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 [6]. But the compressed layers reduce the opportunity for deduplication significantly. Effectively removing reduducncy 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.

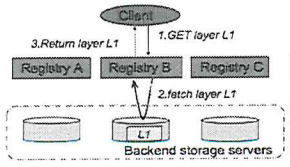


Figure 1: Without dedup.

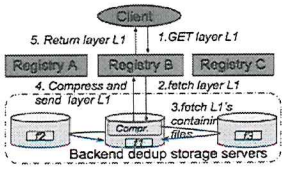


Figure 2: With dedup.

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

2.4 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 [6], 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 [15]. 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` [7] 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.

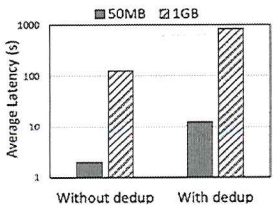


Figure 3: Average latency.

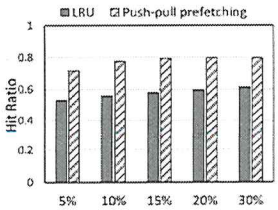


Figure 4: Hit ratio.

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.

3 Sift Design

In this following, we first present an overview of Sift. Then, we present two main components of Sift: LRPA deduplication system and UBLP cache respectively.

3.1 Overview

Figure 5 shows the architecture of Sift. Sift is comprised of a distributed registry cluster and a distributed metadata database. Layers are stored on registry cluster and Metadata, such as Docker image manifests, are stored on distributed NoSQL databases for reliability, consistency, and fast accesses. Each Sift registry consists of a user behavior based layer preconstruct cache (UBLP cache), and a layer restoring latency aware deduplication (LRPA deduplication) system.

In the following, we describe how Docker clients interact with Sift.

Push As shown in Figure 5, client *A* creates a new hello-world image `hello-world:new` from the official image which only contains a single layer *L1* by committing the modifications over *L1* as a new layer *L2*. When client *A* pushes `hello-world:new` to the registry, it only pushes the new layer *L2* to the registry since registry already stores *L1*. When Sift receives *L2*, it first caches *L2* in UBLP cache for later accesses, and at the same time, Sift will also submit *L2* to the backend storage system as shown in Figure 5. UBLP cache uses write through policies. Since there is no modification to the layer, there is no data consistency issue between

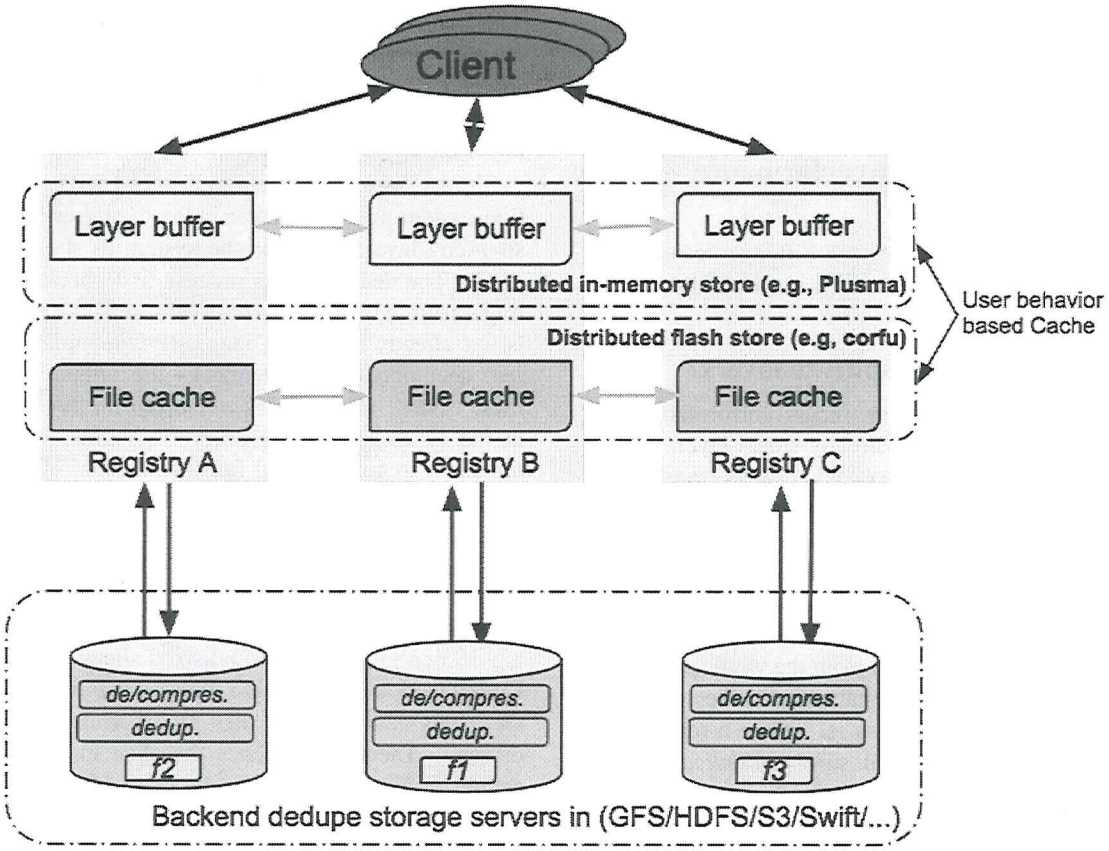


Figure 5: Architecture of Sift.

the cache and the backend storage system. Next, client *A* pushes a new manifest *M1:0* to the registry and finishes image pushing. As shown, *L2* is added to the manifest *M1:0* for image `hello-world:new`.

The LRPA deduplication process runs periodically to deduplicate compressed layer tarballs (detailed in § 3.2) into unique files to save storage space. As shown in Figure 5, cold layer *L2* is selected to be deduplicated. LRPA deduplication process decompresses `L2` and removes the duplicated files from *uncompressed* `L2`. After that, LRPA deduplication process evenly distributes the unique files to the registry servers. In this case, each server stores a **deduplicated slice** of `L2`, from which a layer **slice** of `L2` can be constructed. We define all the per-server files belonging to a layer as a *deduplicated slice*. A server stores deduplicated slices for many layers, and a layer is composed of *slices* that can be restored from the deduplicated slices stored on multiple servers, which allows restoring a layer in parallel. To do that, LRPA deduplication process uses copy-on-write to update the old manifest *M1:0* by adding slices' digests into it and generates new manifest *M1:1* as shown in Figure 5. Slice digest is calculated by hashing slice content [?].

Pull As shown in Figure 5, when client *C* pulls an official image `hello-world` from registry, Sift first checks if the requested layer *L1* presents in UBLP cache. If so, the `pull` layer request will be served by cache. Otherwise, LRPA deduplication process starts parallel slice restoring process. For example, when client *B* pulls image `hello-world:new` from registry, Sift sends the latest manifest *M1:1* to client. After receiving the *M1:1*, client *B* first parses *M1:1* and gets a list of slice digests for *L2*. Instead of sending "`pull layer L2`" request, client will send multiple "`pull slice of L2`" requests to registry. As shown, client *B* sends "`pull slice S1`", "`pull slice S2`", and "`pull slice S3`" to the registry since *L2* is comprised of *S1* , *S2*, and *S3*. These requests are forwarded to the servers that store corresponding deduplicated slice of *L2*. The servers will start to restore slices from local deduplicated slices and send individual slices back to client in parallel as shown in Figure 5. When client *B* receives all the slices, it decompresses them together as an uncompressed layer.

Docker client modifications To interact with Sift, Docker client is modified to parse Sift manifest with additional attributes – slice digests. If slice digests present in a layer object in the manifest JSON file, client will

replace "`pull layer` request with multiple "`pull slice`" requests. Besides, when client receives slices back from registry, it decompresses these slices together into an uncompressed layer.

3.2 Layer restoring performance – aware deduplication

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

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

In Figure 6, the x-axis values correspond to the sizes of 5 random samples drawn from the whole dataset (detailed in § 2). 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 deduplicating layers when the dataset is large, which should be carefully selected to realize significant space savings.

3.2.2 Layer deduplication

After receiving a `push` layer request, Sift will store the layer as it is and return a response back to client. LRPA deduplication system 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 θ_s and the registry traffic RPS (i.e., requests per second) is lower than θ_{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 restoring process will achieve an optimal parallelism for each layer (detailed in § 3.2.2) and accelerate layer restoring process.

Similar with traditional registry, LRPA deduplication system maintains layer index to address the corresponding layers stored in the system as shown in Figure 7. The deduplication process first checks the layer fingerprint in the *layer index* to ensure an identical layer is not already stored. Then, LRPA deduplication system decompresses and unpacks the unique layer into files. Next, it computes a *fingerprint* for every file in the slices and checks every file's fingerprint in the *file index* to get identical files' server addresses. After that, it uses a weighted round robin algorithm to distribute newly added unique files to the registry cluster. These servers that already contains this layer's identical files are assigned a lower weight. This is to ensue that different servers maintain same amount of files that are needed to restoring equal sized slices for this layer. Deduplication process also updates the *file index* with the newly added unique files' fingerprints and host addresses. Then, it calculates the slice fingerprints and creates a slice recipe for each slice. Slice recipe contains partial of the layer tarball's directory tree structure, file fingerprints, file name along with file path in the tarball, and file metadata information (such as permissions and creation date), which are needed for restoring a slice for the layer. Note that layer deduplication process only deduplicates regular files. In the end, deduplication process creates a layer recipe which contains its slices' fingerprints and based on layer recipe, it updates image manifests. The layer's tarball and file duplicates are removed after deduplication.

Parallel slice restoring When a `pull slice` request is received, the LRPA deduplication system first prepares a directory structure for the slice, based on the slice recipe. Then, it copies the files into the directory tree. Next, it compresses the slice's directory tree into a slice tarball, and directly sends it back to the client.

3.2.3 Layer restoring assisted file cache (LRA file cache)

Slice restoring process has four suboperations: slice recipe lookup, slice file copying, slice compression, and slice network transfer. To measure the overhead 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

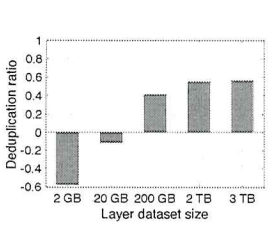


Figure 6: File-level deduplication vs. compression efficiency.

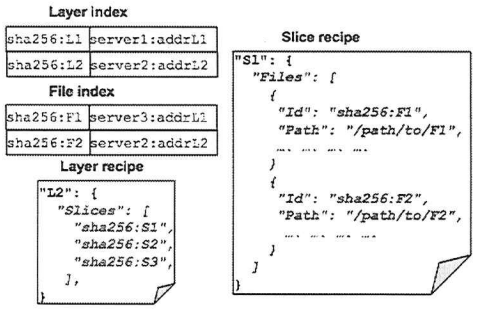


Figure 7: Metadata for deduplication.

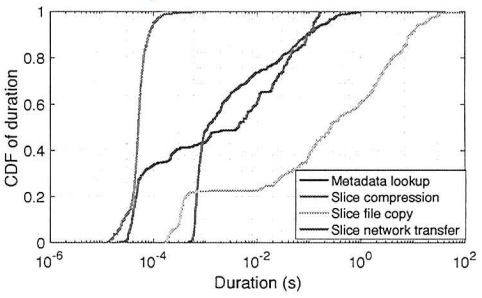


Figure 8: Breakdown of slice restoring time.

deduplication, we sent 400 `pull slice` requests to the cluster with 10 `pull slice` requests issued at a time. Figure 8 shows the CDFs of the latencies for 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 a smaller unit. 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. 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. The overhead of slice copying can be largely mitigated for a large-scale registry cluster since the size of slice roughly equals to S_l/N, where S_l denotes the layer size and N is size of registry cluster. However, it could be a bottleneck for slice restoring on a small-scale registry cluster.

Algorithm 1: File cache assisted slice restoring

Input:
θ_{rsfc}: Slice restoring latency threshold.
s: Slice to be restored.

Function `Restore` (s) **:**
  if *files in s are cached in file cache* **then**
    slice ← `RestoreSlice` s `From` *file cache + disk*
  else
    slice, D_{rs} ← `RestoreSlice` s `From` *disk*
    if $D_{rs} > \theta_{rsfc}$ **then**
      file cache ← `Cache Subsetof` *s.files*

To reduce slice file copying overhead, Sift LRA file cache temporally caches a subset of unique files for bigger and popular slices that have a high slice restoring latency, ie., $D_{rs} > \theta_{rsfc}$, where D_{rs} is the slice restoring latency and θ_{rsfc} is the restoring latency threshold for caching a subset of files from the slice to help improve its restoring performance as shown in Algorithm 1. Upon a `pull slice` request for those slices, LRPA deduplication system fetches a subset of its containing files from LRA file cache and the remaining files from disk for slice restoring.

To identify which slices have a high slice restoring latency, LRPA deduplication system monitors slice restoring performance and maintains a restoring performance profile for each slice that has been restored, which contains the latency breakdown of slice restoring and its containing files' sizes. All the slice restoring performance profiles are also stored in distributed databases, and addressed by slice digests. To estimate the restoring latency for a slice i that hasn't been restored, LRPA deduplication system first lookups the slice restoring performance profiles by slice size, then selects a slice x that is most similar in size to i, and estimates i's restoring latency as: $D_{rs}(i) \approx D_{rs}(x) + \Phi_{rs}(\Delta_S)$, where Δ_S is the size different between two slices, $\Phi_{rs}(\Delta_S)$ denotes a slice restoring latency function of slice size variation. $\Phi_{rs}(\Delta_S)$ is generated by using linear regression [?]. If the estimated slice restoring $D_{rs}(i) > \theta_{rs}$, then, LRPA deduplication lookups the slice restoring performance profiles by slice size, selects a slice y that has a acceptable restoring latency and most similar in size to i. Next, LRPA deduplication system caches a subset of files F for slice i, so that $D_{rs}(i) - \Phi_{rs}(\Sigma_S(F)) \approx D_{rs}(y)$, where $\Sigma_S(F)$ is the sum size of files in F.

Note that LRA file cache size is limited so that LRA file cache only caches subsets of files for big slices that belongs to popular layers. § 3.3 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 deduplication will cache similar amount of files for its slices. Note that all the files in file cache are unique and can be shared for restoring different slices.

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 have a high decompression latency, i.e., $D_{stt} < \theta_{stt}$ and $D_{dc} > \theta_{dc}$,

where D_{stt} and D_{dc} denote slice transfer duration and decompression duration respectively; θ_{stt} and θ_{dc} denote thresholds for them respectively. Consequently, LRPA deduplication system only archives these slices without compressing them and directly sends these archival files back to the clients to eliminate clients' decompression latency.

3.3 User behavior based layer preconstruction cache

In the following, we present our Sift UBLP cache design.

3.3.1 User access pattern based preconstruction

Docker clients stores images as lists of layers and layers are shared among different repositories, which is similar to Docker registry. When a client pulls an image from a repository, it will first `pull` the manifest of the image [3] [7] and parse the manifest to get the layer digests, then lookup each layer digest against a *local layer index*. After that it only pulls the layers that have *not been stored locally*. Theoretically, clients only pull layer once. However, some clients may delete several local images and *repull* layers for these images. Here, a *repull layer* means same user pull this layer multiple times, and a *non-repull layer* means users only pull this layer once.

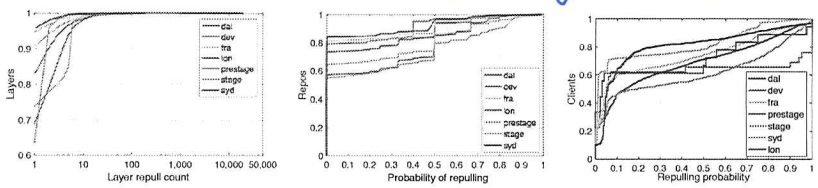
UBLP cache starts parallel slice restoring for layers when a `pull manifest` request is received, which is called layer preconstruction. These preconstructed layer slices are temporally stored in the cache for later `pull slice` requests. In this case, slice restoring process and its overhead can be avoided if the requested slice is found in the cache. Next, we analyze user access patterns to identify which layers in the repository will be pulled by users after `pull manifest` requests.

User access patterns Figure 9(a) shows the CDF of layer repull count. We see that majority of users don't *repull* layers frequently. For `Syd`, only 4% of layers are repulled by the same clients. `Dev` has the highest repull layer ratio of 36% while 83% of the repull layers are only repulled twice. Majority of repulled layers are repulled infrequently. For example, only 3% of layers from `Syd` are repulled more than twice. Layers from `Prestage` and `Lon` have the highest repull frequency. 5% of layers are pulled more than 6 times. We also observe that few clients *repull* layers continuously. The highest layer repull count is 19,300 from `Lon`. We think these clients probably deploy containers on a shared platform such as Cloud, and run ephemeral jobs such as stateless microservices. Once the applications are finished, the container images are automatically deleted, so when users launch containers again, they will repull layers again.

When different clients `pull` the same repository, they will fetch different amount of layers from the repository based on the availability of their local layer dataset. Even the same clients `pull` the same repository at different times, they will fetch different amount of layers from the repository because their local layer dataset changes over time. Therefore, a `pull manifest` requests doesn't usually result in repulling the layers in the repository. Here, we define *repulling repository* as repulling the layers in the repository for the same client. Figure 9(b) shows the CDF of the probability of repository repulling. The probability of repository repulling is calculated as the number of `pull manifest` resulting in repository repulling divided by the total number of `pull manifest` requests issued by the same client for the same repository. We see that majority of repositories aren't repulled. The repull repository ratio ranges from 15% for `Prestage` to 43% for `Prestage`. Majority of repull repositories have a low repulling probability. Only 20% of repositories from `Prestage`, `Stage`, and `Syd` have a repulling probability higher than 0.5. And only 20% of repositories from the rest 4 workloads have a repulling probability higher than 0.33. We also observe that few repositories' repulling probability are 1, meaning every time clients pull these repositories, they always repull the layers in these repositories.

Figure 9(c) shows the client repulling probability. Client repulling probability is calculated as the number of *repull* layer requests divided by the number of `pull` layer requests issued by the same client. We see that majority of clients do repull layers but the probability is low. 60% of clients from `Prestage`, `Dev`, `Lon`, and `Fra` have a repulling probability lower than 0.1. 55% of clients from both `Dal` and `Stage` have a repulling probability lower than 0.1. Less clients have repulling probability range between 0.1 to 0.7. 10%-30% of clients have a repulling probability ranged from 0.2-0.7 across 7 workloads. We find few clients repull layers continuously. 2%-12% of clients have a repulling probability higher than 0.9 from workloads: `Dal`, `Dev`, `Fra`, `Prestage`, `Stage`, `Syd`, and `Lon`.

Monitor user access patterns To monitor user access patterns, UBLP cache first uses a RLmap to record repository-layer relationship. If user *u pushes* a layer *l* to a repository *r*, UBLP cache will add a new entry (*l*) in RLmap denoted as RLmap[*r, l*]. UBLP cache also maintains a URLmap for keeping track of user access patterns. To identify a user, we extract *user end host address* (*r.client*) from each request (*r*). Each URLmap entry maintains a user profile for each user *u* denoted as URLmap[*u*]. User profile contains a list of repository profiles for each accessed repository denoted



(a) Layer repull count (b) Repository repulling probability (c) Client repulling probability

Figure 9: PDF of client repull count, repository repulling probability, and client repulling probability

as URLmap[*u*, *repo*], and each repo profile contains a list of layer profiles for each accessed layer *l* denoted as URLmap[*u*, *repo*, *l*]. Note that layer profiles can be shared among different repo profiles for the same user. If user *u* `pulls` a layer *l* from a repository *repo*, UBLP cache will update layer profile URLmap[*u*, *repo*, *l*] with the corresponding layer repull count, and calculate repository repulling probability and user repulling probability for the corresponding repository profile and user profile.

Algorithm 2: Preconstruction

Input:
θ_{rpc}: Threshold for repull layers to be preconstructed and cached.
$RLmap$: Repository to layers map.
$URLmap$: User to user profile map.

```
r ← request received
if r = GET manifest then
    newLayers ← RLmap[r.repo] − URLmap[r.client].layers
    oldLayers ← RLmap[r.repo] ⋂ URLmap[r.client].layers
    cache ← Restore newLayers.slices
    if URLmap[r.client, r.repo, oldLayers].rp > θ_rpc then
        cache ← Restore oldLayers.slices
else if r = PUT layer then
    cache ← cache r.layer
    update RLmap[r.repo, r.layer]
else if r = GET layer then
    /* r.layer has not been deduplicated */
    if r.layer in cache then
        cache hit for layer
    else
        cache ← Fetch r.layer
else if r = GET slice then
    /* r.layer has already been deduplicated */
    if r.slice in cache then
        cache hit for slice
    else
        cache ← Restore r.slice
update URLmap[r.client, r.repo, r.layer]
set Timer[r.client, r.repo, layer]
```

Preconstruction algorithm Algorithm 2 shows when to preconstruct slices for a layer based on observed user access pattern. When a `GET manifest` request *r* is received, UBLP cache looks up the requested repository *r.repo*'s layers from RLmap and gets a list of corresponding layers. After that, it compares against the layers looked up from URLmap[*r*] (denoted as URLmap[*r*].layers) and gets two groups of layers: *newLayers* and *oldLayers*. *newLayers* means the layers that belongs to *r.repo* but haven't been pulled by client *r.client*. While *oldLayers* means the layers that belongs to both of them. UBLP cache will first restore the slices for *newLayers* because they are not locally available to *r.client*. For *oldLayers*, if an *oldLayer* has a higher repull count and *r.repo* as well as *r.client* have a higher repulling probability, then UBLP cache will restore its slices and cache them. If a *PUT* layer request is received, RLmap and URLmap will be updated accordingly.

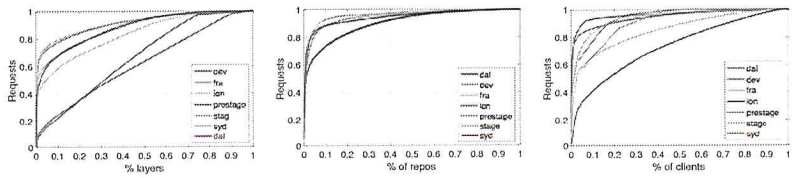
If a *GET* layer request is received, it means that *r.layer* has not been deduplicated. UBLP cache will cache *r.layer* if cache miss happens on *GET* layer request as shown in Algorithm 2. If a *GET* slice request is received, meaning that *r.layer* has already been deduplicated into deduplicated slices, UBLP cache will check if the requested *r.slice* exists in the cache. If not, LRPA deduplication system will start restoring *r.slice* and also put it in cache. In the end, UBLP cache will update URLmap with corresponding repull count and repull probability.

3.3.2 User access pattern based cache replacement

UBLP cache starts cache eviction when free space is low. To decide which layer or slices need to be evicted to make space for new requests, we analyze the temporal trend of user accesses as follows.

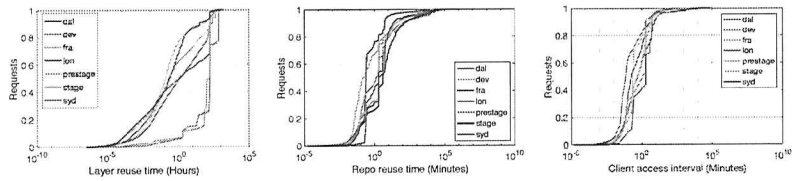
Temporal trend Figure 10(a) shows the CDF of layer popularity. We observe a heavy layer access skewness for `Fra`, `Syd`, `Dal`, `Stage`, and `Lon`. We see that 80%, 70%, and 60% of the `pull` layer requests access only 10% of layers, for `Fra`, `Syd`, `Dal`, `Stage`, and `Lon` respectively. Figure 10(b) shows the CDF of repository popularity. Compare to layer popularity, repository access skewness is heavier acorss 7 workloads. Almost 90% of `pull` layer requests access only 10% of repositories for `Dev`, `Fra`, `Prestage`, `Syd`, and `Stage` respectively. Almost 75% of `pull` layer requests access only 10% of repositories for both `Dal` and `Lon`. Figure 10(c) shows the CDF of client popularity. `Dal`, `Dev`, `Fra`, `Lon`, `Prestage`, and `Stage` shows a heavy client access skewness. 10% of clients send 95% `pull` layer requests for `Lon`. `Syd` shows a slight client skewness. 70% of requests are sent by 36% of clients. Overall, caching a layer with higher pull count will improve the hit ratio, especially for popular repositories and active clients with higher repulling probability.

Next, we analyze the layer and repository reuse time.



(a) Layer repull count (b) Repository repulling probability (c) Client repulling probability

Figure 10: PDF of probability for layers, repositories, and clients.



(a) Layer repull count (b) Repository repulling probability (c) Client repulling probability

Figure 11: CDF of reusetime for layers, repositories and clients' access intervals.

layer reuse time means the duration between two consecutive requests to the same layer while repository reuse time means the duration between two consecutive *pull* manifest requests to the same repository. Figure 11(a) shows the CDF of layer reuse time. We see that layer reuse time distribution varies among different workloads. For `Fra`, `Syd`, and `Stage`, half of the layers' reuse time is shorter than 6 minutes. While half of layers from `Dal` and `Lon` have a reuse time higher than 1 hour. Half of layers from both `Prestage` and `Dev` are not accessed for over 100 hours. Consequently, for `Dal`, `Lon`, `Prestage`, and `Dev`, it may take longer than 1 hour for at lease half newly requested layers to get a hit. These layers or the slices for them are unnecessary for caching since their reuse time is too long and may cause other useful layers or slices to be evicted, called *cache pollution*. Figure 11(b) shows the CDF of repository reuse time. We see that repository reuse time is much shorter than layer reuse time. 80% of repositories are requested within 2-12 minutes across the 7 workloads. Figure 11(c) shows the CDF of client access intervals. Client access interval means the duration between two consective requests issued by the same client. Client access intervals are much shorter than repository reuse time. 80% of client are active within 1 - 3 minutes for the 7 workloads. Hence, to eliminate *cache pollution*, we consider the reuse time of layer and repository as well as client access intervals during cache eviction.

Eviction algorithm As shown in algorithm 3, to exploit the temporal trend of clients and repositories, UBLP cache set timers for its hosted layers, clients, and repositories. Timeout happens when a layer along with its associated clients and repositories are all timed out. UBLP cache will remove this layer from cache. Beside, UBLP cache maintains a LFRU list of cached layers [**?**] to exploit layer temporal trend. If free space is low, UBLP cache selects the least frequent recently used layer which has a lower repulling probability to evict.

Algorithm 3: Eviction

Input:
θ_C: Capacity threshold for cache to trigger replacement.
θ_{nrpc}: Threshold for non-repull layer to be evicted.
$LFRUlayer$: LFRU of layers.
$URLmap$: User to user profile map.
$Timer$: Timer for clients, repos, and layers in cache.

```
foreach layer in LFRUlayer do
    /*Timeout eviction*/
    if Timeout == Timer[clients, repos, layer] then
        Evict layer
while freeSpace < θ_C do
    /*Replacement*/
    layer ← LFRUlayer.last_item()
    if URLmap[clients, repos, layer].rp < θ_rpc then
        Evict layer
        freeSpace += sizeof layer
```

4 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.

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

5 Evaluation

5.1 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.

5.2 Workloads and dataset

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

5.3 Deduplication evaluation

5.3.1 Pull performance

5.3.2 Push performance

5.3.3 Deduplication performance

6 Preliminary Evaluation

Cache hit ratio. We simulate our user-access-history-based cache and replay the `dal` workload [7] 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 **??** 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 6, 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.

6.1 Cache evaluation

7 Related Work

A number of studies investigated various dimensions of Docker storage performance [8, 9, 10, 18, 22, 25, 23, 10]. Anwar et al. [7] performed a detailed analysis of an IBM Docker registry workload but not the dataset. Data deduplication is a well explored and widely applied technique [12, 16, 19, 21, 27]. A number of studies which focus on real-world datasets [11, 13, 14, 17, 20, 24, 26] can be complementary to our approach.

8 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.

9 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] Storreduce. http://storreduce.com.

[7] 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*.

[8] 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*.

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

[10] 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*.

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

[12] 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*.

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

[14] 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*.

[15] 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.

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

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

[18] 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*.

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

[20] 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*.

[21] 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*.

[22] 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*.

[23] 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.

[24] 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*.

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

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

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