DupHunter: Flexible High-Performance Deduplication for Docker Registries

Nannan Zhao, Hadeel Albahar, Subil Abraham, and Keren Chen, *Virginia Tech;* Vasily Tarasov, Dimitrios Skourtis, Lukas Rupprecht, and Ali Anwar, *IBM Research—Almaden;* Ali R. Butt, *Virginia Tech*

https://www.usenix.org/conference/atc20/presentation/zhao

This paper is included in the Proceedings of the 2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

DupHunter: Flexible High-Performance Deduplication for Docker Registries

Nannan Zhao[1], Hadeel Albahar[1], Subil Abraham[1], Keren Chen[1], Vasily Tarasov[2], Dimitrios Skourtis[2], Lukas Rupprecht[2], Ali Anwar[2], and Ali R. Butt[1]
[1]*Virginia Tech* [2]*IBM Research—Almaden*

Abstract

The rise of containers has led to a broad proliferation of container images. The associated storage performance and capacity requirements place high pressure on the infrastructure of container registries that store and serve images. Exploiting the high file redundancy in real-world container images is a promising approach to drastically reduce the demanding storage requirements of the growing registries. However, existing deduplication techniques significantly degrade the performance of registries because of the high layer restore overhead.

We propose DupHunter, a new Docker registry architecture, which not only natively deduplicates layers for space savings but also reduces layer restore overhead. DupHunter supports several configurable *deduplication modes*, which provide different levels of storage efficiency, durability, and performance, to support a range of uses. To mitigate the negative impact of deduplication on the image download times, DupHunter introduces a *two-tier storage hierarchy* with a novel layer prefetch/preconstruct cache algorithm based on user access patterns. Under real workloads, in the *highest data reduction mode*, DupHunter reduces storage space by up to 6.9× compared to the current implementations. In the *highest performance mode*, DupHunter can reduce the `GET` layer latency up to 2.8× compared to the state of the art.

1 Introduction

Containerization frameworks such as Docker [2] have seen a remarkable adoption in modern cloud environments. This is due to their lower overhead compared to virtual machines [7,38], a rich ecosystem that eases application development, deployment, and management [17], and the growing popularity of microservices [69]. By now, all major cloud platforms endorse containers as a core deployment technology [10,28,31,47]. 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 [19].

Container images are at the core of containerized applications. An application's container image includes the executable of the application along with a complete set of its dependencies—other executables, libraries, and configuration and data files required by the application. Images are structured in *layers*. When building an image with Docker, each executed command, such as `apt install`, creates a new layer on top of the previous one [4], which contains the files that the command has modified or added. Docker leverages union file systems [64] to efficiently merge layers into a single file system tree when starting a container. Containers can share identical layers across different images.

To store and distribute container images, Docker relies on image *registries* (e.g., Docker Hub [3]). Docker clients can push images to or pull them from the registries as needed. On the registry side, each layer is stored as a compressed tarball and identified by a content-based address. The Docker registry supports various storage backends for saving and retrieving layers. For example, a typical large-scale setup stores each layer as an object in an object store [32,51].

As the container market continues to expand, Docker registries have to manage a growing number of images and layers. Some conservative estimates show that in spring 2019, Docker Hub alone stored at least 2 million *public* images totaling roughly 1 PB in size [59,72]. We believe that this is just the tip of the iceberg and the number of *private* images is significantly higher. Other popular public registries [9, 27, 35, 46], 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). As a result, 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 layer sharing. 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 [72]. Registry storage backends exacerbate the redundancy further due to the replication they perform to improve image durability and availability [12].

Deduplication is an effective method to reduce capacity demands of intrinsically redundant datasets [52]. However, applying deduplication to a Docker registry is challenging due to two main reasons: 1) layers are stored in the registry as **compressed** tarballs that do not deduplicate well [44]; and 2) 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$\times$ compared to a registry without deduplication.

In this paper, we propose DupHunter, the first Docker registry that natively supports deduplication. DupHunter is designed to increase storage efficiency via layer deduplication while reducing the corresponding layer restoring overhead. It utilizes domain-specific knowledge about the stored data and the storage system to reduce the impact of layer deduplication on performance. For this purpose, DupHunter offers five key contributions:

1. DupHunter exploits existing replication to improve performance. 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.
2. DupHunter deduplicates rarely accessed layers more aggressively than popular ones to speed up accesses to popular layers and achieve higher storage savings.
3. DupHunter monitors user access patterns and proactively restores layers *before* layer download requests arrive. This allows it to avoid reconstruction latency during pulls.
4. DupHunter groups files from a single layer in *slices* and evenly distributes the slices across the cluster, to parallelize and speed up layer reconstruction.
5. We use DupHunter to provide the first comprehensive analysis of the impact of different deduplication levels (file and block) and redundancy policies (replication and erasure coding) on registry performance and space savings.

We evaluate DupHunter on a 6-node cluster using real-world workloads and layers. In the *highest performance mode*, DupHunter outperforms the state-of-the-art Docker registry, Bolt [41], by reducing layer pull latencies by up to 2.8$\times$. In the *highest deduplication mode*, DupHunter reduces storage consumption by up to 6.9$\times$. DupHunter also supports other deduplication modes that support various trade-offs in performance and space savings.

2 Background and Related Work

We first provide the background on the Docker registry and then discuss existing deduplication works.

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 [20, 21]. 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 [51] or others [1, 5, 32, 51]. To scale the registry, organizations typically deploy a load balancer or proxy in front of several independent registry instances [11]. 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 [41] 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 [30]. A number of works have studied various dimensions of registry performance for a Docker image dataset [11, 14, 30, 60, 64, 71, 72]. However, such works do not provide deduplication for the registry. A community proposal exists to add file-level deduplication to container images [8], but as of now lacks even a detailed design, let alone performance analysis. Skourtis et al. [59] 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 [22,29,54,65], and are complementary to DupHunter.

2.2 Deduplication

Data deduplication has received considerable attention, particularly for virtual machine images [33, 36, 61, 73]. Many deduplication studies focus on primary and backup data deduplication [23–25,39,40,42,48,58,63,68,74] and show the effectiveness of file- and block-level deduplication [45, 62]. To further reduce storage space, integrating block-level deduplication with compression has been proposed [66]. In addition to local deduplication schemes, a global deduplication method [49] 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 [15]. 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 §3.2, 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.

3 Motivating Observations

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

3.1 Redundancy in Container Images

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 [72], 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.

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 ×

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.

3.2 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` [67], `Btrfs` [13], `ZFS` [70], `Jdupes` [34], in a single-node setup and on one distributed solution, `Ceph` [16], on a 3-node cluster. The deduplication block sizes are set to 4KB for both `VDO` and `Ceph`, and 128KB for both `Btrfs` [13] and `ZFS` [70] 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 [18, 44]. 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

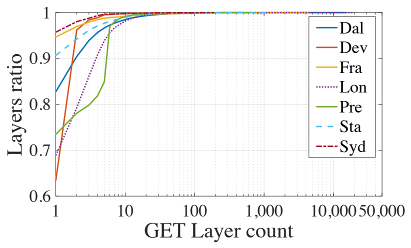

Figure 1: CDF of GET layer request count.

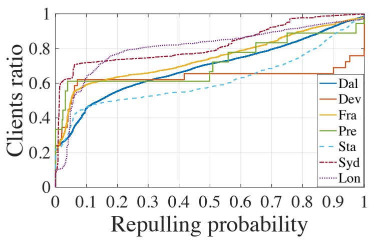

Figure 2: CDF of client repulling probability.

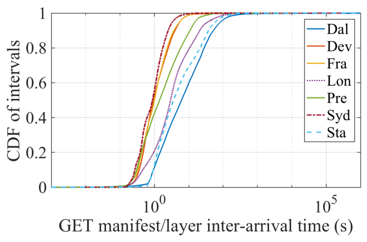

Figure 3: CDF of GET manifest/layer inter-arrival time.

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 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 [30], 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.

3.3 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. However, higher-level container orchestrators allow users to configure different policies for starting new containers. For example, Kubernetes allows policies such as IfNotPresent, i.e., only get the layer if it has not been pulled already, or AlwaysGet, i.e., always retrieve the layer, even if it is already present locally. These different behaviors need to be considered when predicting whether a layer will be pulled by a user or not.

We use the IBM Cloud registry workload [11] to analyze the likelihood for a user to *repull* an already present layer. The traces span ~80 days for 7 registry clusters: Dallas, Frankfurt, London, Sydney, Development, Prestaging, and Staging. Figure 1 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. However, there are clients that pull the same layers repeatedly. E.g., a client from London fetched the same layer 19,300 times.

Figure 2 shows the corresponding client repull probability, calculated as the number of repulled layers divided by the number of total GET layer requests issued by the same client. We see that 50% of the clients have a repull probability of less than 0.2 across all registries. We also observe that the slope of the CDFs is steep at both lower and higher probabilities, but becomes flat in the middle. This suggests that, by observing access patterns, we are able to classify clients into two categories, always-pull clients and pull-once clients, and predict, whether they will pull a layer or not by keeping track of user access history.

Layer preconstruction. We analyze the inter-arrival time between a GET manifest request and the subsequent GET layer request. As shown in Figure 3, 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

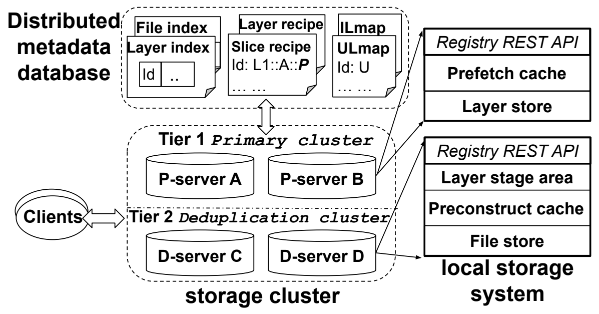

Figure 4: DupHunter architecture.

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 DupHunter (§4.1). We then describe in detail how it deduplicates (§4.2) and restores (§4.3) layers, and how it further improves performance via predictive cache management (§4.4). Finally, we discuss the integration of sub-file deduplication and erasure coding with DupHunter (§4.5).

4.1 Overview

Figure 4 shows the architecture of DupHunter. DupHunter consists of two main components: 1) a cluster of *storage servers*, each exposing the registry REST API; and 2) a distributed *metadata database*. When uploading or downloading layers, Docker clients communicate with any DupHunter server using the registry API. Each server in the cluster contains an API service and a backend storage system. The backend storage systems store layers and perform deduplication, keeping the deduplication metadata in the database. DupHunter uses three techniques to reduce deduplication and restoring overhead: 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. DupHunter also performs layer replication, but additionally deduplicates files inside the replicas.

A *basic deduplication mode* n (B-mode n) defines that DupHunter 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, DupHunter also provides a *selective deduplication mode* (S-mode). The S-mode utilizes the skewness in layer popularity, observed in [11], 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 DupHunter, for the example of B-mode 1, works as follows: DupHunter 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, DupHunter 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, DupHunter divides storage servers into two groups (Figure 4): 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 DupHunter to treat layers and individual files separately and prevent co-location during placement.

P- and D-servers form a 2-tier 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 DupHunter 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. DupHunter speeds up layer reconstruction through parallelism. As shown in Figure 4, each D-server's local storage is divided into three parts: the layer stage area, preconstruction cache, and file store. The layer stage 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 stage area.

DupHunter distributes the layer's unique files onto several servers (see §4.2). 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 DupHunter's metadata database. 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 improve the layer access latency, DupHunter 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, DupHunter 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, DupHunter predicts which layers in the image will be `pulled`, preconstructs the layers, and loads them in the preconstruct cache. To accurately predict which layers to prefetch, DupHunter maintains two maps: *ILmap* and *ULmap*. ILmap stores the mapping between images and layers while ULmap keeps track of a user's access history, i.e., which layers the user has pulled and how many times (see §4.4).

4.2 Deduplicating Layers

As in the traditional Docker registry, DupHunter maintains a *layer index*. After receiving a `PUT` layer request, DupHunter first checks the layer fingerprint in the *layer index* to ensure an identical layer is not already stored. If not, DupHunter, 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 stage areas of the D-servers. Once the replicas have been stored successfully, DupHunter 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* [26]. The file header contains metadata such as file name, path, size, mode, and owner information. DupHunter records every file header in slice recipes (described below) to be able to correctly restore the complete layer archive later.

To deduplicate a file, DupHunter 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. The file index maps different file Ids to their physical replicas stored on different D-servers.

Layer partitioning. DupHunter picks D-servers for 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, DupHunter employs a greedy packing algorithm. Consider first the simpler case in which each file only has a single replica. DupHunter first computes the total size of the layer's existing shared files on each D-server (this might be 0 if a D-server does not store any shared files for the layer). Next, it assigns the largest new unique file to the smallest partition until all the unique files are assigned. Note that during layer partitioning, DupHunter does not migrate existing shared files to reduce I/O overhead.

In the case where a file has more than one replica, DupHunter 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. DupHunter also ensures that two replicas of the same file are never placed on the same node.

Unique file replication. Next, DupHunter 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, DupHunter adds the new slice recipes to the metadata database.

DupHunter also creates a *layer recipe* 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

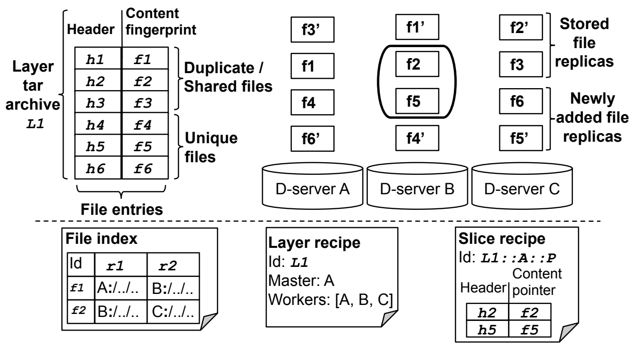

Figure 5: Layer dedup., replication, and partitioning.

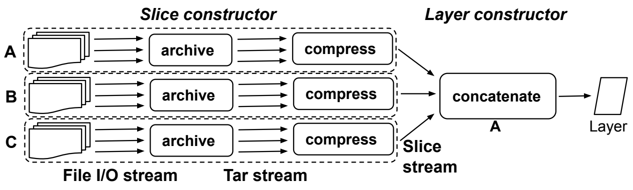

Figure 6: Parallel streaming layer construction.

and rebuilding the layer (see §4.3).

Figure 5 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 DupHunter, 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 $L1$, 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 *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.

4.3 Restoring Layers

The restoring process works in two phases: slice reconstruction and layer reconstruction. Considering the example in Figure 5, restoring works as follows:

According to $L1$'s layer recipe, the restoring workers are 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 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 and send them to A as shown in Figure 6. 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.

The layer restoration performance is critical to keep `pull` latencies low. Hence, DupHunter parallelizes slice reconstruction on a single node and avoids generating intermediate files on disk to reduce disk I/O.

4.4 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 IP address from the request. If DupHunter 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, DupHunter 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:

$$S_\Delta = ILmap[r.img] - ULmap[r.addr].$$

The layers in S_Δ are expected to be accessed soon.

Recall from §3.3 that some users *always* pull layers, no matter if the layers have been previously pulled. To detect such users, DupHunter maintains a *repull probability* γ for each user. For a `GET manifest` request r by a user

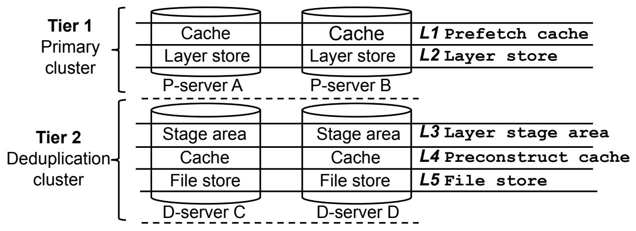

Figure 7: Tiered storage architecture.

r.addr, γ is computed as

$$\gamma[r.addr] = \sum_{l \in RL} l.pullCount / \sum_{l \in L} l.pullCount$$

where RL is the set of layers that the user has repulled before (i.e., with a pull count > 1) and L is the set of all layers the user has ever pulled. DupHunter updates the pull counts every time it receives a `GET layer` request.

DupHunter compares the clients' repull probability to a predefined threshold ε. If $\gamma[r.addr] > \varepsilon$, then DupHunter classifies the user as a repull user and computes the subset, $S_\cap$, of layers from the requested image that have already been pulled by the user:

$$S_\cap = ILmap[r.img] \cap ULmap[r.addr].$$

It then fetches the layers in $S_\cap$ into the cache.

Cache handling in tiered storage. The introduction of the two caches results in a 5-level 2-tier storage architecture of DupHunter as shown in Figure 7. 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 [43], which keeps track of both the frequently and recently used layers and adapts to changing access patterns.

4.5 Discussion

The goal of DupHunter is to provide flexible deduplication modes to meet different space-saving and performance requirements and mitigate layer restore overhead. The above design of DupHunter mainly focuses on file-level deduplication and assumes layer replication.

To achieve a higher deduplication ratio, DupHunter 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` [67] and `Ceph` [49]. 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 DupHunter exploits existing replication schemes, it is not limited to those. If the registry is using erasure coding for reliability, DupHunter can integrate with the erasure coding algorithm to improve space efficiency. Specifically, after removing redundant files from layers, DupHunter can store unique files as erasure-coded chunks. While DupHunter 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 [57]. To provide adequate fault tolerance, DupHunter maintains at least three copies of a layer (either as full layer replicas or unique files that can rebuild the layer) in the cluster.

5 Implementation

We implemented DupHunter[1] in Go by adding $\sim 2,000$ lines of code to Bolt [41]. Note that Bolt is based on the reference Docker registry [20] for high availability and scalability (see Bolt details in §2.1.)

DupHunter can use any POSIX file system to store its data and uses Redis [6] 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

[1]DupHunter's code is available at https://github.com/nnzhaocs/DupHunter.

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

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 DupHunter with a dedicated metadata database cluster.

To ensure that the metadata is in a consistent state, DupHunter 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` [55]). When a key already holds a value, a file duplicate or layer duplicate is identified and is removed from the registry cluster.

Additionally, DupHunter 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 DupHunter are scalable and can handle large image datasets. DupHunter's metadata overhead is about 0.6% in practice, e.g., for a real-world layer dataset of 18 GB, DupHunter stores less than 100 MB of metadata in Redis.

6 Evaluation

We answer two questions in the evaluation: how do deduplication modes impact the performance–redundancy trade-off, and how effective are DupHunter's caches.

6.1 Evaluation Setup

Testbed. Our testbed consists of a 16-node cluster, where each node is equipped with 8 cores, 16 GB RAM, a 500 GB SSD, 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, totalling 2 TB after decompression. Such dataset size allowed us to quickly evaluate DupHunter'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 performs with production registry workloads, we use the IBM Cloud Registry traces [11] that come from four production registry clusters (`Dal`, `Fra`, `Lon`, and `Syd`) and span approximately 80 days. We use Docker registry trace player [11] to replay the first 15,000 requests from each workload as shown in Table 2. We modify the player to match requested layers in the IBM trace with real layers downloaded from Docker Hub based on the layer size[2]. 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 [37] 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. 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's deduplication ratio and performance using different deduplication and redundancy schemes. The base case considers 3-way layer replication and file-level deduplication. In that case, DupHunter provides five deduplication modes: B-mode 0, 1, 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 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 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 [41] with 3-way replication as our baseline (*No-dedup*).

6.2 Deduplication Ratio vs. Performance

We first evaluate DupHunter'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 [53, 56]. Both replication and erasure cod-

[2]The original player generates random or zeroed data for layers.

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

Mode	Dedup. ratio	Performance improvement (P-servers)
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	Performance degradation (D-servers)
	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×

ing 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 (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. 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, even in B-mode 0, significantly decreases the layer restoring overhead compared to the naive approaches shown in Table 1 in §3.2. For example, DupHunter 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.

6.3 Cache Effectiveness

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

6.3.1 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 [50] and ARC [43]. Moreover, we compare DupHunter's prefetch cache with another prefetch algorithm, which makes predictions based on PUT requests [11] (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 8 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.

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. DupHunter'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 8, 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.

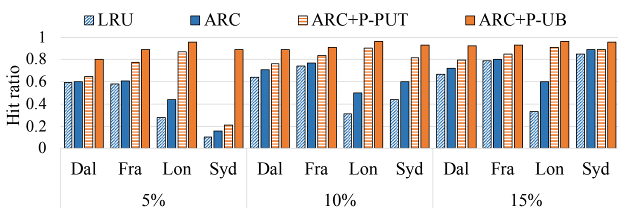

Figure 8: Cache hit ratio on P-servers with different cache algorithms.

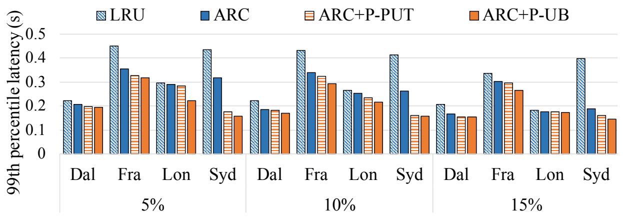

Figure 9: 99th percentile `GET` layer latency of P-servers.

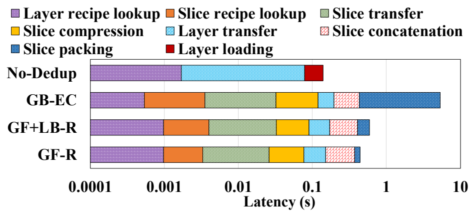

Figure 10: Layer restoring latency breakdown (X-axis is log-scale).

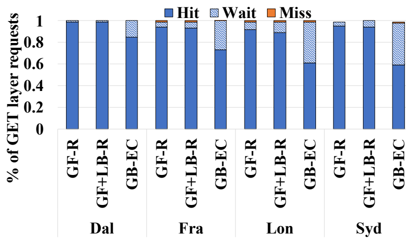

Figure 11: Preconstruct cache hit ratio.

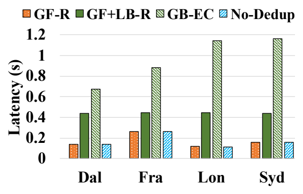

Figure 12: Performance of D-servers.

Figure 9 shows the 99th 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 99th 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 $\times$ compared to LRU for workload `Fra`. Overall, ARC+P-UB can largely improve `GET` layer performance for P-servers with a small cache size.

6.3.2 Preconstruct cache

For the preconstruct cache to be effective, 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 10, GF-R has the lowest layer restoring overhead compared to GF+LB-R and GB-EC. It takes 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$\times$. 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. Compared to GF-R, adding local block-level deduplication increases the overall overhead by up to 1.4$\times$ due to more expensive slice packing. It takes 0.18 s to pack a slice into an archive, 2.7$\times$ 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 11 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

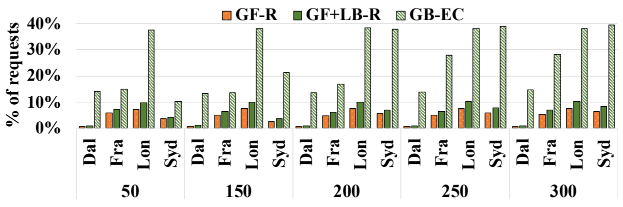

Figure 13: Request wait ratio with different number of clients.

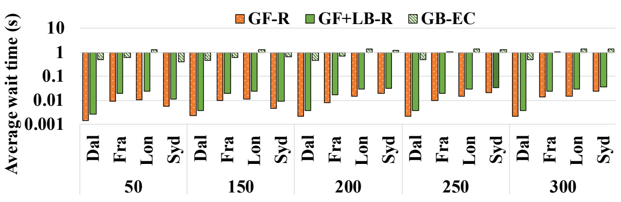

Figure 14: Average wait time with different number of clients (Y-axis is log-scale).

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 perfetch 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. Eg., 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 12 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.

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 13) and the average wait time (Figure 14).

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, a new Docker registry architecture that provides flexible and high performance deduplication for container images and reduces storage utilization. DupHunter supports multiple configurable deduplication modes to meet different space saving and performance requirements. Additionally, it parallelizes layer reconstruction locally and across the cluster to further mitigate overheads. Moreover, by exploiting knowledge of the application domain, DupHunter introduces a two-tier storage hierarchy with a novel layer prefetch/preconstruct 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.

Acknowledgments

We are thankful to the anonymous reviewers and our shepherd Abhinav Duggal for their valuable feedback. This work is sponsored in part by the National Science Foundation under grants CCF-1919113, CNS-1405697, CNS-1615411, and OAC-2004751.

References

[1] Aliyun Open Storage Service (Aliyun OSS). https://cn.aliyun.com/product/oss?spm=5176.683009.2.4.Wma3SL.

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

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

[4] Dockerfile. https://docs.docker.com/engine/reference/builder/.

[5] Microsoft azure storage. https://azure.microsoft.com/en-us/services/storage/.

[6] Redis. https://redis.io/.

[7] K. Adams and O. Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. *ACM SIGOPS Operating Systems Review*, 40(5):2–13, 2006.

[8] Alfred Krohmer. Proposal: Deduplicated storage and transfer of container images. https://gist.github.com/devkid/5249ea4c88aab4c7bff1b34c955c1980.

[9] Amazon. Amazon elastic container registry. https://aws.amazon.com/ecr/.

[10] Amazon. Containers on aws. https://aws.amazon.com/containers/services/.

[11] A. Anwar, M. Mohamed, V. Tarasov, M. Littley, L. Rupprecht, Y. Cheng, N. Zhao, D. Skourtis, A. S. Warke, H. Ludwig, D. Hildebrand, and A. R. Butt. Improving Docker Registry Design Based on Production Workload Analysis. In *16th USENIX Conference on File and Storage Technologies (FAST)*, 2018.

[12] N. Bonvin, T. G. Papaioannou, and K. Aberer. A Self-organized, Fault-tolerant and Scalable Replication Scheme for Cloud Storage. In *1st ACM Symposium on Cloud Computing (SoCC)*, 2010.

[13] Btrfs. https://btrfs.wiki.kernel.org/index.php/Deduplication.

[14] R. S. Canon and D. Jacobsen. Shifter: Containers for HPC. In *Cray User Group*, 2016.

[15] Z. Cao, H. Wen, F. Wu, and D. H. Du. {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)*, pages 309–324, 2018.

[16] Ceph. https://docs.ceph.com/docs/master/dev/deduplication/.

[17] Cloud Native Computing Foundation Projects. https://www.cncf.io/projects/.

[18] B. Compression and Deduplication. https://tinyurl.com/vgvb7wu.

[19] Datadog. 8 Surprising Facts about Real Docker Adoption. https://www.datadoghq.com/docker-adoption/.

[20] Docker. Docker Registry. https://github.com/docker/distribution.

[21] Docker. Docker Registry HTTP API V2. https://github.com/docker/distribution/blob/master/docs/spec/api.md.

[22] DockerSlim. https://dockersl.im.

[23] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, F. Huang, and Q. Liu. Accelerating Restore and Garbage Collection in Deduplication-based Backup Systems via Exploiting Historical Information. In *USENIX Annual Technical Conference (ATC)*, 2014.

[24] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, Y. Zhang, and Y. Tan. Design Tradeoffs for Data Deduplication Performance in Backup Workloads. In *13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[25] Y. Fu, H. Jiang, N. Xiao, L. Tian, and F. Liu. 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)*, 2011.

[26] GNU Tar. Basic Tar Format. https://www.gnu.org/software/tar/manual/html_node/Standard.html.

[27] Google. Google container registry. https://cloud.google.com/container-registry/.

[28] Google compute engine. Google Compute Engine. https://cloud.google.com/compute/.

[29] K. Gschwind, C. Adam, S. Duri, S. Nadgowda, and M. Vukovic. Optimizing Service Delivery with Minimal Runtimes. In *International Conference on Service-Oriented Computing (ICSOC)*, 2017.

[30] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Slacker: Fast Distribution with Lazy Docker Containers. In *14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[31] IBM Cloud Kubernetes Service. Ibm cloud kubernetes service. https://www.ibm.com/cloud/container-service.

[32] IBM Cloud Kubernetes Service. S3 storage driver. https://docs.docker.com/registry/storage-drivers/s3/.

[33] K. Jayaram, C. Peng, Z. Zhang, M. Kim, H. Chen, and H. Lei. An Empirical Analysis of Similarity in Virtual Machine Images. In *Middleware Industry Track Workshop*, 2011.

[34] jdupes. https://github.com/jbruchon/jdupes.

[35] JFrog Artifcatory. https://jfrog.com/artifactory/.

[36] K. Jin and E. L. Miller. The Effectiveness of Deduplication on Virtual Machine Disk Images. In *International Systems and Storage Conference (SYSTOR)*, 2009.

[37] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. 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)*, 1997.

[38] K. Kumar and M. Kurhekar. Economically Efficient Virtualization over Cloud Using Docker Containers. In *IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, 2016.

[39] M. Lillibridge, K. Eshghi, and D. Bhagwat. Improving Restore Speed for Backup Systems that use Inline Chunk-based Deduplication. In *11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[40] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *7th USENIX Conference on File and Storage Technologies (FAST)*, 2009.

[41] M. Littley, A. Anwar, H. Fayyaz, Z. Fayyaz, V. Tarasov, L. Rupprecht, D. Skourtis, M. Mohamed, H. Ludwig, Y. Cheng, and A. R. Butt. Bolt: Towards a Scalable Docker Registry via Hyperconvergence. In *IEEE International Conference on Cloud Computing (CLOUD)*, 2019.

[42] M. Lu, D. Chambliss, J. Glider, and C. Constantinescu. Insights for Data Reduction in Primary Storage: A Practical Analysis. In *International Systems and Storage Conference (SYSTOR)*, 2012.

[43] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *2nd USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[44] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, and J. Kunkel. A Study on Data Deduplication in HPC Storage Systems. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[45] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, and J. Kunkel. A Study on Data Deduplication in HPC Storage Systems. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[46] Microsoft. Azure container registry. https://azure.microsoft.com/en-us/services/container-registry/.

[47] Microsoft Azure. https://azure.microsoft.com/en-us/.

[48] A. Muthitacharoen, B. Chen, and D. Mazieres. A Low-bandwidth Network File System. In *ACM SIGOPS Operating Systems Review*, volume 35, 2001.

[49] M. Oh, S. Park, J. Yoon, S. Kim, K. Lee, S. Weil, H. Y. Yeom, and M. Jung. Design of global data deduplication for a scale-out distributed storage system. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1063–1073, 2018.

[50] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.

[51] OpenStack Swift storage driver. Openstack swift storage driver. https://docs.docker.com/registry/storage-drivers/swift/.

[52] J. Paulo and J. Pereira. A survey and classification of storage deduplication systems. *ACM Computing Surveys (CSUR)*, 47(1):11, 2014.

[53] J. S. Plank, M. Blaum, and J. L. Hafner. Sd codes: erasure codes designed for how storage systems really fail. In *FAST*, pages 95–104, 2013.

[54] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel. Cimplifier: Automatically Debloating Containers. In *11th Joint Meeting on Foundations of Software Engineering (FSE)*, 2017.

[55] Redis. SETNX. https://redis.io/commands/setnx.

[56] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[57] P. Shilane, R. Chitloor, and U. K. Jonnala. 99 deduplication problems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Denver, CO, June 2016. USENIX Association.

[58] H. Shim, P. Shilane, and W. Hsu. Characterization of Incremental Data Changes for Efficient Data Protection. In *USENIX Annual Technical Conference (ATC)*, 2013.

[59] D. Skourtis, L. Rupprecht, V. Tarasov, and N. Megiddo. Carving Perfect Layers out of Docker Images. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2019.

[60] R. P. Spillane, W. Wang, L. Lu, M. Austruy, R. Rivera, and C. Karamanolis. Exo-clones: Better Container Runtime Image Management Across the Clouds. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2016.

[61] K. Srinivasan, T. Bisson, G. R. Goodson, and K. Voruganti. iDedup: latency-aware, inline data deduplication for primary storage. In *10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[62] Z. Sun, G. Kuenning, S. Mandal, P. Shilane, V. Tarasov, N. Xiao, and E. Zadok. A Long-Term User-Centric Analysis of Deduplication Patterns. In *32nd International Conference on Massive Storage Systems and Technology (MSST)*, 2016.

[63] V. Tarasov, D. Jain, G. Kuenning, S. Mandal, K. Palanisami, P. Shilane, S. Trehan, and E. Zadok. Dmdedup: Device Mapper Target for Data Deduplication. In *Ottawa Linux Symposium*, 2014.

[64] V. Tarasov, L. Rupprecht, D. Skourtis, A. Warke, D. Hildebrand, M. Mohamed, N. Mandagere, W. Li, R. Rangaswami, and M. Zhao. In Search of the Ideal Storage Configuration for Docker Containers. In *2nd IEEE International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, 2017.

[65] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci. Cntr: Lightweight OS Containers. In *USENIX Annual Technical Conference (ATC)*, 2018.

[66] A. Upadhyay, P. R. Balihalli, S. Ivaturi, and S. Rao. Deduplication and compression techniques in cloud design. In *2012 IEEE International Systems Conference SysCon 2012*, pages 1–6. IEEE, 2012.

[67] Vdo. https://github.com/dm-vdo/vdo.

[68] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu. Characteristics of Backup Workloads in Production Systems. In *10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[69] E. Wolff. *Microservices: Flexible Software Architecture*. Addison-Wesley Professional, 2016.

[70] ZFS. https://en.wikipedia.org/wiki/ZFS.

[71] F. Zhao, K. Xu, and R. Shain. Improving Copy-on-Write Performance in Container Storage Drivers. In *Storage Developer Conference (SDC)*, 2016.

[72] N. Zhao, V. Tarasov, H. Albahar, A. Anwar, L. Rupprecht, D. Skourtis, A. S. Warke, M. Mohamed, and A. R. Butt. Large-scale analysis of the docker hub dataset. In *IEEE International Conference on Cluster Computing (Cluster)*, 2019.

[73] R. Zhou, M. Liu, and T. Li. Characterizing the efficiency of data deduplication for big data storage management. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2013.

[74] B. Zhu, K. Li, and R. H. Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *6th USENIX Conference on File and Storage Technologies (FAST)*, 2008.