Sift: Fast and Lightweight Docker Registries

Nannan Zhao[1], Hadeel Albahar[1], Subil Abraham[1], Ali Anwar[2], and Ali R. Butt[1]
[1]Virginia Tech, [2]IBM Research—Almaden

Abstract

The fast growing number of container images and the associated performance and storage capacity demands are key obstacles to sustaining and scaling Docker registries. The inability to efficiently and effectively deduplicate Docker image layers that are stored in a compressed format is a major obstacle. In this paper, we propose a new Docker registry architecture, Sift, that integrates caching and deduplication with Docker registries to increase deduplication efficacy and reduce storage needs while mitigating any performance overhead. Sift uses a highly-effective user-access-history-based prefetch algorithm, and a two-tier heterogeneous cache comprising memory and flash storage. The approach enables it to achieve a 96% hit ratio and save 56% more cache space.

1 Introduction

Containers have become an effective means for deploying modern applications because of desirable features such as tight isolation, low overhead, and efficient packaging of the execution environment [3]. However, sustaining and scaling container systems in the face of exponential growth is challenging. For example, Docker Hub [4]—a popular public container registry—stores more than 2 million public repositories. These repositories have grown at the rate of about 1 million annually —and the rate is expected to increase which requires provisioning a large amount of new storage consistently. This puts intense pressure on the availability and scalability of Docker registry storage infrastructure because scaling-out involves data migration between existing storage serves and new storage servers, even among existing storage servers for load balancing, which largely hurts performance. In this paper, we propose a new Docker registry architecture, Sift, that integrates caching and deduplication with Docker registries to help reduce the storage requirements while mitigating any performance overhead.

Containers are running instances of *images* that encapsulate complete runtime environment for an application. An image comprises a set of shareable and content addressable *layers*. Each layer is made up of a set of files that are compressed in a single archive. Docker images/layers are stored in an online store called Docker registry [4] and accessed by clients. Since a layer is uniquely identified by a collision-resistant hash of its content, no duplicate layers are stored in the registry. Many container registries use remote cloud storage, e.g., S3 [2], as their backend storage system (Figure 1).

Is current registry deduplication effective? To guide our design, we downloaded and analyzed 47 TB (167 TB uncompressed) of Docker images from Docker Hub (containing over 5 billion files). We found that only 3% of the stored files across the layers are unique; the remaining are redundant copies. This clearly shows that the current layer-based sharing mechanism is unable to effectively remove data duplicates. This is despite the fact that many cloud storage systems implement deduplication to effectively eliminate redundant data. For example, Google cloud and AWS, employ in-line transparent data deduplication [8]. But the compressed layers reduce the opportunity for deduplication significantly. Effectively removing redudncy from layers entails decompressing them before performing deduplication.

The challenge is that decompressing layers can have overheads besides the obvious decompression/compression. As shown in Figure 2, after decompression and simple file-level deduplication, the unique files may become scattered on multiple servers. To restore a layer, the layer's files would need to be first fetched from multiple servers, then compressed as a layer (to preserve transparency and avoid client-side API modifications) and sent back to the client. The extra network, I/O, and computation will slow down the response time for retrieving (pulling) an image.

Can caching help? Large container registry service providers such as Google and IBM use regional private registries across the world to facilitate a fast and highly-available service [1, 5]. This geographical distribution allows users to store images near their compute instances and experience a fast response time. An intuitive solution is to leverage these distributed registries as a cache to temporarily store popular layers and improve pull latency. The challenge here is that the layer access pattern is greatly different with traditional workloads since once a layer is pulled by a user, the layer won't be pulled again by this user because the layer is locally available to this user. This pattern has been observed by our registry workload analysis. The inter-access time for layer pulls can range from minutes to hours to months, making it difficult to manage the cache using traditional approaches such as LRU. Moreover, our Docker Hub image analysis shows that the layer sizes vary from a few MB to several GB and that a majority of layers sizes are around several MB. A small main memory cache cannot accommodate many layers, and thus is ineffective.

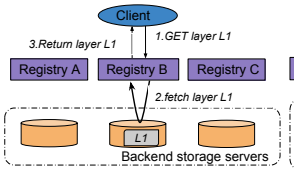

Figure 1: Without dedup.

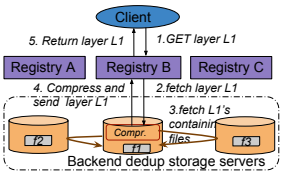

Figure 2: With dedup.

Sift to the rescue. Sift addresses the above challenges as follows. First, it explores layer access pattern and uses a user-access-history-based prefetch algorithm to accurately prefetch the layers that will be accessed shortly. Second, Sift embodies a two-tier heterogeneous cache architecture comprising a *layer buffer* and a *file cache* to hold prefetched layers. The layer buffer stores layers in main memory. The file cache stores the *deduped* unique files for the layers evicted from the layer buffer on a flash-based storage system. The mix of main memory and flash memory can realize the needed large capacity for caching prefetched layers and files. Our prefetch algorithm yields a hit ratio of 96%. Moreover, our two-tier heterogeneous cache is able to store 56% more layers into the file cache.

Sift use cases. on-prem registry. public-cloud registry.

2 Dataset analysis

2.1 Layer size distribution

2.2 File size distribution

2.3 Redundant file distribution

3 Deduplication performance analysis

On-cloud global deduplication software is widely adopted by cloud enterprises for reducing cloud storage consumption and overall storage cost. For example, StorReduce [8], the deduplication software used by Google cloud and AWS, performs in-line transparent data deduplication. Intuitively, such deduplication techniques can be leveraged to eliminate redundant data from the Docker image storage system. Except, the Docker image dataset is not amenable to deduplication as the images are *compressed archival files*.

As discussed in § 1, only 3% of the files in a sample Docker hub image collection were found to be unique, mainly becuase compressed files have a very low deduplication ratio [18]. Thus, we can realize significant space savings if we can remove the duplicate files. This entails decompressing files before performing deduplication, and collecting components of layers from multiple servers. To quantify the performance overhead of such an approach involving decompressing, deduplication, and then re-compressing, we setup five registry instances. Each instance has a local file system as their backend storage system. We implemented file-level deduplication with decompression and compression operations. We replayed the IBM registry workload `dal` [10] randomly to our five registries and measured the latency. Figure 3 shows the average latency observed across five registries. Note that since `dal` does not contain real layers, we extract the layer digest from each request and match it with a layer randomly selected from our Docker Hub dataset to emulate realistic requests.

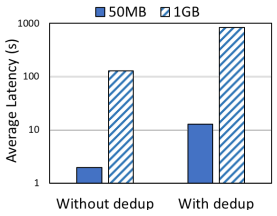

Figure 3: Average latency.

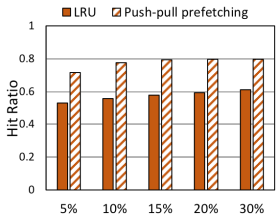

Figure 4: Hit ratio.

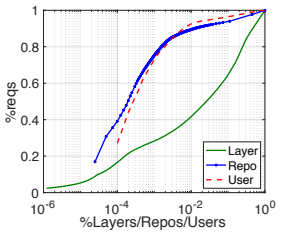

Figure 5: Popularity.

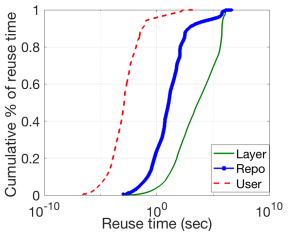

Figure 6: CDF of reuse time.

Without deduplication, the average latency for requesting a layer is about 2 s for layers with sizes <50 MB. The latency increases to 12 s when the above deduplication is implemented in the backend storage system. Furthermore, Docker registry performance drops down dramatically for larger layers. We observe that the average latency for requesting layers >50 MB and <1 GB is about 128 s. The latency worsens with dedupliation to on average of about 800 s.

4 Registry workload analysis

Understanding layer access patterns. We analyzed the Dallas(`dal`) registry workload collected from IBM Container Registry over the course of 75 days [10]. Figure 5 shows the registry accesses to layers and repositories, as well as users accesses to the layers and manifests. Layer accesses are heavily skewed. For example, 25% of popular layers account for 80% of all requests. For repository accesses and accesses by users, the skew is more significant than it is for layers. 94% of all requests accessed only 10% of the (popular) repositories. Similarly, only 9% of (most active) users issued 97% of all requests. This means that only a few extremely active users create their repositories in the `dal` registry and issue the majority of requests to the registry.

Figure 6 shows the reuse time of layers and repositories and users. Layer reuse time is the duration between two consecutive requests to the same layer or repository. Similarly, reuse time by user is duration between two

consecutive same-user requests. The layer reuse time is long. The median reuse time of a layer is 1.3 hours. 80% of repositories experience the highest request frequency, with a reuse time of 2 minutes. 90% of users remain active for at least 0.06 seconds, thus most layers are not requested within a short time period.

To quantify the efficacy of traditional LRU and push-pull prefetching, we implemented an LRU algorithm and push-pull prefetching [10] and replayed IBM registry workload `dal` with different cache size as shown in Figure 4. The hit ratio for LRU was found to be lower than 60% across different sizes. This is because layer accesses do not follow LRU temporal trend: recently pulled layer will not be pulled again by the same user. Although push-pull prefetching slightly improved the hit ratio compared to LRU, the maximum hit ratio is still low and stays stable around 80%. This is because push-pull prefetching predicts the future accesses based on pushed layers. However, based on our above trace analysis, only half of the `pull` layer requests have a preceding `push` layer request within the trace collection period of 75 days. This means that, after a user pushes a layer to the registry, it takes a few days, weeks, or even months for a user to make a `pull` request.

Registry as a cache for improving performance. Traditionally, caches are placed as close to the requesting client as proxy caches, or web/HTTP caches for the temporary storage of frequently requested data to mitigate the remote server lag. For example, Varnish [9] and Squid [7] are high-performance web/HTTP caches that accelerate web applications and improve response times by caching frequently requested web content. Docker registry is a web server that serves docker `pull` and docker `push` requests. Intuitively, registries can be deployed as proxy caches, i.e. a pull-through cache [6], to host frequently requested layers. This will speedup image pulls and improve performance.

However, the layer access pattern differs greatly from traditional request streams. Traditional LRU will place recently requested data into the cache because it is highly likely that the data will be requested again. However, it is the opposite for the layer access pattern: if a layer is pulled by a user, then this layer will not be pulled by the same user in the future since the layer is now locally available to the user. Thus, we have to design innovative mechanisms to manage any caches.

5 Sift Design

5.1 Overview

Sift seamlessly integrates caching and deduplication on the backend storage system (*backend dedup storage*) with Docker registries. We address a set of unique challenges to enable this integration. First, for caching layers, `pull` layer requests are difficult to predict because

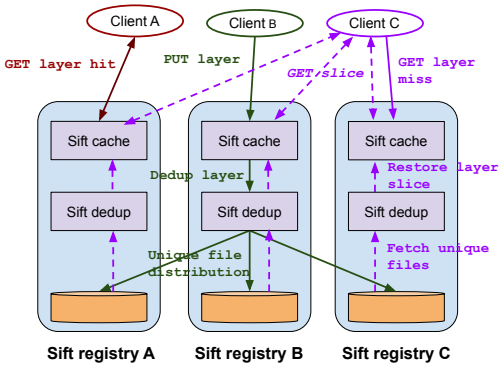

Figure 7: Architecture of Sift.

```
{
    …. …. …. ….
   "RootFS": {
          "Type": "layers",
          "Layers": [
                        {
                              "Id": "sha256:layer1Digest",
                              "Slices": [
                                    "sha256:layer1Slice1Digest",
                                    "sha256:layer1Slice2Digest",
                                    "sha256:layer1Slice3Digest",
                              ]
                        },
                        {
                              "Id": "sha256:layer2Digest",
                              "Slices": [
                                    "sha256:layer2Slice1Digest",
                                    "sha256:layer2Slice2Digest",
                                    "sha256:layer2Slice3Digest",
                              ]
                        },
          ]
    },
    …. …. …. ….
}
```

Figure 8: Sift manifest.

layers are accessed infrequently. In § 3, we have observed that about half of the layers are not accessed again for at least 1.3 hours. Which means that if we cache a layer, we may need to wait a long time before we observe a hit on that layer. This is mainly because when a user pulls an image from the registry, the Docker daemon on the requesting host will only pull the layers that are not locally stored.

Second, we can not deduplicate compressed layers. For deduplication, each layer needs to be uncompressed, and only then can undergo file-level deduplication. Similarly, to restore a layer, we need to fetch files from multiple servers, and only then compress them in to a tar file to serve a `pull` layer request. This whole process can incur a considerable performance overhead on `pull` layer requests. Deduplication also slows down `push` layer requests because of its high demand for CPU, memory, I/O, and network resources.

To address these issues, we propose a new registry design. The key feature of our design is a user-access-history-based prefetch algorithm that helps mitigate the performance degradation due to the backend dedup storage system (Figure 8). Based on layer access pattern we observed in § 3 and user access history information, Sift precisely prefetch the layers that may be pulled shortly.

Considering that layer sizes are typically about several MB [10], a small main memory cache will be unable to

accommodate all prefetched layers for all active users. To address this issue, we create separate caches for layers and *unique* files, called *layer buffer* and *file cache*, respectively. Note that, layers are compressed tarballs and buffered in layer buffer, and *unique* files are uncompressed files from which duplicates have been removed and stored on flash-based storage. For cache evictions, we first evict inactive users' layers from the layer buffer. Next, we *dedup* the evicted layers, then store the *unique* files into the file cache (detailed in § 5.2).

When a user requests a layer that is not present in the layer buffer, the request is forwarded to the file cache (detailed in § 5.2). If a layer is also not found in the file cache, the request is forwarded to the backend dedup storage system. Note that after layer deduplication, unique files are scattered across multiple servers. We define all the per-server files belonging to a layer as a *slice*. A server stores slices for many layers, and a layer is composed of slices stored on multiple servers. To avoid the network latency caused by fetching slices from different servers and assembling them into a whole compressed layer, we split a `pull` request into several `pull slice` requests. Those requests will then be forwarded to all the backend servers that store the requested layer's slices. After a `pull slice` request is received, each backend server compresses the slice and directly sends it back to the user. We modify the Docker client interface such that when it receives all the compressed slices, it can decompress them into a single layer. Furthermore, compressing slices in parallel considerably lowers the layer compression latency, since compression time depends on the size of the uncompressed data.

After receiving a `push` layer request from the client, Sift first caches the layer in Sift cache for later accesses. At the same time, Sift will also submit the layer to the backend deduplication storage system as shown in Figure 8. Our Sift cache use write through policies. Since there is no modification to the layer, there is no data consistency issue between the cache and the backend dedup storage system.

Docker client modifications

5.2 Layer restoring performance – aware deduplication

In the following, we describe our Sift layer restoring performance –aware deduplication (i.e., Sift LRPA dedup) design.

5.2.1 When to deduplicate layers?

Consider that deduplication incurs performance overhead and current registry already stores layers in compressed format to save space and network transfer overhead, we first analyze the space efficiency of a registry

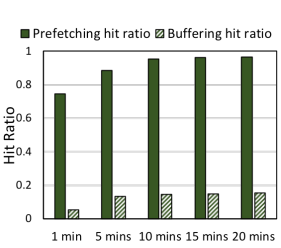

Figure 9: Hit ratio.

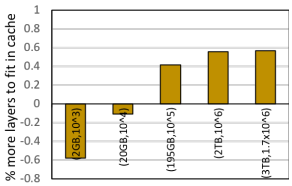

Figure 10: File-level deduplication vs. compression efficiency. **■ Nannan: figure needs redraw**

that performs decompress and file-level deduplication compared to a registry that naively stores compressed layers.

In Figure 10, the x-axis values correspond to the sizes of 5 random samples drawn from the whole dataset detailed in xxx and the size of the dataset in terms of capacity and layer count. For a traditional registry, the compressed layer tarballs will be kept as is. While a registry with file-level deduplication will store *deduplicated* layers (i.e., unique files). The y-axis shows how much space a registry with file-level deduplication can save compared to naively storing compressed layer tarballs. For the first two samples of the dataset, with size less than 20 GB, there is no benefit to *deduplicate* layers because the deduplication ratio is very low. However, when the dataset size is 3 TB, we can save 56% more space. The space saved by applying decompression and file-level deduplication increases almost linearly with the size of the layer dataset. This verifies the benefit of deduplicate layers when the dataset is large, which should be carefully selected to realize significant space savings.

5.2.2 Layer deduplication

After receiving a `push` layer request, Sift LRPA dedup will store the layer as it is and return a response back to client. LRPA dedup initiates layer deduplication process only if layer deduplication will achieve significant space savings and the process won't impact foreground requests. Sepcially, layer deduplication process is triggered when the layer dataset S is greater than a predefined threshold δ_s and the registry traffic RPS (i.e., requests per second) is lower than δ_{RPS}. Thus, layer deduplication process runs periodically. The process always stars with the cold layers that haven't been access for a long time.

The deduplication process has tree major steps: layer decompression, file-level deduplication, and unique file distribution. The first two steps are necessary for removing file duplicates from compressed layer tarballs. The last step unique file distribution is to balance the layer restoring load among registries so that layer restor-

ing process will achieve an optimal parallelism for each layer (detailed in xxx) and accelerate layer restoring process. After layer deduplication, unique files are evenly distributed across multiple registry servers. We define all the per-server files belonging to a layer as a *slice*. A server stores slices for many layers, and a layer is composed of slices stored on multiple servers, which allows restoring a layer in parallel.

Similar with traditional file-level deduplication, LRPA dedup maintains file fingerprint indexes to address the corresponding files in the storage system. Similar with traditional registry, LRPA dedup maintains layer fingerprint indexes to address the corresponding layers stored in the system. Besides, LRPA dedup creates a list of slice recipes identified by layer digest for each layer that is deduplicated. Slice recipe contains partial of the layer tarball's directory tree structure, file digests, and file metadata information (such as, permission and attributes), which are needed for restoring a slice for the layer. Note that, layer deduplication process only deduplicate regular files. Figure **??** shows an example of layer fingerprint indexes, file fingerprint indexes, and slice recipe. Fingerprint indexes, layer recipes, and Sift manifest are stored on distributed NoSQL databases for reliability, consistency, and fast accesses.

The deduplication process is detailed as follows:

1. check the layer fingerprint in the *layer fingerprint index* to ensure an identical layer is not already stored.
2. decompress and unpack the unique layer into files;
3. compute a *fingerprint* for every file in the layer;
4. check every file's fingerprint in the *file fingerprint index* to ensure an identical file is not already stored;
5. distribute the newly added unique files to the registry cluster by using round-robin, and;
6. calculate the slice digests based on unique file redistribution, and update the manifest by using COW.
7. update the *file index* with the unique files' fingerprints, host address;
8. create and store a list of *slice recipes* comprising the file path, address, metadata, and fingerprint of the layer's file;
9. remove the layer's tarball.

5.2.3 Parallel slice restoring

Each `pull layer` request has a precedent `pull manifest` request. Upon receiving a `pull manifest` request, Sift sends the updated Sift manifest to client. After receiving a Sift manifest, the client parses the manifest and sends either a `pull layer` request if the layer hasn't been deduplicated, or a list of `pull slice` requests if a list of corresponding slices presents in the manifest. Those `pull slice` requests will then be forwarded to all the registry servers that store the requested layer's slices as shown in Figure 8. After a `pull slice` request is received, the backend server compresses the slice and directly sends it back to the client.

The slice construction involves the following steps that are performed *inline*:

1. prepare a directory structure for the slice, based on the slice recipe;
2. copy the files into the directory tree;
3. compresses the slice's directory tree into a temporary tarball; and
4. send the slice tarball back to the client, and then discard the tarball.

5.2.4 Layer restoring assisted file cache (LRA file cache)

The layer slice restoring process has four suboperations: slice recipe lookup, slice file copying, slice compression, and slice network transfer. To measure the latency for each suboperation, we implemented layer deduplication and parallel slice restoring on a 4-node registry cluster. We first warmup the cluster by pushing 200 layers to the cluster and initiating layer deduplication process. The layers were randomly selected from our layer dataset detailed in xxx limited to 50MB. After finishing layer deduplication, we sent 400 `pull slice` requests to the cluster with 10 `pull slice` requests issued at a time. Figure **??** shows the CDFs of each suboperation. We see that across the four suboperations, the duration for slice compress is the shortest. Slice compression only took less than 0.001 s because a slice is much smaller than a layer. The next shortest suboperation is network transfer since we pulled layer slice through Ethernet. 90% of slice recipe lookups took less than 0.1 s while the highest slice recipe lookup duration almost reaches 0.8 s, which is caused by high concurrent lookup requests (note that we use redis to store metadata ■ **Nannan: use mongodb instead**). The most time consuming suboperation is slice file copying, which involves copying all the files that belong to the slice to their destination directory based on the slice recipe. Note that we implemented a thread pool on each registry server to read files in parallel and write data in RAMdisk to reduce disk IOs. 40%of slice file copying duration is greater than 1 s and 10% of slice file copying duration is higher than 10 s. This is because bigger slices contains more files and requires more disk IOs.

To reduce slice file copying overhead, Sift LRA file cache temporally cache a subset of files for bigger and popular slices that have a high slice restoring latency, ie., $D_{rs} > \theta_{rs}$. Upon a pull slice request for those slices, LRPA dedup fetches a certain amount of files from LRA file cache instead of disks. For on-premise or private registry cluster, the network transfer speed is usually faster than remote cloud. Thus, slice compression is less

important for medium to small size slices, especially for the slices that has a high decompression latency, i.e., $D_{nt} < \theta_{nt}$ and $D_{dc} > \theta_{dc}$. Consequently, LRPA dedup only archives these slices without compressing them and directly sends these archival files back to the clients to eliminate clients' decompression latency.

To know which slices have a high slice restoring latency, LRPA dedup monitors slice restoring performance and maintains a restoring performance profile for each slice that has been restored as shown in Figure **??**, which contains the latency breakdown of slice restoring (,and a decompression latency updated by layer decompression process) and its containing files' metadata, such as file size. All the slice restoring performance profiles are also stored in distributed NoSQL databases, and addressed by slice digests. To estimate the restoring latency for a slice i that hasn't been restored, LRPA dedup lookups the slice restoring performance profiles by slice size, selects a slice (x)'s restoring performance profiles that is most similar in size to it, and estimates i's restoring latency as: $D_{rs}(i) \approx D_{rs}(x) + \varepsilon_{rs}$, where ε_{rs} is the standard error of restoring latency estimation for the layers similar in size. If the estimated slice restoring $D_{rs}(i) > \theta_{rs}$, then, LRPA dedup lookups the slice restoring performance profiles by slice size, selects a slice (y)'s restoring performance profiles that has a low restoring latency and most similar in size to it. Next, LRPA dedup caches a certain amount of files $\bigcup(files)$ for slice i, so that $D_{rs}(i) - D_{cp}(\bigcup(files)) \approx D_{rs}(y)$.

Note that LRA file cache size is limited so that LRA file cache only caches popular slices that will be accessed later. Next section will describe how to determine popular layers based on user access patterns. Note that the slices for the same layer have similar sizes, restoring latencies, and popularity because of unique file distribution. Thus, once a layer is determined as popular layer, LRPA dedup will apply the same estimation function to all its slices and cache similar amount of files for its slices.

5.3 User behavior based layer preconstruction cache

5.3.1 User access patterns

To improve our cache hit ratio for `pull` layer request, we propose a user-access-history-based prefetch algorithm. The algorithm exploits the uniqueness of the registry's dataset hierarchy: repositories comprise a list of layers. When a user *pulls* an image from a repository, it will first `pull` the manifest of the image [3] [10] and parse the manifest to get the layer digests, then lookup each layer digest against a *local layer digest index*. After that it only *pulls* the layers that has *not been stored locally*.

User profiles Based on the above pattern and hierarchy, we can record the users' repository and layer access history. Theoretically, once a user issues a `pull` manifest from a repository, all the layers that belong to this repository but have not been *pulled* by this user should be prefetched into the cache. In this case, the cache hit ratio will reach 1.

As shown in algorithm 1, Sift maintains two maps: a RLMap for recording layer-repository relationship, and a URLMap for recording users' repository and layer access history information. For example, if a user *U pulls* a layer *L* from a repository *R*, Sift will add an new entry (*U,L*) in URLMap. While if a user *U pushes* a layer *L* to a repository *R*, Sift will add an new entry (*R,L*) in RLMap. Note that to identify which layers are locally available for a user, we extract *user end host address* (*r.client*) from each request and define the user end host address as user, and keep track of all layers that have been downloaded by *r.client*. When either a GET manifest request is received or a miss on a GET layer request happens, Sift will lookup RLMap and get the requested repository's containing layers, and compare against the layers that are already *pulled* by the user by looking up URLMap, then prefetch the layers that have not been *pulled*. We set a timer for each cached layer and evict it when its timer is $> U_{thresh}$. To incorporate the algorithm with Sift, we prefetch `slices` in parallel from backend servers, buffer them in the layer buffer first, then evict them into the file cache after they *cool down*.

User "Pull manifest" request as an indicator

5.3.2 Layer preconstruction

5.3.3 Temporal trend

LRU of (user+repo) based cache eviction

6 Implementation

We decoupled our Sift implementation into cache implementation and backend dedup storage implementation. We first modify the Docker registry source code – local file system driver part by implementing decompression/compression, file-level deduplication, and unique file distribution modules to the local file system driver so that the driver decompresses the layers and removing the duplicate files once the registry receives a layer tarfile, and fetching and compress the files into layer slices once the registry receives a get layer request. We also implemented a simple round-robin based distributed key-value object store by modifying the local file system driver. Once the file-level deduplication is done, the driver will re-distributed the newly added unique files to different destination servers based on round-robin. Besides, we uses Zookeeper as the cluster coordinator, and MongoDB to store dedup metadata.

Algorithm 1: User access history based prefetch

```
Input:
L_thresh: Threshold for duration to keep a prefetched layer.
/*when L_timer[layer] > L_thresh, layer is evicted or demoted
to Flashcache */
RLMap: Repository to layers map.
URLMap: User to layers map.
while true do
    r ← request received
    if r = GET manifest then
        layers ← RLMap[r.repo] − URLMap[r.client]
        OnTimelayers, NotOnTimelayers ←
          OnTimeCalculation(layers)
        MEMcache ← Prefetch(OnTimelayers)
        FLASHcache ← Prefetch(NotOnTimelayers)
        set L_timer[layer] for each layer in layers
    else if r = PUT layer then
        update URLMap[(r.client, r.layer)]
        update RLMap[(r.repo, r.layer, put)]
        MEMcache ← buffer r.layer
        set L_PUT_timer[r.layer]
    else if r = GET layer then
        if r.layer in MEMcache or r.layer in FLASHcache then
            serve from MEMcache or FLASHcache
            update URLMap[(r.client, r.layer)]
            Reset L_timer[r.layer]
            hit++
            /* if r.layer in FLASHcache, layer is promoted to
            MEMcache/
        else
            serve from backend storage system
            update URLMap[(r.client, r.repo, repulled)]
            RepulledLayers ← RLMap[r.repo]
            FLASHcache ← Prefetch(RepulledLayers)
            set L_timer[layer] for each layer in RepulledLayers
```

Algorithm 2: User access history based eviction.

```
Input:
T_mem: Capacity threshold for MEM cache to trigger demotion.
T_flash: Capacity threshold for FLASH cache to trigger eviction.
UsrLRU: LRU of users.
LayerLRU[Usr]: LRU of layers that are accessed by user Usr.
RepoLRU: LRU of repostiories.
LayerLRU[Repo]: LRU of layers that are associated with
repository Repo.
while free_MEM < T_mem do
    last_usr ← UsrLRU.last_item()
    for last_layer ← LayerLRU[last_usr].last_item() do
        if layer exclusively belongs to last_usr then
            FLASHcache ← Demote(last_layer)
            free_MEM + = sizeof(last_layer)
        end
    end
end
while free_FLASH < T_flash do
    last_repo ← RepoLRU.last_item()
    for last_layer ← LayerLRU[last_repo].last_item() do
        if layer exclusively belongs to last_repo then
            Discard(last_layer)
            free_FLASH + = sizeof(last_layer)
        end
    end
end
```

Our Sift cache is also implemented by modifying Docker registry source code – cache part. The original Docker registry only caches local layer digests to identify if a requested layer is stored locally or not. We first deploy our cache cluster by using Redis cluster. We add our Sift cache into Docker registry source code as a new cache module which makes prefetch decisions when a GET/PUT request is received and talks to Redis cache. The metadata tables used by cache is also stored in MongoDB. For cache demotion/eviction and monitoring cache space utilization/backend dedup storage system performance, we elect a master registry node to do that by using Zookeeper as the cluster coordinator.

Therefore, by modifying Docker registry source code, we can configure Docker registry to provide two different kinds of functionalities. It can be either used as a distributed backend layer dedup storage system with the same Docker registry APIs or used as a distributed user-oriented cache with the same Docker registry APIs.

Gzip compression

7 Evaluation

7.1 Testbed

Our testbed includes two clusters: backend registry storage cluster and frontend registry cache cluster. Backend storage cluster includes 12 servers. Each server is equipped with 32 cores, 64 GB RAM, 500 GB SSD and 1 TB HDD. Our frontend cache cluster container 24 servers. Each server is equipped with 8 cores,16 GB RAM and 500 GB SSD. We implemented Sift cache on frontend cache cluster and installed Sift dedup on backend storage cluster. We use extra 5 machines and each machine launches different number of clients to emulate client requests.

7.2 Workloads and dataset

We emulated a real-world workloads by combining IBM traces and an image dataset downloaded from Docker Hub.

7.3 Deduplication evaluation

7.3.1 Pull performance

7.3.2 Push performance

7.3.3 Deduplication performance

8 Preliminary Evaluation

Cache hit ratio. We simulate our user-access-history-based cache and replay the `dal` workload [10] to measure the hit ratio. We set the cache size to be 20% of the data ingress for `dal`. We set 10% of the cache for buffering incoming `put` layer requests, and the rest for caching prefetched layer slices from backend servers. Note that in this evaluation, the cache only contains the

layer buffer without the file cache. Figure 9 shows the results. We observe a significant increase in the hit ratio, 74% to 95% as the duration threshold grows from 1 to 10 minutes. This is because prefetched layers are kept in the cache for more time. The hit ratio stablizes at 96% as the duration threshold increases from 15 to 20 minutes. The layers responsible for the 4% miss rate are the ones being *re-pulled* by the same user. We see that, across different duration thresholds, the hits upon buffering newly put requests (denoted as buffering hit ratio) is very low, confirming that it takes a long time for a recently *pushed* layer to be pulled. We also observe a 22% average cache utilization. That is because our algorithm is based on users demand so it adapts to workload changes.

Space efficiency. We analyze the space efficiency of the file cache compared to a cache that naively stores compressed layers. In Figure 10, the x-axis values correspond to the sizes of 4 random samples drawn from the whole dataset and the size of the dataset in terms of capacity and layer count. For a traditional cache, the compressed layer tarballs will be kept as is. While Sift will store *deduped* layers. The y-axis shows how many more *deduped* layers can fit in our file cache compared to naively storing compressed layer tarballs. For the first two samples of the dataset, with size less than 20 GB, there is no benefit to *dedup* layers because the deduplication ratio is very low. However, when the dataset size is 3 TB, we can store 56% more *deduped* layers' unique files in file cache. The number of extra *deduped* layers that can fit in the file cache increases almost linearly with the size of the layer dataset. This verifies the benefit of the file cache when the cache size is large, which should be carefully selected to realize significant space savings.

8.1 Cache evaluation

9 Related Work

A number of studies investigated various dimensions of Docker storage performance [11, 12, 13, 21, 25, 28, 26, 13]. Anwar et al. [10] performed a detailed analysis of an IBM Docker registry workload but not the dataset. Data deduplication is a well explored and widely applied technique [15, 19, 22, 24, 30]. A number of studies which focus on real-world datasets [14, 16, 17, 20, 23, 27, 29] can be complementary to our approach.

10 Conclusion

We presented Sift, a new Docker registry architecture that integrates caching and deduplication to improve the registry's performance and decrease the storage capacity requirement. Sift leverages a two-tier heterogeneous cache architecture to efficiently improve cache space utilization. Our prefetch algorithm can improve the cache hit ratio and mitigate the overhead of decompression-enabled deduplication.

11 Discussion

While the performance aspects of Docker containers on client side have been studied extensively, we believe the current registry design is inefficient and will become a bottleneck when handling the expected massive amount of images. In this work, we have attempted to solve the registry redundant data issue using an integrated caching and deduplication approach that is guided by user access patterns. Our approach showcases the benefits for both the Docker registry service providers as well as the clients.

We believe that our paper will lead to discussion on the following points of interest: (1) The need for a deduplication approach tailored for Docker registry. Since deduplication is a well-studied area, we believe that the paper will lead to hot discussion on how the extant approaches can be applied/adapted to diverse data formats such as compressed files. (2) Issues of how to effectively manage a layer/file cache by exploiting user access patterns, and where to place the cache (e.g., at registry side, remote cloud storage side, or Docker client side) to benefit overall container performance. (3) The role of layer and image. The redundant data issue stems from containers' storage virtualization technique that use union file systems to pack everything inside different layers (that are used as a plugin OS image). An open aspect of our work is improving the efficiency of current union file systems and container storage virtualization technique.

An unexplored issue is redundant data on the Docker client side, which differs greatly from registry storage. This is because the registry stores compressed layers, while client-end host-storage system stores uncompressed layers. Therefore, the client-side redundancy issue under limited local storage space is just as bad or worse as the redundancy in the registry storage system. However, applying current deduplication methods—that are usually deployed on backup storage systems—on the client side would greatly impact the container runtime performance. This is because performing file-level deduplication on the Docker client side requires intensive file fingerprint calculations, and file fingerprint lookup, which will slowdown performance. Furthermore, we believe that our work can help synchronize deduplication for both client side deduplication and registry side deduplication to yield desirable win-win solutions.

References

[1] About ibm cloud container registry. https://cloud.ibm.com/docs/services/Registry?topic=registry-registry_overview#registry_overview.

[2] Amazon s3. https://aws.amazon.com/s3/.

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

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

[5] Google's container registry. https://cloud.google.com/container-registry/.

[6] Registry as a pull through cache. https://docs.docker.com/registry/recipes/mirror/.

[7] Squid Cache - Optimising Web Delivery. http://www.squid-cache.org/.

[8] Storreduce. http://storreduce.com.

[9] Varnish HTTP Cache. https://varnish-cache.org/.

[10] ANWAR, A., MOHAMED, M., TARASOV, V., LITTLEY, M., RUPPRECHT, L., CHENG, Y., ZHAO, N., SKOURTIS, D., WARKE, A. S., LUDWIG, H., HILDEBRAND, D., AND BUTT, A. R. Improving docker registry design based on production workload analysis. In *USENIX FAST'18*.

[11] BHIMANI, J., YANG, J., YANG, Z., MI, N., XU, Q., AWASTHI, M., PANDURANGAN, R., AND BALAKRISHNAN, V. Understanding performance of I/O intensive containerized applications for NVMe SSDs. In *IEEE IPCCC'16*.

[12] CANON, R. S., AND JACOBSEN, D. Shifter: Containers for HPC. In *Cray User Group'16*.

[13] HARTER, T., SALMON, B., LIU, R., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Slacker: Fast Distribution with Lazy Docker Containers. In *USENIX FAST'16*.

[14] JIN, K., AND MILLER, E. The effectiveness of deduplication on virtual machine disk images. In *ACM SYSTOR'09*.

[15] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZISE, G., AND CAMBLE, P. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *USENIX FAST '09*.

[16] LU, M., CHAMBLISS, D., GLIDER, J., AND CONSTANTINESCU, C. Insights for data reduction in primary storage: A practical analysis. In *ACM SYSTOR'12*.

[17] MEISTER, D., KAISER, J., BRINKMANN, A., CORTES, T., KUHN, M., AND KUNKEL, J. A study on data deduplication in HPC storage systems. In *ACM/IEEE SC'12*.

[18] MEISTER, D., KAISER, J., BRINKMANN, A., CORTES, T., KUHN, M., AND KUNKEL, J. A study on data deduplication in hpc storage systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012), IEEE Computer Society Press, p. 7.

[19] MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. A low-bandwidth network file system. In *ACM SOSP'01*.

[20] SHIM, H., SHILANE, P., AND HSU, W. Characterization of incremental data changes for efficient data protection. In *USENIX ATC'13*.

[21] SPILLANE, R. P., WANG, W., LU, L., AUSTRUY, M., RIVERA, R., AND KARAMANOLIS, C. Exoclones: Better Container Runtime Image Management Across the Clouds. In *USENIX HotStorage'16*.

[22] SRINIVASAN, K., BISSON, T., GOODSON, G., AND VORUGANTI, K. iDedup: Latency-aware, inline data deduplication for primary storage. In *USENIX FAST'12*.

[23] SUN, Z., KUENNING, G., MANDAL, S., SHILANE, P., TARASOV, V., XIAO, N., AND ZADOK, E. A long-term user-centric analysis of deduplication patterns. In *IEEE MSST'16*.

[24] TARASOV, V., JAIN, D., KUENNING, G., MANDAL, S., PALANISAMI, K., SHILANE, P., TREHAN, S., AND ZADOK, E. Dmdedup: Device mapper target for data deduplication. In *Ottawa Linux Symposium'14*.

[25] TARASOV, V., RUPPRECHT, L., SKOURTIS, D., WARKE, A., HILDEBRAND, D., MOHAMED, M., MANDAGERE, N., LI, W., RANGASWAMI, R., AND ZHAO, M. In Search of the Ideal Storage Configuration for Docker Containers. In *IEEE AMLCS'17*.

[26] THALHEIM, J., BHATOTIA, P., FONSECA, P., AND KASIKCI, B. Cntr: Lightweight OS containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, 2018), USENIX Association, pp. 199–212.

[27] WALLACE, G., DOUGLIS, F., QIAN, H., SHILANE, P., SMALDONE, S., CHAMNESS, M., AND HSU, W. Characteristics of backup workloads in production systems. In *USENIX FAST'12*.

[28] ZHAO, F., XU, K., AND SHAIN, R. Improving Copy-on-Write Performance in Container Storage Drivers. In *SNIA SDC'16*.

[29] ZHOU, R., LIU, M., AND LI, T. Characterizing the efficiency of data deduplication for big data storage management. In *IEEE IISWC'13*.

[30] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *USENIX FAST '08*.