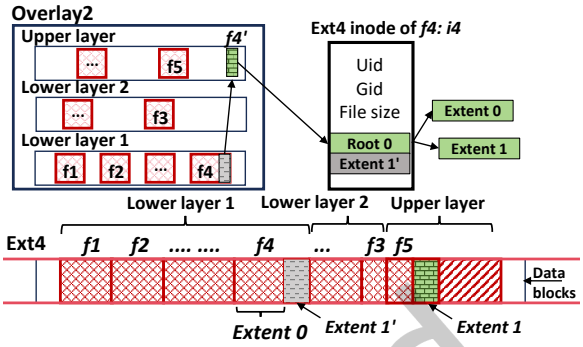


Fig. 13. Container-aware redirect on write.

new files or modifications. In such cases, we rely on the vanilla Ext4 allocator to allocate additional space for the new data.

Figure 12 provides an example of container-aware preallocation for container *A*, wherein files within each layer are positioned closer together on the disk during unpacking. Since layers are immutable, each file contains only one single extent. Additionally, all layers belonging to an image are also positioned in close proximity to one another, with a reserved space allocated for the upper layer of the container. After the container is started, data written to the upper layer are stored within the preallocated reserved space.

4.4.2 Container-aware redirect-on-write. To reduce the overhead of CoWs, we use a container-aware redirect-on-write (ROW) technique that redirects the updates to the read-only files in the lower layers and stores the updates nearby instead of copying the whole files up into the upper layer.

The existing CoW mechanism is implemented in the storage driver. For example, in Overlay2, the CoW mechanism is injected in the Overlay2 open function. If a file in a lower layer is opened with a write-mode flag, the whole file will be copied up into the upper layer no matter whether the file would be updated or not. Instead, we inject our container-aware ROW mechanism in the backing file system because the container data is actually stored and managed by the backing file system.

Figure 13 illustrates an example of the container-aware ROW. The backing file system Ext4 is divided into an array of data blocks. The rewrites to a data block of file *f4* in layer *L1* are redirected to the reserved space for the upper layer. This approach ensures that all data belonging to a container is stored in close proximity. After that, *f4* in Layer *L1* is split into three extents: *Extent 0*, an old *Extent 1'*, and a new *Extent 1* that stores the rewrites to *Extent 1'*. The three extents are all included in the inode *i4* as shown in Figure 13. *Extent 0* and *Extent 1* constitute a new file that is stored in the upper layer while *Extent 0* and *Extent 1'* form a read-only file in the lower layer *L1*. Moreover, to speed up file data lookup, the extents are arranged as trees if the number of extents exceeds a tunable threshold as shown in Figure 13.

Moreover, a new Overlay2 file (i.e., Overlay2 inode) in the upper layer is created. As shown in Figure 13, a new Overlay2 file *f4'* is created and is associated with Ext4 file inode *i4*. During container-aware ROW, the valid and invalid extent information is recorded in the Ext4 file inode, allowing it to serve read requests from the container correctly. Note that we need to keep *Extent 1'* and its reference in the inode *i4*. This is because layers are shared among images and containers, therefore other images or containers may access the old extents.

Write request processing in storage driver. The CoW mechanism in the storage driver is changed accordingly. Writes to a read-only file in a lower layer will not trigger the CoW mechanism. Instead, it will invoke the backing file system's container-aware ROW functions.

The storage driver, such as Overlay2, utilizes virtual inodes known as Overlay2 inodes to intercept the operations on a backing file system like Ext4. An Overlay2 inode contains overlay-specific metadata and is associated with the actual file's inode in the backing file system. In the Overlay2 open function, we disable the Copy-on-Write (CoW) mechanism. When there are write requests to a read-only file in a lower layer, Overlay2 invokes the backing file system's container-aware ROW functions and creates a new Overlay2 inode in the upper layer. Note that our modifications to the backing file system do not impact the behavior of read/write requests from non-containerized applications, as these applications do not invoke the container-aware CoW functions.

5 IMPLEMENTATION

Our implementation of DupHunter builds upon the Docker registry and container storage system on the Docker client side, with a focus on the implementation of the three main components: the Dedup-engine, deduplication offloading, and the container-aware storage system. The Dedup-engine is implemented in Go by adding ~2,000 lines of code to Bolt [48]. Note that Bolt is based on the Docker registry [18] for high availability and scalability (see Bolt details in §2.1.) Deduplication offloading is implemented on both Moby [62] and the Dedup-engine. The container-aware storage system is implemented by modifying both Linux's Ext4 file system [44] and Overlay2 file system [23].

Dedup-engine implementation. DupHunter Dedup-engine can use any POSIX file system to store its data and uses Redis [64] for metadata, i.e., slice and layer recipes, file and layer indices, and ULmap and ILmap. We chose Redis because it provides high lookup and update performance and it is widely used in production systems. Another benefit of Redis is that it comes with a Go client library, which makes it easy to integrate with the Docker Registry. We enable *append-only file* in Redis to log all changes for durability purposes. Moreover, we configure Redis to save snapshots every few minutes for additional reliability. To improve availability and scalability, we use 3-way replication. In our setup, Redis is deployed on all nodes of the cluster (P-servers and D-servers) so that a dedicated metadata database cluster is not needed. However, it is also possible to setup Dedup-engine with a dedicated metadata database cluster.

To ensure that the metadata is in a consistent state, Dedup-engine uses Redis' atomicity so that no file duplicates are stored in the cluster. For the file and layer indices and the slice and layer recipes, each key can be set to hold its value only if the key does not yet exist in Redis (i.e., using `SETNX` [65]). When a key already holds a value, a file duplicate or layer duplicate is identified and removed from the registry cluster.

Additionally, Dedup-engine maintains a synchronization map to ensure that multiple layer restoring processes do not attempt to restore the same layer simultaneously. If a layer is currently being restored, subsequent `GET` layer requests to this layer wait until the layer is restored. Other layers, however, can be constructed in parallel.

Both the metadata database and layer store used by Dedup-engine are scalable and can handle large image datasets. Dedup-engine's metadata overhead is about 0.6% in practice, e.g., for a real-world layer dataset of 18 GB, Dedup-engine stores less than 100 MB of metadata in Redis.

Deduplication offloading implementation. Deduplication offloading is implemented by modifying both the client side and the registry side. Since the registry is unaware of users' layer operations except layer pulling/pushing, such as deleting layer in image deletion operations (i.e., `Docker rmi`). Hence, to accurately record the client-side layer set state, DupHunter needs to retrieve client-side layer set changes. To do this, we modify the Docker client module in Moby so that it sends the layer set changes to Dedup-engine whenever the layer set changes including container instantiation. Then Dedup-engine will update the corresponding *UFmap* after the changes are received.

Table 2. Workload parameters.

Trace	#GET Layer	#GET Manifest	#PUT Layer	#PUT Manifest	#Uniq. Layer	#Accessed Uniq. Dataset Size (GB)
Dal	6963	7561	453	23	1870	18
Fra	4117	10350	508	25	1012	9
Lon	2570	11808	582	40	1979	13
Syd	3382	11150	453	15	558	5

Container-aware storage system implementation. DupHunter's container storage system uses Overlay2 as the storage driver and Ext4 as the backing file system in our implementation. The container-aware RoW is implemented by modifying the allocation strategy in Ext4. In the future, we plan to implement the container-aware storage system based on other backing file systems, such as XFS, Btrfs, etc.

6 EVALUATION

Our evaluation aims to answer two primary questions: first, how effective is our end-to-end deduplication scheme; and second, how does this scheme affect the performance of both the Docker registry's layer pulling process and the container storage system.

6.1 Evaluation Setup

Testbed. Our testbed is composed of two clusters: the DupHunter registry cluster and the DupHunter client cluster. The DupHunter registry cluster has six nodes, each equipped with 8 cores, 16 GB of RAM, a 500 GB SSD, and a 10 Gbps NIC. The DupHunter client cluster has ten nodes, each with 8 cores, 32 GB of RAM, a 500 GB SSD, a 2 TB HDD, and a 10 Gbps NIC.

Dataset. We downloaded 0.93 TB of popular Docker images (i.e., images with a pull count greater than 100) with 36,000 compressed layers, totaling 2 TB after decompression. Such dataset size allowed us to quickly evaluate Dedup-engine's different modes without losing the generality of results. The file-level deduplication ratio of the decompressed dataset is 2.1.

Workload generation. To evaluate how DupHunter registry performs with production registry workloads, we use the IBM Cloud Registry traces [6] that come from four production registry clusters (`Dal`, `Fra`, `Lon`, and `Syd`) and span approximately 80 days. We use Docker registry trace player [6] to replay each workload as shown in Table 2. Note that the accessed unique dataset size in Table 2 is calculated based on the requested layer size in each workload. As the traces do not contain specific image data and the trace player only generates random layers, we modify the player to match requested layers in the IBM trace with real layers downloaded from Docker Hub based on the layer size.Consequently, each layer request pulls or pushes a real layer. For manifest requests, we generate random well-formed, manifest files.

In addition, our workload generator uses a proxy emulator to decide the server for each request. The proxy emulator uses consistent hashing [42] to distribute layers and manifests. It maintains a ring of registry servers and calculates a destination registry server for each `push` layer or manifest request by hashing its digest. For `pull` manifest requests, the proxy emulator maintains two consistent hashing rings, one for the P-servers, and another for the D-servers in DupHunter registry cluster. By default, the proxy first queries the P-servers but if the requested P-server is not available, it pulls from the D-servers.

Schemes. We evaluate DupHunter registry's deduplication ratio and `GET` layer (i.e., complete layer) performance using different deduplication and redundancy schemes in §6.2. The base case considers 3-way layer replication and file-level deduplication. In that case, DupHunter registry provides five deduplication modes: B-mode 0, 1,

Table 3. Dedup. ratio vs. GET layer latency.

Mode	Dedup. ratio	GET layer performance improvement
B-mode 1	1.5	1.6×
S-mode	1.3	2×
B-mode 2	1.2	2.6×
B-mode 3	1	2.8×
B-mode 0	Dedup ratio	GET layer performance degradation
	GF-R (Global file-level [3 replicas])	
	2.1	-1.03×
	GF+LB-R (Global file- and local block-level [3 replicas])	
	3.0	-2.87×
	GB-EC (Global block-level [Erasure coding])	
	6.9	-6.37×

2, 3, and S-mode. Note that B-mode 0 deduplicates all layer replicas (denoted as global file-level deduplication with replication or GF-R) while B-mode 3 does not deduplicate any layer replicas. To evaluate how DupHunter registry works with block-level deduplication, we integrate B-mode 0 with VDO. For each D-server, all unique files are stored on a local VDO device. Hence, in that mode DupHunter provides global file-level deduplication and local block-level deduplication (GF+LB-R). We also evaluate DupHunter registry with an erasure coding policy instead of replication. We combine B-mode 0 with Ceph such that each D-server stores unique files on a Ceph erasure coding pool with global block-level deduplication enabled. We denote this scheme as GB-EC. We compare each scheme to Bolt [48] with 3-way replication as our baseline (No-dedup).

Second, we evaluated the container deployment time and running performance of DupHunter by enabling deduplication offloading and compared it with Gear [22]. To show the impact of deduplication offloading and on-demand remote image technique on container deployment and running performance, we disabled our container-aware preallocation and ROW techniques in this experiment.

Finally, we measured the impact of DupHunter's container-aware storage system on container I/O performance by enabling container-aware preallocation and ROW techniques, and compared it with both Overlay2 [23] and Baoverlay [73] by using HDDs and SSDs as the storage devices respectively.

6.2 Deduplication Ratio vs. layer pulling performance

We first evaluate DupHunter registry's performance/deduplication ratio trade-off for all of the above described deduplication schemes. For the replication scenarios, we use 3-way replication and for GB-EC, we use a (6,2) Reed Solomon code [61, 66]. Both replication and erasure coding policies can sustain the loss of two nodes. We use 300 clients spread across 10 nodes and measure the average GET layer latency across the four production workloads. Table 3 shows the results normalized to the baseline.

We see that all four performance modes of DupHunter registry (B-mode 1, 2, and 3, and S-mode) have better GET layer performance compared to No-dedup. B-mode 1 and 3 reduce the GET layer latency by 1.6× and 2.8×, respectively. This is because the prefetch cache hit ratio on P-servers is 0.98 and a high cache hit ratio significantly reduces disk accesses. B-mode 3 has the highest GET layer performance but does not provide any space savings since each layer in B-mode 3 has three full replicas. B-mode 1 and 2 maintain only one and two layer replicas for each layer, respectively. Hence, B-mode 1 has a lower performance improvement (i.e., 1.6×) than B-mode 2 (i.e., 2.6×), but has a higher deduplication ratio of 1.5×. S-mode lies between B-mode 1 and 2 in terms of the deduplication ratio and performance. This is because, in S-mode, popular layers have three layer replicas while cold layers only have a single replica.

Compared to the above four modes, B-mode 0 has the highest deduplication ratio because *all* layer replicas are deduplicated. Consequently, B-mode 0 adds overhead to GET layer requests compared to the baseline performance.

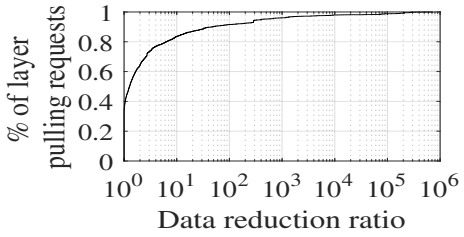


Fig. 14. Data reduction ratio distribution (X-axis is log-scale).

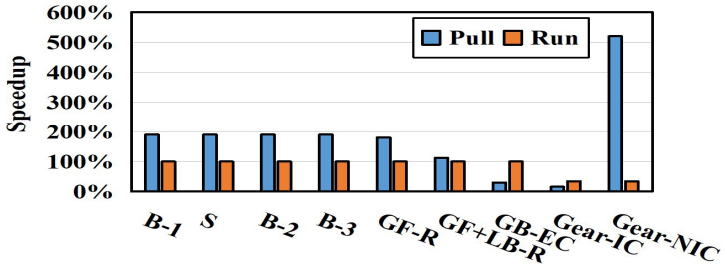


Fig. 15. Deployment improvements.

As shown in Table 3, if file-level deduplication and 3-way replication are used, the deduplication ratio of B-mode 0 is 2.1 while the GET layer performance is 1.03× slower.

If block-level deduplication and block-level compression are used (GF+LB-R), the deduplication ratio increases to 3.0 while the GET layer performance decreases to 2.87×. This is because of the additional overhead added by restoring the layer's files prior to restoring the actual layer. Compared to replication, erasure coding naturally reduces storage space. The deduplication ratio with erasure coding and block-level deduplication is the highest (i.e., 6.9). However, the GET layer performance decreases by 6.37× because to restore a layer, its containing files, which are split into data chunks and spread across different nodes, must first be restored.

Overall, DupHunter registry, even in B-mode 0, significantly decreases the layer restoring overhead compared to the naive approaches shown in Table 1 in §3.3. For example, DupHunter registry's B-mode 0 with VDO (the GF+LB-R scheme) has a GET layer latency only 2.87× slower than the baseline compared to a the VDO-only scheme which is 60× slower compared to the baseline. In addition, the DupHunter registry currently provides five deduplication modes. But we believe that DupHunter can provide more modes by combining two redundant policies together. For example, we decompose S-mode by using two redundant policies: popular layers using 3-way replication, and cold layers using erasure coding. In this case, the DupHunter registry can achieve a deduplication ratio of 3.3× and a 2.1× performance degradation. In the future, we plan to implement various deduplication modes for different uses.

6.3 The impact of deduplication offloading

To understand how deduplication offloading accelerates container deployment, we first show the traffic reduction ratio during layer pulling by enabling deduplication offloading. Then we compare the container deployment time and running performance improvements for both Gear [22] and DupHunter compared to Docker [16]. Furthermore, we evaluated the overhead of data deduplication to explain the necessity of offloading deduplication from the client side to the registry side.

6.3.1 Layer pulling speedup.

Network traffic reduction. Figure 14 shows the ratio of the volume of transferred data during layer pulling operations for Bolt [48] to that of DupHunter, which is called *data reduction ratio*. DupHunter significantly reduces network traffic for layer pulling operations. On average, DupHunter reduces the amount of data transferred during layer pulling operations by 3,440×. The majority of layers that are transmitted are effectively reduced in size. For example, the sizes of 65% of layers transferred during layer pulling operations are reduced more than 2×, respectively. The maximum data reduction reaches 734,530×. Overall, the amount of data transferred can be significantly reduced by using deduplication offloading.

Deployment improvements. To measure the deployment improvements by using deduplication offloading, we experimented using the following setup: We utilize the workload in [22], which consists of two parts: (1) a collection of the top 50 popular official image series, totaling 0.1 TB, and (2) a test harness for executing the simplest tasks in the container. The dataset has a file-level deduplication ratio of 2×. We use a single client for both Gear [22] and DupHunter. During the experiment, the client executes pull and run commands separately so we can measure the average time taken for the pull and run phases independently.

Figure 15 shows the pull and run speedup for both DupHunter and Gear [22] relative to Docker. As shown, the first three schemes are DupHunter deduplication modes, i.e., B-mode 1, S-mode, B-mode 2-3, and B-mode 0s (GF-R, GF+LB-R, and GB-EC). The rest two schemes – Gear-IC and Gear-NIC belong to Gear. We evaluated Gear [22] in two scenarios: If the requested image has not been converted into a Gear image, the image has to be first converted into a Gear image and then pulled by the Gear client, which is called Gear with image format conversion (denoted as Gear-IC as shown in Figure 15); Otherwise, Gear clients can directly pull the Gear image, which is an ideal scenario for Gear and is referred to as Gear without image format conversion (denoted as Gear-NIC as shown in Figure 15). The reason for evaluating Gear-IC is the infeasibility of converting all Docker images to Gear images in advance, especially with the continuous increase of Docker Hub images.

As shown in Figure 15, Gear without image format conversion can significantly shorten the pulling time but consumes more time in the run phase. For example, Gear without image format conversion accelerates pulling performance to 5.2×. But the run time for the two Gear schemes increases 2.9× on average. This is because Gear uses an on-demand remote image technique that only downloads a smaller amount of required files that cannot be found locally on demand, significantly accelerating pulling operations. However, Gear still needs to download the required files on demand during the run phase, which takes extra time and slows down running performance. The total amount of data transferred for Gear is 0.02 TB.

The pulling time of Gear with image format conversion increases 6.2×. This is because Gear image format conversion consumes a lot of time (detailed in §2.3), resulting in a longer container deployment time.

The maximum pulling speedup for DupHunter is 1.9× (i.e., B-mode 1-3 and S-mode). The pulling time of the rest 3 deduplication modes is longer due to deduplicated layer construction overhead. GB-EC has the longest pulling time but it provides the highest deduplication ratio. Its deduplication ratio reaches 6.9×, respectively while Gear only saves half of the storage space [22]. Furthermore, DupHunter provides equivalent running performance with Docker since DupHunter does not fetch files from the registry during the run phase.

Note that deduplication offloading involves constructing deduplicated layers for users. This reduces the deduplication overhead on the client side but also leads to an overhead of constructing the deduplicated layer on the registry side. When the deduplication offloading is integrated with deduplication mode B-mode 0 (GF-R, GF+LB-R, and GB-EC), the layer construction overhead is considerably reduced as only a small deduplicated layer needs to be constructed. In the experiment, deduplication offloading reduces the total amount of data being constructed by a significant margin, resulting in a transfer of approximately 0.05 TB of data, up to a 2× reduction compared to Docker registry. Additionally, the preconstruct cache can further improve the pulling speed. The preconstruction cache hit ratio for GF-R, GF+LB-R, and GB-EC reaches 0.95, 0.9, and 0.8. We see that the pulling speedup for GF-R and GF+LB-R is 1.8× and 1.12× respectively while pulling time increases by 3.3× for GB-EC as shown in Figure 15.

When deduplication offloading is integrated with the rest deduplication modes (B-mode 1-3 and S-mode), it also introduces an additional deduplicated layer construction overhead. Without deduplication offloading, these four deduplication modes maintain at least one intact layer replica, allowing them to directly deliver the layer replica to the users without any layer construction overhead. However, with deduplication offloading enabled, the Duphunter registry first needs to construct a deduplicated layer before sending it to users. This involves decompressing the requested layer, packing the necessary files, and creating the deduplicated layer. Nevertheless, the deduplicated layer construction overhead can be mitigated by reducing the amount of transferred data and

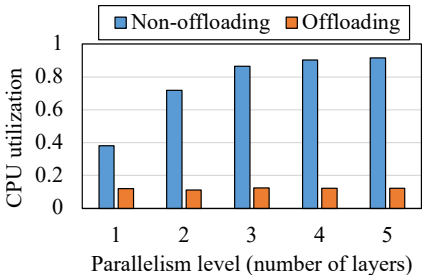


Fig. 16. CPU overhead on the client side.

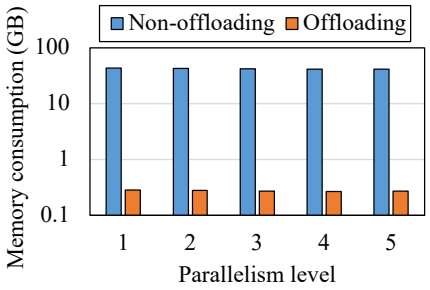


Fig. 17. Memory consumption on the client side.

layer preconstruction. The preconstruction cache hit ratio for B-mode 1-3 and S-mode reaches 0.98. The high preconstruction cache hit ratio is achieved because the Duphunter registry has prior knowledge of the layer dataset on the client side, which allows it to preconstruct the requested deduplicated layers before it receives the GET layer request.

6.3.2 CPU and memory overhead due to data deduplication. To justify the necessity of deduplication offloading, we compared CPU and memory overhead for two schemes: (1) Deduplication offloading scheme: data deduplication is offloaded from the client side to the registry side; (2) Non-deduplication offloading scheme: data deduplication is conducted on the client side. In this case, the client performs layer deduplication on its local layer dataset independently to remove the redundant files stored in the layer dataset on the client side. The client iterates over the layers and performs the following steps: 1) Decompress the layer into individual files; 2) Compute a fingerprint for each file in the layer; 3) Comparing the fingerprints of all files with the file index to determine if identical files are already stored locally; 4) Store the unique files locally; 5) Update the metadata related to deduplication; 6) Remove the layer from the client. Note that the main overhead in the deduplication process arises from decompression and file digest calculation, which can result in high CPU and memory usage. We measure the CPU (Figure 16) and memory utilization (Figure 17) on the client side.

CPU consumption. To analyze the CPU overhead because of data deduplication, we measure the average CPU utilization on the Docker client sides for both the deduplication offloading scheme (denoted as Offloading in Figure 16) and the Non-deduplication offloading scheme (denoted as Non-offloading in Figure 16), respectively. As shown in Figure 16, Non-deduplication offloading scheme introduces considerable computation overhead to the clients. For example, the average CPU utilization increases from 38% to 92% as the concurrent layer pulling threads increase from 1 to 5. This is because decompression and file digest calculation require a high CPU utilization. The CPU utilization for the deduplication offloading scheme remains stable and low (around 12%) as the number of concurrent layer pulling threads increases. This is because the deduplication offloading scheme offloads the data deduplication from the Docker client side to the Docker registry side, resulting in negligible CPU overhead on the client side.

Memory consumption. To understand the memory overhead due to data deduplication, we measure the memory consumption on the Docker client side for both the deduplication offloading scheme and the Non-deduplication offloading scheme, respectively. As shown in Figure 17, Non-deduplication offloading scheme consumes significantly more memory than the deduplication offloading scheme. For example, Non-deduplication offloading scheme consumes 41.4 GB–43.3 GB memory totally as the concurrent layer pulling threads increase from 1 to 5. This suggests that the data deduplication process is a CPU-memory intensive workload. While for deduplication

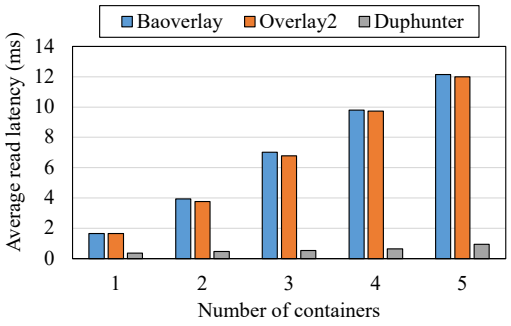


Fig. 18. Read performance for HDD.

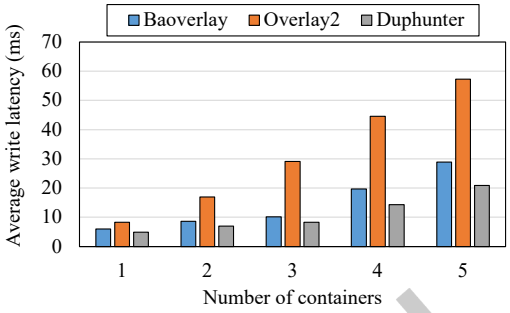


Fig. 19. Write performance for HDD.

offloading scheme, the total memory consumption remains stable at 0.28 GB as the concurrent layer pulling threads increase. This is because the data deduplication is offloaded to DupHunter registry servers. Consequently, the impact of data deduplication on the clients is negligible.

To sum up, applying deduplication on the client side can introduce significant overhead and consequently slow down container runtime performance. On the other hand, deduplication offloading technique does not bring overhead to the client side. In our implementation, the registry stores the layers in staging areas once a layer is pushed by a client, and performs offline distributed layer deduplication, which ensures that image distribution performance is unaffected.

6.4 The effectiveness of container-aware storage system

To understand how DupHunter's container-aware storage system improves containers' I/O performance, we compared it with Baoverlay and vanilla Overlay2. Specifically, we first launch multiple `fio` containers [25] and create a new layer containing 1,000 16 MB files. After that, we commit the containers as new images and run the newly created images as container instances. We then randomly rewrite 4KB of data to each of the first 500 files in the preceding read-only layer for each container, and measure the write performance. The read performance is measured by randomly reading 4KB of data from the 1,000 files. Moreover, we conducted experiments both on HDDs and SSDs to show how DupHunter's container-aware storage system performs on different storage devices.

When an HDD is used as the storage device, the container-aware storage system of DupHunter can significantly improve read and write performance, as shown in Figure 18 and Figure 19. For read operations, DupHunter shows 75% and 93% performance improvement for 1 container and 5 containers compared to Overlay2 and Baoverlay, respectively. This is because the container-aware storage system of DupHunter collocates data for each container to improve data locality, which largely reduces disk seek time for HDDs. For write operations, DupHunter brings 40%–64% performance improvement as the number of containers increases from 1 to 5 compared to Overlay2. This is because DupHunter only writes a small amount of new data by using RoWs while Overlay2 copies the whole files.

We observe that read performance improvement is much higher than write performance improvements. The reason is that RoW operations need to modify metadata in both Overlay2 and Ext4 file systems, which may cause competition. Compared to Baoverlay, DupHunter shows 17%–28% write performance improvements as the number of containers increases from 1 to 5. This is because the container-aware storage system of DupHunter can reduce disk seek time for HDDs by colocating data.

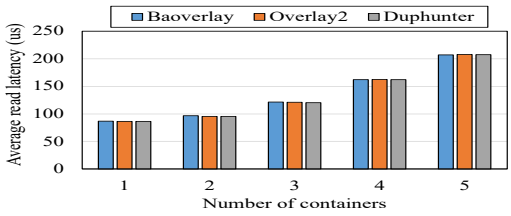


Fig. 20. Read performance for SSD.

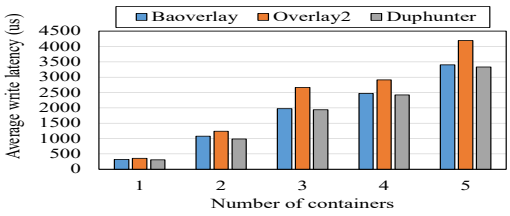


Fig. 21. Write performance for SSD.

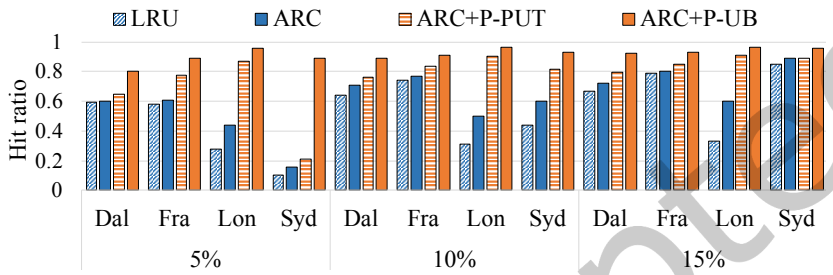


Fig. 22. Cache hit ratio on P-servers with different cache algorithms.

Unlike HDDs, SSDs have no moving parts, so disk seek time is not an issue for SSDs. When an SSD is used as the storage device, we observe that DupHunter, Overlay2, and Baoverlay2 show similar read performance as shown in Figure 20. For write performance, both DupHunter and Baoverlay show 10%-20% performance improvement compared to Overlay2 as the number of containers increases from 1 to 5 as shown in Figure 21. This is because both DupHunter and Baoverlay only write the new data instead of copying the whole files.

In summary, the container-aware storage system of DupHunter exceeds both Overlay2 and Baoverlay for HDD-based storage systems. When deploying on SSDs, container-aware storage system performs similarly to Baoverlay and has better write performance than Overlay2.

6.5 Dedup-engine cache Effectiveness

Next, we analyze Dedup-engine's caching behavior. We first study the prefetch cache and then the preconstruct cache.

6.5.1 Prefetch cache. To understand how the prefetch cache improves the P-servers' performance, we first show its hit ratio compared to two popular cache algorithms: LRU [58] and ARC [51]. Moreover, we compare DupHunter's prefetch cache with another prefetch algorithm, which makes predictions based on PUT requests [6] (denotes as ARC+P-PUT). Both of these algorithms are implemented on ARC since ARC outperforms LRU. DupHunter's prefetch algorithm, based on user behavior (UB), is denoted as ARC+P-UB. We vary the cache sizes from 5% to 15% of each workload's unique dataset size. Figure 22 shows the results for the four production workloads (Dal, Syd, Lon, and Fra).

For a cache size of 5%, the hit ratios of LRU are only 0.59, 0.58, 0.27, and 0.10, respectively. ARC hit ratios are higher compared to LRU (e.g., 1.6× Lon) because after a user pulls a layer, the user is not likely to repull this layer in the future as it is locally available. Compared to LRU, ARC maintains two lists, an LRU list and an LFU list, and adaptively balances them to increase the hit ratio.

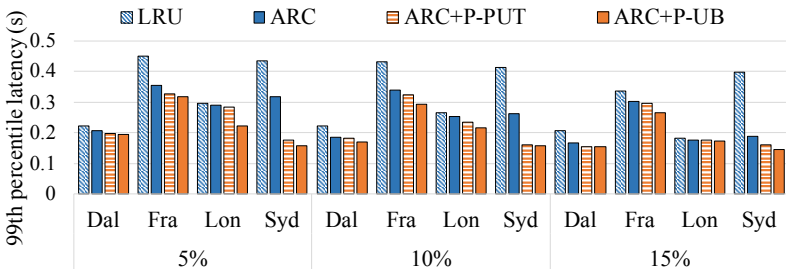


Fig. 23. 99^{th} percentile GET layer latency of P-servers.

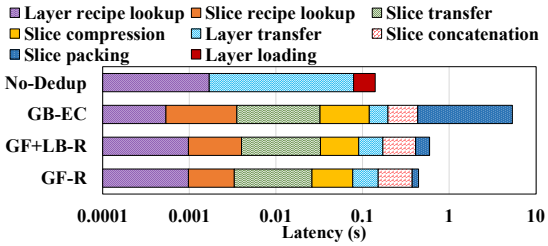


Fig. 24. Layer restoring latency breakdown (X-axis is log-scale).

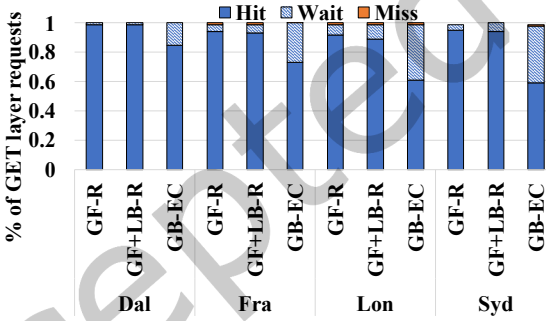


Fig. 25. Preconstruct cache hit ratio.

ARC+P-PUT improves the ARC hit ratio by 1.9× for Lon. However, ARC+P-PUT only slightly improves the hit ratio for the other workloads. This is because ARC+P-PUT acts like a write cache which temporally holds recently uploaded layers and waits for the clients that have not yet pulled these layers to issue GET requests. This is not practical because the layer reuse time (i.e., interval between a PUT layer request and its subsequent GET layer request) is long. For example, the reuse time is 0.5 hr for Dal on average based on our observation. Moreover, ARC+P-PUT ignores the fact that some clients always repull layers. Dedup-engine's ARC+P-UB achieves the highest hit ratio. For example, ARC+P-UB's hit ratio for Dal is 0.89, resulting in a 4.2× improvement compared to ARC+P-PUT.

As shown in Figure 22, the hit ratio increases as the cache size increases. For example, when cache size increases from 5% to 15%, the hit ratio for ARC under workload Lon increases from 0.44 to 0.6. ARC+P-UB achieves the highest hit ratio of 0.96 for a cache size of 15% under workload Lon. Overall, this shows that by exploiting user behavior ARC+P-UB can achieve high hit ratios, even for smaller cache sizes.

Figure 23 shows the 99^{th} percentile of GET request latencies for P-servers with different cache algorithms. The GET layer latency decreases with higher hit ratios. For example, when the cache size increases from 5% to 15%, the 99^{th} percentile latencies decrease from 0.19 s to 0.15 s for DupHunter's ARC+P-UB under workload Dal and the cache hit ratio increases from 0.8 to 0.92. Moreover, when the cache size is only 5%, ARC+P-UB significantly outperforms the other 3 caching algorithms. For example, ARC+P-UB reduces latency by 1.4 × compared to LRU for workload Fra. We also observe that there is an inconsistency between cache hit ratio improvements and tail latency reduction because of the uneven distribution of layer sizes. When the cache size increases, caching more big-sized layers will result in a significant reduction in tail latency. Otherwise, it will lead to a moderate reduction in tail latency.

6.5.2 Preconstruct cache. For the preconstruct cache to be effective, layer (i.e., complete layer) restoring must be fast enough to complete within the time window between the GET manifest and GET layer request.

Layer restoring performance. To understand the layer restoring overhead, we disable the preconstruct cache and measure the average GET layer latency when a layer needs to be restored on D-servers. We evaluate GB-EC, GB+LB-R, and GF-R and compare it to No-dedup.

We break down the average reconstruction latency into its individual steps. The steps in layer reconstruction include looking up the layer recipe, fetching and concatenating slices, and transferring the layer. Fetching and concatenating slices in itself involves slice recipe lookup, slice packing, slice compression, and slice transfer. No-dedup contains three steps: layer metadata lookup, layer loading from disk to memory, and layer transfer.

As shown in Figure 24, GF-R has the lowest layer restoring overhead compared to GF+LB-R and GB-EC. It takes, on average, 0.44 s to rebuild a layer tarball for GF-R. Compared to the No-Dedup scheme, the GET layer latency of GF-R increases by 3.1×. Half of the GET layer latency is spent on slice concatenation. This is because slice concatenation involves writing each slice into a compressed tar archive, which is done sequentially. Slice packing and compression are faster, 0.07 s and 0.05 s, respectively, because slices are smaller and evenly distributed on different D-servers.

For the GF+LB-R scheme, it takes 0.55 s to rebuild a layer on average. Compared to GF-R, adding local block-level deduplication increases the overall overhead by up to 1.4× due to more expensive slice packing. It takes 0.18 s to pack a slice into an archive, 2.7× higher than GF-R's slice packing latency as reading files from the local VDO device requires an additional file restoring process.

The GB-EC scheme has the highest layer restoring overhead. The bottleneck is again slice packing which takes 5 s. This is because each file is split into four data chunks, distributed on different D-servers, and deduplicated. To pack a slice, each involved file needs to be reconstructed from different D-servers and then written to a slice archive, which incurs considerable overhead.

Preconstruct cache impact. To understand how the preconstruct cache improves D-servers' GET layer performance, we first show its hit ratio on D-servers with three deduplication schemes (GF-R, GF+LB-R, and GF-EC). The cache size is set to 10% of the unique dataset.

Figure 25 shows the preconstruct cache hit ratio breakdown. **Hit** means the requested layer is present in the cache while **Wait** means the requested layer is in the process of preconstruction and the request needs to wait until the construction process finishes. **Miss** means the requested layer is neither present in the cache nor in the process of preconstruction. As shown in the figure, GF-R has the highest hit ratio, e.g., 0.98 for the Dal workload. Correspondingly, GF-R also has the lowest wait and miss ratios because it has the lowest restoring latency and a majority of the layers can be preconstructed on time.

Note that the miss ratio of the preconstruct cache is slightly lower than that of the prefetch cache across all traces. This is because we use an in-memory buffer to hold the layer archives that are in the process of construction to avoid disk I/O. After preconstruction is done, the layers are flushed to the on-disk preconstruct cache. In this case, many requests can be served directly from the buffer and consequently, layer preconstruction does not immediately trigger cache eviction like layer prefetching. The preconstruct cache eviction is delayed til the layer preconstruction finishes.

GF+LB-R shows a slightly higher wait ratio than GF-R. For example, the wait ratios for GF-R and GF+LB-R are 0.04 and 0.06, respectively under workload Syd. This is because the layer restoring latency of GF+LB-R is slightly higher than GF-R. GB-EC's wait ratio is the highest. Under workload Syd, 39% of GET layer requests are waiting for GB-EC as layers cannot be preconstructed on time.

Figure 26 shows the corresponding average GET layer latencies of D-servers compared to No-dedup. GF-R and GF+LB-R increase the latency by 1.04× and 3.1×, respectively, while GB-EC adds a 5× increase. This is due to GB-EC's high wait ratios.

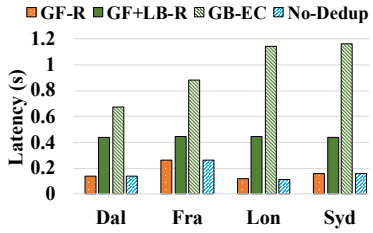


Fig. 26. Performance of D-servers.

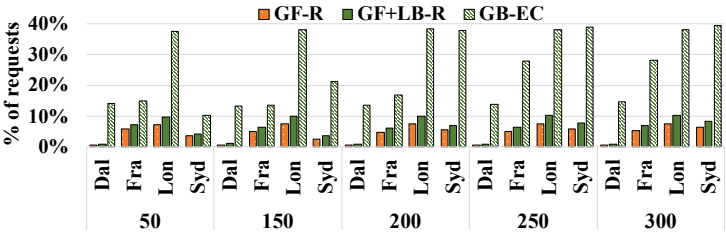


Fig. 27. Request wait ratio with different number of clients.

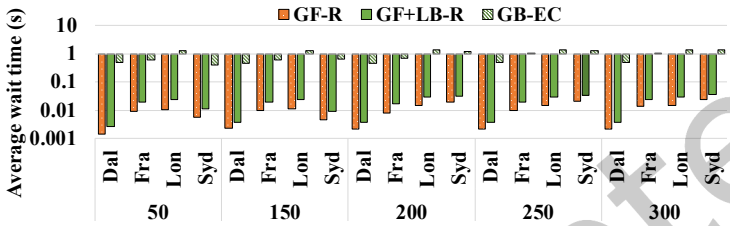


Fig. 28. Average wait time with different number of clients (Y-axis is log-scale).

Scalability. To analyze the scalability of the preconstruct cache under higher load, we increase the number of concurrent clients sending GET layer requests, and measure the request wait ratio (Figure 27) and the average wait time (Figure 28).

Under workload Fra and Syd, the wait ratio for GB-EC increases dramatically with the number of concurrent clients. For example, the wait ratio increases from 15% to 28% as the number of concurrent clients increases from 50 to 300. This is because the layer restore latency for GB-EC is higher and with more concurrent client requests, more requested layers cannot be preconstructed on time. Under workload Lon and Dal, the wait ratio for GB-EC remains stable. This is because the client requests are highly skewed. A small number of clients issue the majority of GET layer requests. Correspondingly, GB-EC also has the highest wait time. Under workload Fra and Syd, the average wait time increases from 0.6 s to 1.1 s and 0.4 s to 1.4 s respectively as the number of clients increases from 50 to 300 for GB-EC.

Although some layers cannot be preconstructed before the GET layer requests arrive, the preconstruct cache can still reduce the overhead because layer construction starts prior to the arrival of the GET requests. This is shown by the fact that the wait times are significantly lower than the layer construction times. For GF-R and GF+LB-R, the average wait times are only 0.001 s and 0.003 s, respectively under workload Dal. When the number of concurrent clients increases, the average wait time of GF-R and GF+LB-R remains low. This means that the majority of layers can be preconstructed on time for both GF-R and GF+LB-R, and the layers that cannot be preconstructed on time do not incur high overhead.

7 CONCLUSION

We presented DupHunter, an end-to-end high performance deduplication scheme that provides high performance deduplication for both Docker registries and container storage systems. To deduplicate layers stored in Docker registries, DupHunter supports multiple configurable deduplication modes to meet different space saving and performance requirements. DupHunter parallelizes layer reconstruction locally and across the Docker registry cluster to further mitigate overheads. By exploiting knowledge of the application domain, DupHunter introduces a novel layer prefetch/pre-construct cache algorithm based on user access patterns. DupHunter's prefetch cache

can improve GET latencies by up to 2.8× while the preconstruct cache can reduce the restore overhead by up to 20.9× compared to the state of the art. Moreover, to deduplicate redundant data from the container storage system on the Docker client side, DupHunter avoids storing redundant files on the container storage system and further speeds up layer pulling by using deduplication offloading. In addition, to eliminate the data duplicates caused by CoW mechanism in the container storage system, DupHunter uses a container-aware storage system that preallocates space for the container and makes sure that files in a container and their modifications are placed and redirected closer on the disk to speedup container I/O performance.

To enable the deduplication offloading scheme, it is required that the registry has knowledge of the client-side layer dataset information. However, this requirement may raise concerns regarding data privacy and security when used on cloud platforms. The deduplication offloading scheme can be used for on-premises infrastructure where one can have complete access to the container clusters that run DupHunter clients and the storage cluster that runs the DupHunter registry. In this scenario, the registries can keep track of the layer dataset on the client side, which leads to a significant improvement in the storage efficiency of Docker images. In the future, our focus will be on developing a secure deduplication offloading scheme that ensures the protection of data privacy.

8 ACKNOWLEDGMENTS

This work is sponsored by the Guangdong Basic and Applied Basic Research Foundation (No. 2021A1515110080), National Science Foundation for Young Scientists of China (No. 62202382), the Chinese National Key Research and Development Program (No. 2022YFB2702101), Shaanxi Key Research and Development Program (No.2021ZDLGY03-02 and No.2021ZDLGY03-08), Major Research Plan of the National Natural Science Foundation of China (No.92152301), National Science Foundation of China for General Program (No.62272394), the grants in BITS Pilani - BBF/BITS(G)/FY2022-23/BCPS-123, GOA/ACG/2022-2023/Oct/11, and BPGC/RIG/2021-22/06-2022/02, and the NSF under the grants: CSR-2106634, CSR-2312785, CCF-1919113, and OAC-2004751.

REFERENCES

[1] Keith Adams and Ole Agesen. 2006. A Comparison of Software and Hardware Techniques for x86 Virtualization. *ACM SIGOPS Operating Systems Review* 40, 5 (2006), 2–13.
[2] Alfred Krohmer. 2023. Proposal: Deduplicated storage and transfer of container images. https://gist.github.com/devkid/5249ea4c88aab4c7bff1b34c955c1980.
[3] Aliyun Open Storage Service (Aliyun OSS). 2023. https://cn.aliyun.com/product/oss?spm=5176.683009.2.4.Wma3SL.
[4] Amazon. 2023. Amazon Elastic Container Registry. https://aws.amazon.com/ecr/.
[5] Amazon. 2023. Containers on AWS. https://aws.amazon.com/containers/services/.
[6] Ali Anwar, Mohamed Mohamed, Vasily Tarasov, Michael Littley, Lukas Rupprecht, Yue Cheng, Nannan Zhao, Dimitrios Skourtis, Amit S. Warke, Heiko Ludwig, Dean Hildebrand, and Ali R. Butt. 2018. Improving Docker Registry Design Based on Production Workload Analysis. In *16th USENIX Conference on File and Storage Technologies (FAST)*.
[7] D. Bonino, F. Corno, and G. Squillero. 2003. Dynamic prediction of Web requests. In *The 2003 Congress on Evolutionary Computation, 2003. CEC '03.*, Vol. 3. 2034–2041 Vol.3. https://doi.org/10.1109/CEC.2003.1299923
[8] Btrfs. 2023. https://btrfs.wiki.kernel.org/index.php/Deduplication.
[9] Richard Shane Canon and Doug Jacobsen. 2016. Shifter: Containers for HPC. In *Cray User Group*.
[10] Zhichao Cao, Hao Wen, Fenggang Wu, and David HC Du. 2018. {ALACC}: Accelerating Restore Performance of Data Deduplication Systems Using Adaptive Look-Ahead Window Assisted Chunk Caching. In *16th {USENIX} Conference on File and Storage Technologies ({FAST} 18)*. 309–324.
[11] Ceph. 2023. https://docs.ceph.com/docs/master/dev/deduplication/.
[12] Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C. Gall. [n. d.]. An Empirical Analysis of the Docker Container Ecosystem on GitHub. In *IEEE MSR'17*.
[13] Cloud Native Computing Foundation Projects. 2023. https://www.cncf.io/projects/.
[14] Backup Compression and Deduplication. 2023. https://tinyurl.com/vgvb7wu.
[15] Datadog. 2023. 8 Surprising Facts about Real Docker Adoption. https://www.datadoghq.com/docker-adoption/.
[16] Docker. 2023. https://www.docker.com/.
[17] Docker Hub. 2023. https://hub.docker.com/.

[18] Docker Inc. 2023. Docker Registry. https://github.com/docker/distribution.
[19] Docker Inc. 2023. Docker Registry HTTP API V2. https://github.com/docker/distribution/blob/master/docs/spec/api.md.
[20] DockerSlim. 2023. https://dockersl.im.
[21] Xing Dongshan and Shen Junyi. 2002. A new Markov model for Web access prediction. *Computing in Science & Engineering* 4, 6 (2002), 34–39. https://doi.org/10.1109/MCISE.2002.1046594
[22] Hao Fan, Shengwei Bian, Song Wu, Song Jiang, Shadi Ibrahim, and Hai Jin. 2021. Gear: Enable Efficient Container Storage and Deployment with a New Image Format. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. 115–125. https://doi.org/10.1109/ICDCS51616.2021.00020
[23] Overlay file system. 2023. https://docs.kernel.org/filesystems/overlayfs.html.
[24] filefrag(8) — Linux manual page. 2023. https://man7.org/linux/man-pages/man8/filefrag.8.html.
[25] fio(1) Linux man page. 2023. https://linux.die.net/man/1/fio.
[26] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Fangting Huang, and Qing Liu. 2014. Accelerating Restore and Garbage Collection in Deduplication-based Backup Systems via Exploiting Historical Information. In *USENIX Annual Technical Conference (ATC)*.
[27] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Yucheng Zhang, and Yujuan Tan. 2015. Design Tradeoffs for Data Deduplication Performance in Backup Workloads. In *13th USENIX Conference on File and Storage Technologies (FAST)*.
[28] Yinjin Fu, Hong Jiang, Nong Xiao, Lei Tian, and Fang Liu. 2011. AA-Dedupe: An Application-aware Source Deduplication Approach for Cloud Backup Services in the Personal Computing Environment. In *IEEE International Conference on Cluster Computing (Cluster)*.
[29] GNU Tar. 2023. Basic Tar Format. https://www.gnu.org/software/tar/manual/html_node/Standard.html.
[30] Google. 2023. Google Container Registry. https://cloud.google.com/container-registry/.
[31] Google Inc. 2013. Google Compute Engine. https://cloud.google.com/compute/.
[32] Katharina Gschwind, Constantin Adam, Sastry Duri, Shripad Nadgowda, and Maja Vukovic. 2017. Optimizing Service Delivery with Minimal Runtimes. In *International Conference on Service-Oriented Computing (ICSOC)*.
[33] Fan Guo, Yongkun Li, Min Lv, Yinlong Xu, and John CS Lui. 2019. HP-mapper: A high performance storage driver for docker containers. In *Proceedings of the ACM Symposium on Cloud Computing*. 325–336.
[34] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Slacker: Fast Distribution with Lazy Docker Containers. In *14th USENIX Conference on File and Storage Technologies (FAST)*.
[35] IBM Cloud Kubernetes Service. 2023. IBM Cloud Kubernetes Service. https://www.ibm.com/cloud/container-service.
[36] IBM Cloud Kubernetes Service. 2023. S3 storage driver. https://docs.docker.com/registry/storage-drivers/s3/.
[37] KR Jayaram, Chunyi Peng, Zhe Zhang, Minkyong Kim, Han Chen, and Hui Lei. 2011. An Empirical Analysis of Similarity in Virtual Machine Images. In *Middleware Industry Track Workshop*.
[38] jdupes. 2023. https://github.com/jbruchon/jdupes.
[39] JFrog Artifcatory. 2023. https://jfrog.com/artifactory/.
[40] Keren Jin and Ethan L Miller. 2009. The Effectiveness of Deduplication on Virtual Machine Disk Images. In *International Systems and Storage Conference (SYSTOR)*.
[41] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing* (El Paso, Texas, USA) *(STOC '97)*. Association for Computing Machinery, New York, NY, USA, 654–663. https://doi.org/10.1145/258533.258660
[42] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *29th Annual ACM Symposium on Theory of Computing (STOC)*.
[43] K. Kumar and M. Kurhekar. 2016. Economically Efficient Virtualization over Cloud Using Docker Containers. In *IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*.
[44] Aneesh Kumar KV, Mingming Cao, Jose R Santos, and Andreas Dilger. 2008. Ext4 block and inode allocator improvements. In *Linux Symposium*, Vol. 1.
[45] Huiba Li, Yifan Yuan, Rui Du, Kai Ma, Lanzheng Liu, and Windsor Hsu. 2020. {DADI}:{Block-Level} Image Service for Agile and Elastic Application Deployment. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 727–740.
[46] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. 2013. Improving Restore Speed for Backup Systems that use Inline Chunk-based Deduplication. In *11th USENIX Conference on File and Storage Technologies (FAST)*.
[47] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. 2009. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *7th USENIX Conference on File and Storage Technologies (FAST)*.
[48] Michael Littley, Ali Anwar, Hannan Fayyaz, Zeshan Fayyaz, Vasily Tarasov, Lukas Rupprecht, Dimitrios Skourtis, Mohamed Mohamed, Heiko Ludwig, Yue Cheng, and Ali R Butt. 2019. Bolt: Towards a Scalable Docker Registry via Hyperconvergence. In *IEEE International Conference on Cloud Computing (CLOUD)*.

[49] Maohua Lu, David Chambliss, Joseph Glider, and Cornel Constantinescu. 2012. Insights for Data Reduction in Primary Storage: A Practical Analysis. In *International Systems and Storage Conference (SYSTOR)*.
[50] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. 2007. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, Vol. 2. Citeseer, 21–33.
[51] Nimrod Megiddo and Dharmendra S Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *2nd USENIX Conference on File and Storage Technologies (FAST)*.
[52] Dirk Meister, Jürgen Kaiser, Andre Brinkmann, Toni Cortes, Michael Kuhn, and Julian Kunkel. 2012. A Study on Data Deduplication in HPC Storage Systems. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*.
[53] Dirk Meister, Jürgen Kaiser, Andre Brinkmann, Toni Cortes, Michael Kuhn, and Julian Kunkel. 2012. A Study on Data Deduplication in HPC Storage Systems. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*.
[54] Microsoft. 2023. Azure Container Registry. https://azure.microsoft.com/en-us/services/container-registry/.
[55] Microsoft Azure. 2023. https://azure.microsoft.com/en-us/.
[56] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. 2001. A Low-bandwidth Network File System. In *ACM SIGOPS Operating Systems Review*, Vol. 35.
[57] M. Oh, S. Park, J. Yoon, S. Kim, K. Lee, S. Weil, H. Y. Yeom, and M. Jung. 2018. Design of Global Data Deduplication for a Scale-Out Distributed Storage System. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. 1063–1073.
[58] Elizabeth J O'Neil, Patrick E O'Neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. *Acm Sigmod Record* 22, 2 (1993), 297–306.
[59] OpenStack Swift storage driver. 2023. OpenStack Swift storage driver. https://docs.docker.com/registry/storage-drivers/swift/.
[60] João Paulo and José Pereira. 2014. A survey and classification of storage deduplication systems. *ACM Computing Surveys (CSUR)* 47, 1 (2014), 11.
[61] James S Plank, Mario Blaum, and James L Hafner. 2013. SD codes: erasure codes designed for how storage systems really fail.. In *FAST*. 95–104.
[62] Moby project. 2023. https://github.com/moby/moby.
[63] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. 2017. Cimplifier: Automatically Debloating Containers. In *11th Joint Meeting on Foundations of Software Engineering (FSE)*.
[64] Redis. 2023. https://redis.io/.
[65] Redis. 2023. SETNX. https://redis.io/commands/setnx.
[66] Irving S Reed and Gustave Solomon. 1960. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics* 8, 2 (1960), 300–304.
[67] Philip Shilane, Ravi Chitloor, and Uday Kiran Jonnala. 2016. 99 Deduplication Problems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*. USENIX Association, Denver, CO. https://www.usenix.org/conference/hotstorage16/workshop-program/presentation/shilane
[68] H. Shim, P. Shilane, and W. Hsu. 2013. Characterization of Incremental Data Changes for Efficient Data Protection. In *USENIX Annual Technical Conference (ATC)*.
[69] Dimitris Skourtis, Lukas Rupprecht, Vasily Tarasov, and Nimrod Megiddo. 2019. Carving Perfect Layers out of Docker Images. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*.
[70] Richard P. Spillane, Wenguang Wang, Luke Lu, Maxime Austruy, Rawlinson Rivera, and Christos Karamanolis. 2016. Exo-clones: Better Container Runtime Image Management Across the Clouds. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*.
[71] Kiran Srinivasan, Timothy Bisson, Garth R Goodson, and Kaladhar Voruganti. 2012. iDedup: latency-aware, inline data deduplication for primary storage. In *10th USENIX Conference on File and Storage Technologies (FAST)*.
[72] Microsoft Azure Storage. 2023. https://azure.microsoft.com/en-us/services/storage/.
[73] Yu Sun, Jiaxin Lei, Seunghee Shin, and Hui Lu. 2020. Baoverlay: a block-accessible overlay file system for fast and efficient container storage. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 90–104.
[74] Zhen Sun, Geoff Kuenning, Sonam Mandal, Philip Shilane, Vasily Tarasov, Nong Xiao, and Erez Zadok. 2016. A Long-Term User-Centric Analysis of Deduplication Patterns. In *32nd International Conference on Massive Storage Systems and Technology (MSST)*.
[75] Vasily Tarasov, Deepak Jain, Geoff Kuenning, Sonam Mandal, Karthikeyani Palanisami, Philip Shilane, Sagar Trehan, and Erez Zadok. 2014. Dmdedup: Device Mapper Target for Data Deduplication. In *Ottawa Linux Symposium*.
[76] V. Tarasov, L. Rupprecht, D. Skourtis, A. Warke, D. Hildebrand, M. Mohamed, N. Mandagere, W. Li, R. Rangaswami, and M. Zhao. 2017. In Search of the Ideal Storage Configuration for Docker Containers. In *2nd IEEE International Workshops on Foundations and Applications of Self* Systems (FAS*W)*.
[77] Vasily Tarasov, Lukas Rupprecht, Dimitris Skourtis, Amit Warke, Dean Hildebrand, Mohamed Mohamed, Nagapramod Mandagere, Wenji Li, Raju Rangaswami, and Ming Zhao. 2017. In search of the ideal storage configuration for Docker containers. In *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE, 199–206.

[78] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. 2018. Cntr: Lightweight OS Containers. In *USENIX Annual Technical Conference (ATC)*.
[79] James Turnbull. 2023. The Docker Book: Containerization Is the New Virtualization.
[80] Amrita Upadhyay, Pratibha R Balihalli, Shashibhushan Ivaturi, and Shrisha Rao. 2012. Deduplication and compression techniques in cloud design. In *2012 IEEE International Systems Conference SysCon 2012*. IEEE, 1–6.
[81] Vdo. 2023. https://github.com/dm-vdo/vdo.
[82] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. 2012. Characteristics of Backup Workloads in Production Systems. In *10th USENIX Conference on File and Storage Technologies (FAST)*.
[83] Eberhard Wolff. 2016. *Microservices: Flexible Software Architecture*. Addison-Wesley Professional.
[84] Song Wu, Zhuo Huang, Pengfei Chen, Hao Fan, Shadi Ibrahim, and Hai Jin. 2022. Container-aware I/O stack: bridging the gap between container storage drivers and solid state devices. In *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 18–30.
[85] Xingbo Wu, Wenguang Wang, and Song Jiang. 2015. Totalcow: Unleash the power of copy-on-write for thin-provisioned containers. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*. 1–7.
[86] Qiumin Xu, Manu Awasthi, Krishna T Malladi, Janki Bhimani, Jingpei Yang, Murali Annavaram, and Ming Hsieh. 2017. Performance analysis of containerized applications on local and remote storage. In *Proc. of MSST*, Vol. 3. 24–28.
[87] ZFS. 2023. https://en.wikipedia.org/wiki/ZFS.
[88] Shiqiang Zhang, Song Wu, Hao Fan, Deqing Zou, and Hai Jin. 2020. BED: A Block-Level Deduplication-Based Container Deployment Framework. In *Green, Pervasive, and Cloud Computing: 15th International Conference, GPC 2020, Xi'an, China, November 13–15, 2020, Proceedings 15*. Springer, 504–518.
[89] Frank Zhao, Kevin Xu, and Randy Shain. 2016. Improving Copy-on-Write Performance in Container Storage Drivers. In *Storage Developer Conference (SDC)*.
[90] Nannan Zhao, Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Amit S. Warke, Mohamed Mohamed, and Ali R. Butt. 2019. Large-Scale Analysis of the Docker Hub Dataset. In *IEEE International Conference on Cluster Computing (Cluster)*.
[91] Ruijin Zhou, Ming Liu, and Tao Li. 2013. Characterizing the efficiency of data deduplication for big data storage management. In *IEEE International Symposium on Workload Characterization (IISWC)*.
[92] Benjamin Zhu, Kai Li, and R Hugo Patterson. 2008. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *6th USENIX Conference on File and Storage Technologies (FAST)*.