DADI: Block-Level Image Service for Agile and Elastic Application Deployment

Huiba Li, Yifan Yuan, Rui Du, Kai Ma, Lanzheng Liu, and Windsor Hsu,
Alibaba Group

https://www.usenix.org/conference/atc20/presentation/li-huiba

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

July 15–17, 2020

978-1-939133-14-4

DADI Block-Level Image Service for Agile and Elastic Application Deployment

Huiba Li, Yifan Yuan, Rui Du, Kai Ma, Lanzheng Liu and Windsor Hsu
Alibaba Group

Abstract

Businesses increasingly need agile and elastic computing infrastructure to respond quickly to real world situations. By offering efficient process-based virtualization and a layered image system, containers are designed to enable agile and elastic application deployment. However, creating or updating large container clusters is still slow due to the image downloading and unpacking process. In this paper, we present DADI Image Service, a block-level image service for increased agility and elasticity in deploying applications. DADI replaces the waterfall model of starting containers (downloading image, unpacking image, starting container) with fine-grained on-demand transfer of remote images, realizing instant start of containers. DADI optionally relies on a peer-to-peer architecture in large clusters to balance network traffic among all the participating hosts. DADI efficiently supports various kinds of runtimes including cgroups, QEMU, etc., further realizing "build once, run anywhere". DADI has been deployed at scale in the production environment of Alibaba, serving one of the world's largest ecommerce platforms. Performance results show that DADI can cold start 10,000 containers on 1,000 hosts within 4 seconds.

1 Introduction

As business velocity continues to rise, businesses increasingly need to quickly deploy applications, handle unexpected surge, fix security flaws, and respond to various real world situations. By offering efficient process-based virtualization and a layered image system, containers are designed to enable agile and elastic application deployment. However, creating or updating large container clusters is still slow due to the image downloading and unpacking process. For example, Verma et al. in [37] report that the startup latency of containers is highly variable with a typical median of about 25s, and pulling layers (packages) accounts for about 80% of the total time.

Highly elastic container deployment has also become expected of modern cloud computing platforms. In serverless computing [23], high cold-start latency could violate responsiveness SLAs. Workarounds for the slow start are cumbersome and expensive, and include storing all images on all possible hosts. Therefore, minimizing cold-start latency is considered a critical system-level challenge for serverless computing [14, 23].

There has been recent work on reducing container startup time by accelerating the image downloading process with a peer-to-peer (P2P) approach [20, 22, 24, 30, 37]. We relied on a P2P download tool for several years to cope with the scalability problem of the Container Registry. However, the startup latency was still unsatisfactory. Another general approach to the problem is to read data on-demand from remote images [9, 16, 18, 19, 21, 33, 41]. Because container images are organized as overlaid layers of files and are presented to the container runtime as a file system directory, all of the previous work adhered to the file system interface, even though some of them actually used block stores as their backends.

Implementing a POSIX-complaint file system interface and exposing it via the OS kernel is relatively complex. Moreover, using file-based layers has several disadvantages. First, updating big files (or their attributes) is slow because the system has to copy whole files to the writable layer before performing the update operations. Second, creating hard links is similarly slow, because it also triggers the copy action as cross layer references are not supported by the image. Third, files may have a rich set of types, attributes, and extended attributes that are not consistently supported on all platforms. Moreover, even on one platform, support for capabilities such as hard links, sparse files, etc. tends to be inconsistent across file systems.

With the rapid growth of users running containers on public cloud and hybrid cloud, virtualized secure containers are becoming mainstream. Although it is possible to pass a file-based image from host to guest via 9p [1] or virtio-fs [10], there is usually a performance cost. There are also complications in handling heterogeneous containers such as Windows guest on Linux host, or vice versa. This means that some users may not be able to burst efficiently to public clouds, i.e. run their applications primarily on premise with an efficient container runtime, and scale them out under load to public clouds with a virtualized secure container runtime.

In this paper, we observe that the benefits of a layered image are not contingent on representing the layers as sets of file changes. More specifically, we can achieve the same effect with block-based layers where each layer still corresponds to a set of file changes but is physically the set of changes at the block level underneath a given file system. Such a design allows the image service to be file system and platform agnostic. The image service is solely responsible for managing and distributing physical images to the appropriate hosts. It is up to the individual host or container on the host to interpret

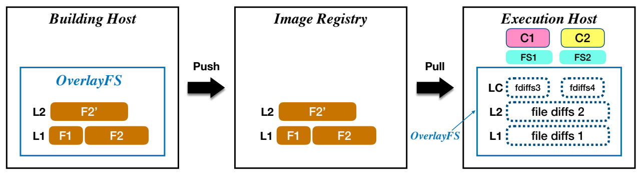

Figure 1: Layered Container Image. The image layers (L1, L2) are read-only shared by multiple containers (C1, C2) while the container layers (LC) are privately writable.

the image with an appropriate file system. This approach also allows dependency on the file system to be explicitly captured at image creation time, further enhancing consistency in the runtime environment of applications.

We have designed and implemented a complete system called *DADI Image Service* (*DADI* in short) based on this approach. The name *DADI* is an acronym for *Data Acceleration for Disaggregated Infrastructure* and describes several of our initiatives in enabling a disaggregated infrastructure. At the heart of the DADI Image Service is a new construct called Overlay Block Device (OverlayBD) which provides a merged view of a sequence of block-based layers. Conceptually, it can be seen as the counterpart of union file systems that are usually used to merge container images today. It is simpler than union file systems and this simplicity enables optimizations including flattening of the layers to avoid performance degradation for containers with many layers. More generally, the simplicity of block-based layers facilitates (1) fine-grained on-demand data transfer of remote images; (2) online decompression with efficient codecs; (3) trace-based prefetching; (4) peer-to-peer transfer to handle burst workload; (5) flexible choice of guest file systems and host systems; (6) efficient modification of large files (cross layer block references); (7) easy integration with the container ecosystem.

We have applied DADI to both cgroups [3] runtime and QEMU runtime. Support for other runtimes such as Firecracker [13], gVisor [5], OS^v [25], etc. should be technically straightforward. DADI has been deployed at scale in the production environment of Alibaba to serve one of the world's largest ecommerce platforms. Performance results show that DADI can cold start 10,000 containers on 1,000 hosts within 4 seconds. We are currently working on an edition of DADI for our public cloud service.

2 Background and Related Work

2.1 Container Image

Container images are composed of multiple incremental layers so as to enable incremental image distribution. Each layer is essentially a tarball of differences (addition, deletion or update of files) from a previous layer. The container system may apply the diffs in a way defined by its storage driver. The layers are usually much lighter than VM images that contain full data. Common layers are downloaded only once on a host, and are shared by multiple containers as needed. Each container has a dedicated writable layer (also known as container layer) that stores a private diff to the image, as shown in Figure 1. Writing to a file in the image may trigger a copy-on-write (CoW) operation to copy the entire file to the writable layer.

To provide a root file system to containers, the container engine usually depends on a union file system such as overlayfs, aufs, etc. These union file systems provide a merged view of the layers which are stored physically in different directories. The container system can also make use of Logical Volume Manager (LVM) thin-provisioned volumes, with each layer mapped to a snapshot.

The container system has a standard web service for image uploading and downloading called the Container Registry. The Container Registry serves images with an HTTP(S)-based protocol which, together with the incremental nature of layers, makes it a lot easier to distribute container images widely as compared to VM images.

2.2 Remote Image

The image distribution operation, however, consumes a lot of network and file system resources, and may easily saturate the service capacity allocated to the user/tenant, especially when creating or updating large container clusters. The result is long startup latencies for containers. After an image layer is received, it has to be unpacked. This unpacking operation is CPU, memory (for page cache) and I/O intensive at the same time so that it often affects other containers on the host and sometimes even stalls them.

To some extent, the current container image service is a regression to a decade ago when VM images were also downloaded to hosts. A similar problem has been solved once with distributed block stores [26, 28, 29, 38] where images are stored on remote servers, and image data is fetched over the network on-demand in a fine-grained manner rather than downloaded as a whole. This model is referred to as "remote image". There are several calls for this model in the container world (e.g. [18, 21]).

The rationale for remote image is that only part of the image is actually needed during the typical life-cycle of a container, and the part needed during the startup stage is even smaller. According to [19], as little as 6.4% of the image is used during the startup stage. Thus remote image saves a lot of time and resources by not staging the entire image in advance. And with the help of data prefetching (by OS) or asynchronous data loading (by applications themselves), the perceived time to start from a remote image can be effectively reduced further.

Remote image, however, requires random read access to the contents of the layers. But the standard layer tarball was designed for sequential reading and unpacking, and does not support random reading. Thus the format has to be changed.

2.3 File-System-Based Remote Image

CRFS [21] is a read-only file system that can mount a container image directly from a Container Registry. CRFS introduces an improved format called Stargz that supports random reads. Stargz is a valid tar.gz format but existing images need to be converted to realize remote image service. Instead of having the read-only file system read files directly from the layer, one could also extract the files in each layer and store them in a repository such as CernVM-FS [18] where they can be accessed on demand. CFS [27] is a distributed file system to serve unpacked layer files for hosts. Wharf [41], Slacker [19], Teleport [9] serve unpacked layer files through NFS or CIFS/SMB.

Due to the complexity of file system semantics, there are several challenges with file-system based image service. For example, passing a file system from host to guest across the virtualization boundary tends to limit performance. The I/O stack involves several complex pieces (including virtio-fs [10], FUSE [36], overlayfs [8], remote image itself) that need to be made robust and optimized. When compared to a block device, the file system also presents a larger attack surface that potentially reduces security in public clouds.

POSIX-compliant features are also a burden for non-POSIX workloads such as serverless applications, and an obstacle to Windows workloads running on Linux hosts (with virtualization). In addition, unikernel [5, 25, 39] based applications are usually highly specialized, and tend to prefer a minimalistic file system such as FAT [4] for efficiency. Some of them may even require a read-only file system. These different requirements are difficult to be satisfied by a file system that is predefined by the image service.

Furthermore, some desirable features of popular file systems such as XFS [11], Btrfs [2], ZFS [12], etc., are missing in current file-system-based image services, and are not likely to be supported soon. These include file-level or directory-level snapshot, deduplication, online defragmentation, etc. It is even difficult to efficiently (without copy) support standard features such as hard links and modification of files or file attributes.

2.4 Block-Snapshot-Based Remote Image

Modern block stores [26, 28, 29, 38] usually have a concept of copy-on-write snapshot which is similar to layer in the container world. Cider [16] and Slacker [19] are attempts to make use of such similarity by mapping image layers to the snapshots of Ceph and VMstore [17], respectively.

Container image layer and block store snapshot, however, are not identical concepts. Snapshot is a point-in-time view of a disk. Its implementation tends to be specific to the block store. In many systems, snapshots belong to disks. When a disk is deleted, its snapshots are deleted as well. Although this is not absolutely necessary, many block stores behave this way by design. Layer, on the other hand, refers to the incremental change relative to a state which can be that of a different image. Layer emphasizes sharing among images, even those belonging to different users, and has a standard format to facilitate wide distribution.

2.5 Others

File System Changesets. Exo-clones [33] implement volume clones efficiently with file system changesets that can be exported. DADI images are conceptually exo-clones with block level deltas that are not tied to any specific file system.

P2P downloading. Several systems allow container hosts to download image layers in a P2P manner, significantly reducing the download time in large environments [20, 22, 24, 30, 37]. VMThunder [40] adopts a tree-structured P2P overlay network to deliver fine-grained data blocks on-demand for large VM clusters. We reuse this general idea in DADI's optional P2P subsystem with a refined design and a production-level implementation.

Trimmed images. In order to pull less data and start a container in less time, DockerSlim [6] uses a combination of static and dynamic analyses to generate smaller-sized container images in which only files needed by the core application are included. Cntr [35] improves this by allowing dynamic accesses to trimmed files in uncommon cases via a FUSE-based virtual files system.

Storage Configuration for Containers. The layering feature of container image introduces new complexities in configuring storage. [34] demonstrates the impact of Docker storage configuration on performance.

VM images. Standard VM image formats such as qcow2, vmdk, vhd, etc. are block-level image formats and are technically reusable for containers. The major drawback of these image formats is that they are not layered. It is possible to emulate layering by repeatedly applying QEMU's backing-file feature, but doing this incurs significant performance overhead for reads. As we shall see in Section 3.1, the translation tables for standard VM image formats are also much bigger than those needed for DADI.

3 DADI Image Service

DADI is designed to be a general solution that can become part of the container ecosystem. The core of DADI (Sections 3.1-3.4) is a remote image design that inherits the layering model of container image, and remains compatible with the Registry by conforming to the OCI-Artifacts [31] standard. DADI is independent of transfer protocols so it is possible to insert an optional P2P transfer module to cope with large-scale applications (Section 3.5). DADI is also independent of the underlying storage system so users can choose an appropriate storage system such as HDFS, NFS, CIFS, etc., to form a fully networked solution (Section 4.4). DADI uses a block-level interface which minimizes attack surface, a design point especially relevant for virtualized secure containers.

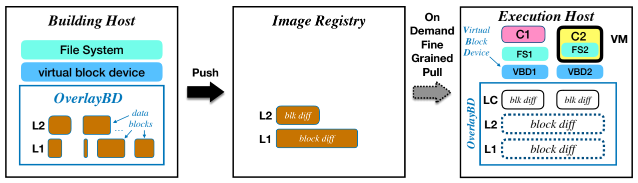

Figure 2: DADI Image. DADI image layer (L1, L2) consists of modified data blocks. DADI uses an overlay block device to provide each container (C1, C2) with a merged view of its layers.

3.1 DADI Image

As shown in Figure 2, DADI models an image as a virtual block device on which is laid a regular file system such as ext4. Note that there is no concept of file at the block level. The file system is a higher level of abstraction atop the DADI image. When a guest application reads a file, the request is first handled by the regular file system which translates the request into one or more reads of the virtual block device. The block read request is forwarded to a DADI module in user space, and then translated into one or more random reads of the layers.

DADI models an image as a virtual block device while retaining the layered feature. Each DADI layer is a collection of modified data blocks under the filesystem and corresponding to the files added, modified or deleted by the layer. DADI provides the container engine with merged views of the layers by an overlay block device (OverlayBD) module. We will use layer and changeset interchangeably in the rest of the paper for clearer statements. The block size (granularity) for reading and writing is 512 bytes in DADI, similar to real block devices. The rule for overlaying changesets is simple: for any block, the latest change takes effect. The blocks that are not changed (written) in any layer are treated as all-zero blocks.

The raw data written by the user, together with an index to the raw data, constitutes the layer blob. The DADI layer blob format further includes a header and a trailer. To reduce memory footprint and increase deployment density, we design an index based on variable-length segments, as illustrated in

```
struct Segment {
  uint64_t offset:48; // offset in image's LBA
  uint16_t length;    // length of the change
  uint64_t moffset:48;// mapped offset in layer blob
  uint16_t pos:12;    // position in the layer stack
  uint8_t flags:4;    // zeroed? etc.
  uint64_t end() {return offset + length;}
};
```

Figure 3: Definition of Segment. LBA is short for logical block address, offsets and lengths are in unit of blocks (512 bytes), size of the struct is 16 bytes.

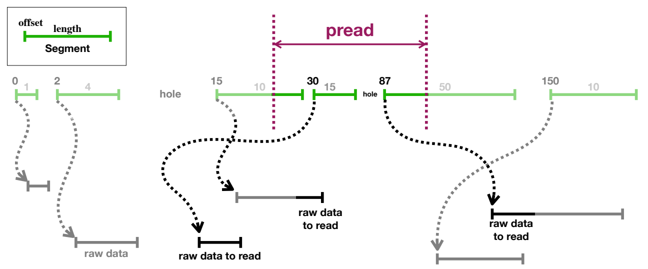

Figure 4: Index Lookup for Reading DADI Image. The lookup operation is a range query on a set of ordered non-overlapping variable-length segments, each of which points to the location of its raw data in the layer blob(s).

Figure 3. A segment tells where a change begins and ends in the image's logical block address (LBA) space, and also where the latest data is stored in the layer blob file's offset space. In this design, adjacent segments that are contiguous can be merged into a single larger segment to reduce the index size. The segment struct can record a change as small as 1 block which is the minimal write size for a block device. This avoids Copy-on-Write operations and helps to yield consistent write performance.

The index is an array of non-overlapping segments sorted by their offset. According to statistics from our production environment, the indices have fewer than 4.5K segments (see Section 5 for details) which corresponds to only 72KB of memory. In contrast, the qcow2 image format of QEMU has a fixed block size of 64KB by default, and an index based on radix-tree. QEMU allocates MBs of memory by default to cache the hottest part of its index.

To realize reading, DADI performs a range lookup in the index to find out where in the blob to read. The problem can be formally stated as given a set of disjoint segments in the LBA space, find all the segments (and "holes") within the range to read. This problem is depicted in Figure 4. For efficiency, the algorithm deals with variable-length segments directly without expanding them to fixed-sized blocks. As the index is ordered and read-only, we simply use binary search for efficient lookup, as shown in Algorithm 1. A B-tree could achieve higher efficiency but as the index contains only a few thousand entries in practice, we leave this optimization as a possible future work.

3.2 Merged View of Layers

When there are multiple layers, if the lookup procedure goes through the layers one by one, the time complexity is $O(n \cdot \log m)$ where n is the number of layers and m is the average number of segments in a layer. In other words, the cost increases linearly with n. We optimize this problem with a merged index that is pre-calculated when the indices are

```
Input: the range (offset, length) to look up
end ← offset + length;
i ← index.binary_search_first_not_less(offset);
if i < index.size() then
    delta ← offset - index[i].offset;
    if delta > 0 then  // trim & yield 1st segment
        s ← index[i]; s.offset ← offset;
        s.moffset += delta; s.length -= delta;
        yield s; offset ← s.end(); i++;
    end
end
while i < index.size() and index[i].offset < end do
    len ← index[i].offset - offset;
    if len > 0 then                // yield a hole
        yield Hole(offset, len);
        offset ← index[i].offset;
    end
    s ← index[i];             // yield next segment
    s.length ← min(s.length, end - offset);
    yield s; offset ← s.end(); i++;
end
if offset < end then             // yield final hole
    yield Hole(offset, end - begin);
end
```

Algorithm 1: Index Lookup. Yields a collection of segments within the specified range (offset, length) with `i` initialized to the first element in the index that is not less than `offset`, and Hole being a special type of segment representing a range that has never been written.

```
Input: an array of indices[1..n];
       subscript i of the indices array for this recursion;
       the range to merge (offset, length)
for s in indices[i].lookup(offset, length) do
    if s is NOT a Hole then
        s.pos ← i;
        yield s;
    else if i > 0 then // ignore a real hole
        indices_merge(indices, i-1, s.offset, s.length);
    end
end
```

Algorithm 2: Index Merge by Recursion.

```
Input: an array of file objects blobs[0..n];
       a rage (offset, length) to pread
for s in merged_index.lookup(offset, length) do
    // s.pos == 0 for Hole segments
    // blobs[0] is a special virtual file object
    // that yields zeroed content when pread
    blobs[s.pos].pread(s.offset, s.length);
end
```

Algorithm 3: Read Based on Merged Index.

loaded, thus reducing the complexity to $O(\log M)$ where M is the number of segments in the merged index. The merging problem is illustrated in Figure 5.

To merge the indices, we put them in an array indexed from 1 to n where n is the number of layers, and in an order such that base layers come earlier. Algorithm 2 shows the recursive procedure to merge indices for a specified range. To merge them as whole, the algorithm is invoked for the entire range of the image. We make use of the `pos` field in the final merged index to indicate which layer a segment comes from. With the merged index, random read operation (pread) can be easily implemented as Algorithm 3, supposing that we have an array of file objects representing the ordered layers' blobs.

We analyzed 1,664 DADI image layers from 205 core applications in our production environment to extract the size of the merged indices. The statistics are summarized in Figure 6. They show that the indices have no more than 4.5K segments so the algorithm for merging indices is efficient enough to be run when an image is launched. Observe also that the number of segments is not correlated with the number of layers. This suggests that the performance of DADI OverlayBD does not degrade as the number of layers increases. Figure 7 plots the throughput of index queries on a single CPU core. Observe that at an index size of 4.5K segments, a single CPU core can perform more than 6 million index queries per second. In Section 5.4, we find that IOPS tops out at just under 120K for both LVM and DADI, suggesting that DADI spends no more than 1/50 of a CPU core performing index lookups.

3.3 Compression and Online Decompression

Standard compression file formats such as gz, bz2, xz, etc., do not support efficient random read operation. Files in these formats usually need to be decompressed from the very beginning until the specified part is reached. To support compression of the layers' blobs and enable remote image at the same time, DADI introduces a new compression file format called ZFile.

ZFile includes the source file compressed in a fixed-sized chunk-by-chunk manner and a compressed index. To read

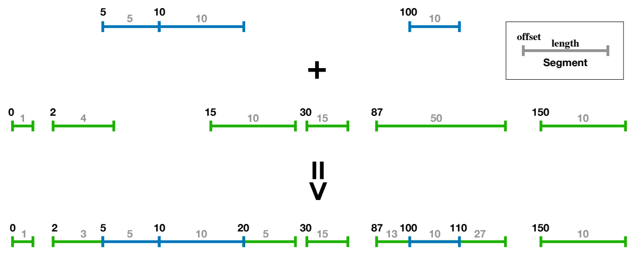

Figure 5: Index Merge.

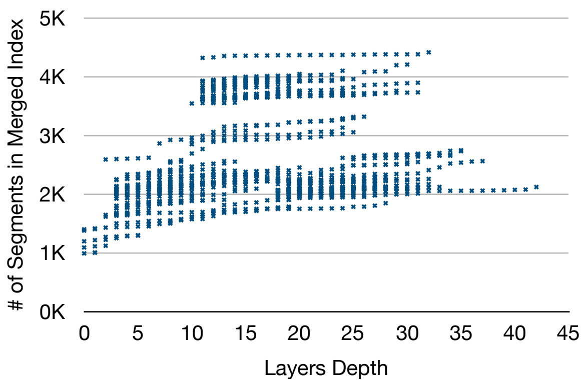

Figure 6: Index Size of Production Applications.

an offset into a ZFile, one looks up the index to find the offset and length of the corresponding compressed chunk(s), and decompresses only these chunks. ZFile supports various efficient compression algorithms including lz4, zstd, gzip, etc., and can additionally store a dictionary to assist some compression algorithms to achieve higher compression ratio and efficiency. Figure 8 illustrates the format of ZFile.

The index stored in ZFile is an array of 32-bit integers, each of which denotes the size of the corresponding compressed chunk. The index is compressed with the same compression algorithm as the data chunks. When loaded into memory, the index is decompressed and accumulated into an array of 64-bit integers denoting the offsets of the compressed chunks in the ZFile blob. After the conversion, index lookup becomes a simple array addressing at $offset/chunk\_size$.

Due to the fixed-size nature of chunks and the aligned nature of the underlying storage device, ZFile may read and decompress more data than requested by a user read. The decompression itself is an extra cost compared to the conventional I/O stack. In practice, however, ZFile improves user-perceived I/O performance even on servers with high-speed NVMe SSD. The advantage is even larger for slower storage

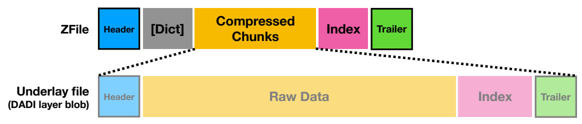

Figure 8: ZFile Format.

(e.g. HDD or Registry). This is because, with the compression agorithm in use (lz4), the time saved reading the smaller amount of compressed data more than offsets the time spent decompressing the data. See Section 5.4 for detailed results.

In order to support online decompression, a fast compression algorithm can be used at some expense to the compression ratio. We typically use lz4 in our deployment. Individually compressing chunks of the original files also impacts the compression ratio. As a result, DADI images are usually larger than the corresponding .tgz images but not by much.

We analyzed the blob sizes of 205 core applications in our production environment. Figure 9 shows the blob sizes in various formats relative to their .tar format sizes. In general, DADI uncompressed format (.dadi) produces larger blobs than .tar, due to the overhead of the image file system (ext4 in this case) but the overhead is usually less than 5% for layers that are larger than 10MB. Note that the compression ratio varies greatly among these images and some of them are not compressible. As discussed, ZFile blobs tend to be larger than their .tgz counterparts.

By adhering to the layered model of container image, DADI images are able to share layers. To further save space and network traffic, deduplication can be performed at the chunk level of DADI images, followed by compression of the unique chunks.

3.4 DADI Container Layer

Unlike other remote image systems (e.g. [18, 21]), DADI realizes a writable container layer. The writable layer is not only a convenient way to build new image layers, but also

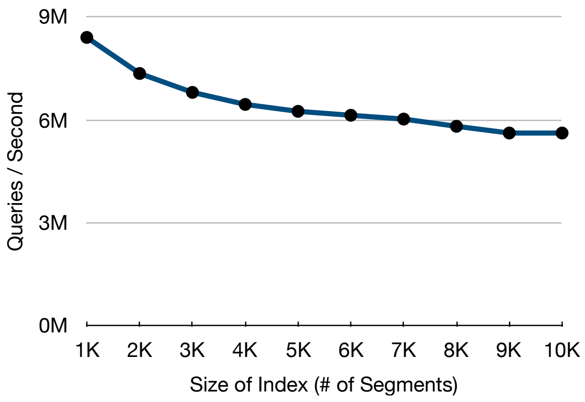

Figure 7: Index Performance on Single CPU Core.

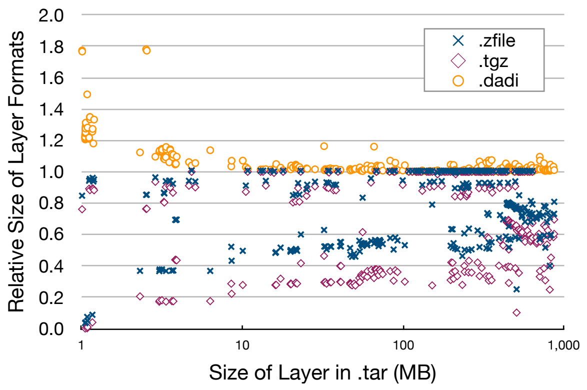

Figure 9: Relative Layer Blob Size.

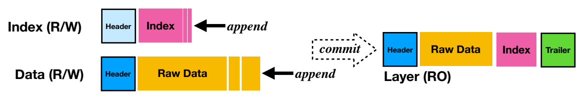

Figure 10: DADI's Writable Layer.

provides an option to eliminate the dependency on a union file system. We base the writable layer on a log-structured [32] design because this makes DADI viable on top of virtually all kinds of storage systems, including those that do not support random writes (e.g. HDFS). The log-structured writable layer is also technically a natural extension of the read-only layers.

As shown in Figure 10, the writable layer consists of one file for raw data and one for index. Both of these files are open-ended, and are ready to accept appends. As overwrites occur in the writable layer, these files will contain garbage data and index records. When there is too much garbage, DADI will spawn a background thread to collect the garbage by copying the live data to a new file, and then deleting the old file. When the writable layer is committed, DADI will copy the live data blocks and index records to a new file in layer format, sorting and possibly combining them according to their LBAs.

The index for the writable layer is maintained in memory as a red-black tree to efficiently support lookup, insertion and deletion. On a write, DADI adds a new record to the index of the writable layer. On a read, DADI first looks up the index of the writable layer. For each hole (a segment with no data written) within the range to read, DADI further looks up the merged index of the underlying read-only layers. DADI supports `TRIM` by adding to the writable layer an index record that is flagged to indicate that the range contains all-zero content.

3.5 P2P Data Transfer

Although remote image can greatly reduce the amount of image data that has to be transferred, there are situations where more improvement is necessary. In particular, there are several critical applications in our production environment that are deployed on thousands of servers, and that comprise layers as large as several GBs. The deployment of these applications places huge pressure on the the Registry and the network infrastructure.

To better handle such large applications, DADI caches recently used data blocks on the local disk(s) of each host. DADI also has the option to transfer data directly among the hosts in a peer-to-peer manner. Given that all the peers need roughly the same set of data and in roughly the same order during the startup time period, DADI adopts a tree-structured overlay topology to realize application-level multicast similar to VMThunder [40] instead of the rarest-first policy commonly used in P2P downloading tools [15, 20, 22].

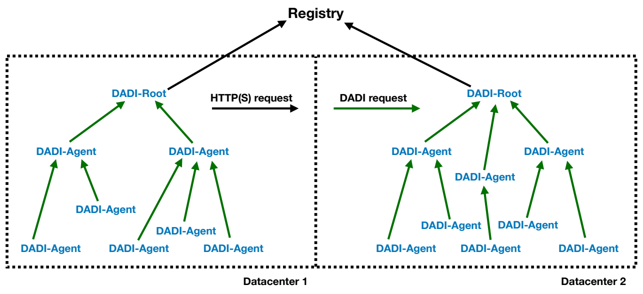

Figure 11: DADI's Tree-Structured P2P Data Transfer.

As shown in Figure 11, each container host runs a P2P module called DADI-Agent. In addition, there is a P2P module called DADI-Root in every data center that plays the role of root for topology trees. DADI-Root is responsible for fetching data blocks from the Registry to a local persistent cache, and also for managing the topology trees within its own datacenter. A separate tree is created and maintained for each layer blob.

Whenever an agent wants to read some data from a blob for the first time or when its parent node does not respond, it sends a request RPC to the root. The root may service the request by itself, or it may choose to rearrange the topology and redirect the requesting agent to a selected parent. The requesting agent is considered to join the tree as a child of the selected parent. Every node in a tree, including the root, serves at most a few direct children. If the requested data is not present in the parent's cache, the request flows upward until a parent has the data in its cache. The data received from the parent is added to the child's cache as it will probably be needed soon by other children or the node itself.

DADI-Root manages the topology. It knows how many children every node has. When a node needs to be inserted into the tree, the root simply walks down the tree in memory, always choosing a child with the fewest children. The walk stops at the first node with fewer direct children than a threshold. This node becomes the selected parent for the requesting agent. When a node finds that its parent has failed, it reverts to the root to arrange another parent for it. As the P2P transfer is designed to support the startup of containers and this startup process usually does not last long, DADI-Root expires topology information relatively quickly, by default after 20 minutes.

DADI-Root is actually a replicated service running on several servers for availability, and deployed separately for different clusters. An agent randomly chooses a root server in the same cluster when joining a transfer topology. It switches to another root server when it encounters a failure. The Registry tends to be shared by many clusters and possibly across a long distance so its performance may not always be high. In order to ensure that data blocks are likely to exist on the root when they are needed, we warm up the root servers' cache in

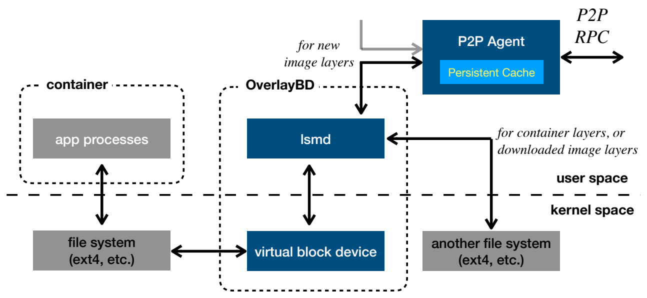

Figure 12: I/O Path for cgroups Runtime.

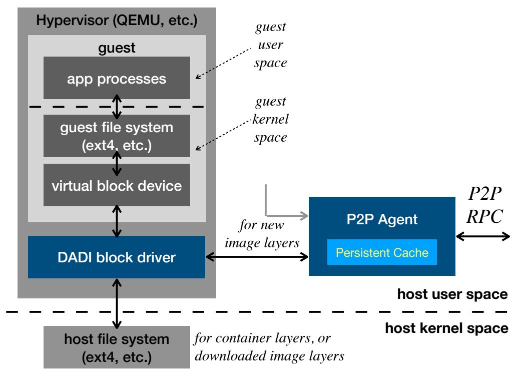

Figure 13: I/O Path for Virtualized Runtime (QEMU, etc).

our production environment whenever a new layer is built or converted.

To protect against potential data corruption, we create a separate checksum file for each and every layer blob as part of the image building or conversion process. The checksum file contains the CRC32 value for each fixed-sized block of the layer. As the checksum files are small, they are distributed whole to every involved node as part of the image pulling process. The data blocks are verified on arrival at each node.

4 Implementation and Deployment

This section discusses how DADI interfaces with applications and container engines, as well as how DADI can be deployed in different user scenarios.

4.1 Data Path

DADI connects with applications through a file system mounted on a virtual block device. DADI is agnostic to the choice of file system so users can select one that best fits their needs. By allowing the dependency on the file system to be explicitly captured at image creation time, DADI can help applications exploit the advanced features of file systems such as XFS [11], Btrfs [2], ZFS [12].

In the case of the cgroups runtime, we use an internal module called `vrbd` to provide the virtual block device. `vrbd` is similar to `nbd` but contains improvements that enable it to perform better and handle crashes of the user-space daemon. As shown in Figure 12, I/O requests go from applications to a regular file system such as ext4. From there they go to the virtual block device and then to a user-space daemon called `lsmd`. Reads of data blocks belonging to layers that have already been downloaded are directed to the local file system where the layers are stored. Other read operations are directed to DADI's P2P agent which maintains a persistent cache of recently used data blocks. Write and trim operations are handled by lsmd which writes the data and index files of the writable layer to the local file system.

We have also realized a QEMU driver for its block device backend to export an image to virtualized containers. As shown in Figure 13, the data path in this case is conceptually similar to that for the cgroups runtime except that the image file system and virtual block device are running in the guest context, and the block driver takes the place of lsmd. Integration with other hypervisors should be straightforward. It is also possible to pass the virtual block device from the host into the guest context. This approach works with virtually all hypervisors but incurs a slightly higher overhead. As the block device interface is narrower and simpler than a file system interface, it exposes a small attack surface to the untrusted guest container.

4.2 Container Engine Integration

DADI is integrated with Docker through a graph driver which is a Docker plugin to compose the root file system from layers. The layers form a graph (actually tree) topology hence the name graph driver. DADI is also integrated with containerd through a snapshotter which provides functions similar to those of the graph driver. We will use the term "driver" to refer to either of them in the rest of the paper.

We implemented the drivers to recognize existing and DADI image formats. When they encounter .tgz image, they invoke existing drivers. When they come across DADI image, they perform DADI-specific actions. In this way, the container engine can support both types of images at the same time so that the deployment of DADI to a host does not require the eviction of existing .tgz based containers or images from that host. This enables us to use a canary approach to systematically roll out DADI across our complex production environment.

DADI currently fakes the image pulling process with a small tarball file consisting of DADI-specific metadata. The tarball is very small so that the image pull completes quickly. We are preparing a proposal to the container community for extensions to the image format representation to enable lazy image pulling and make the engine aware of remote images.

4.3 Image Building

DADI supports image building by providing a log-structured writable layer. The log-structured design converts all writes into sequential writes so that the build process with DADI is usually faster than that for regular .tgz images (see Section 5 for details). As DADI uses faster compression algorithms, the commit operation is faster with DADI than it is for regular .tgz images. DADI also avoids pulling entire base images and this saves time when building images on dedicated image building servers where the base images are usually not already local.

In order to build a new layer, DADI first prepares the base image file system by bringing up a virtual block device and mounting the file system on it. When the layer is committed, DADI unmounts the file system and brings down the device. These actions are repeated for each layer produced in a new image, adding up to a lot of time. According to the specification of the image building script (dockerfile), each line of action will produce a new layer. It is not uncommon to see tens of lines of actions in a dockerfile in our environment so a single build job may result in an image with many new layers. This design was supposed to improve the speed of layer downloading by increasing parallelism, but it may become unnecessary with remote image.

We optimized the DADI image build process by bringing the device up and down only once. The intermediate down-and-ups are replaced with a customized operation called stack-and-commit. As its name suggests, stack-and-commit first stacks a new writable layer on top of existing layers, and then commits the original writable layer in the background. This optimization significantly increases image building speed, especially on high-end servers with plenty of resources.

To convert an existing .tgz image into the DADI format, DADI proceeds from the lowest layer of the image to its highest layer. For each layer, DADI creates a new writable layer and unpacks the corresponding .tgz blob into the layer while handling whiteouts, a special file name pattern that indicates deletion of an existing file. If users want to build a DADI image from a .tgz base image, the base image layers must first be converted into the DADI format using this process.

Some container engines implicitly create a special init layer named as xxxxx-init between the container layer and its images layers. This init layer contains some directories and files that must always exist in containers (e.g. /proc, /dev, /sys). During commit, DADI merges this init layer with the container layer so as to keep the integrity of the image file system.

4.4 Deployment Options

The P2P data transfer capability of DADI is optional and targeted at users with large applications. Other users may prefer to use DADI with the layer blobs stored in a high-performance shared storage system as a compromise between fetching the layer blobs from the Registry and storing the layer blobs on every host. Similar solutions have been proposed in the community (e.g. Teleport [9], Wharf [41]). DADI further enhances these solutions by not requiring the layers to be unpacked and supporting alternative storage systems such as HDFS.

For users who do not wish to set up shared storage, DADI provides them with the option to fetch layer blobs on-demand from the Registry and cache the data blocks on local disk(s). This approach greatly reduces cold startup latencies by avoiding the transfer of data blocks that are not needed. If there is a startup I/O trace available when launching a new container instance, DADI can make use of the trace to prefetch the data blocks needed by the starting container, yielding a near-warm startup latency. The trace can be simply collected with `blktrace`, and replayed with `fio`. See Section 5.2 for details.

Users may also choose to use DADI by downloading the layer blobs to local disk(s). DADI layers do not need to be unpacked, saving a time-consuming sequential process needed for .tgz layers. Thus pulling DADI images is much faster. The downloading can be optionally offloaded to P2P tools such as [20,22,24,30,37]. We use this approach as a backup path in case our on-demand P2P transfer encounters any unexpected error.

5 Evaluation

In this section, we evaluate the performance and scalability of DADI Image Service.

5.1 Methodology

We compare the container startup latency with DADI to that with the standard tarball image, Slacker, CRFS, LVM (dm or device mapper), and P2P image download. We also analyze the I/O performance as observed by an application inside the container.

Slacker uses Tintri VMstore as its underlying storage system. We do not have access to such a system so we use LVM together with NFS as an approximation of Slacker (denoted as pseudo-Slacker). At the time of this writing, CRFS has not yet achieved its goal of realizing an internal overlayfs so we rely on the kernel implementation of overlayfs for the comparisons.

We generally use NVMe SSDs as local storage. We also emulate a low-speed disk by limiting IOPS to 2,000 and throughput to 100 MB/s. These are the performance characteristics of the most popular type of virtual disks on public clouds so we refer to such a disk as “cloud disk” in the rest of the paper. We use ZFile by default for DADI unless explicitly noted. Before starting a test, we drop the kernel page cache in the host and guest (if applicable) as well as the persistent cache of DADI.

The physical servers we use are all equipped with dual-way multi-core Xeon CPUs and 10GbE or higher-speed NICs. The

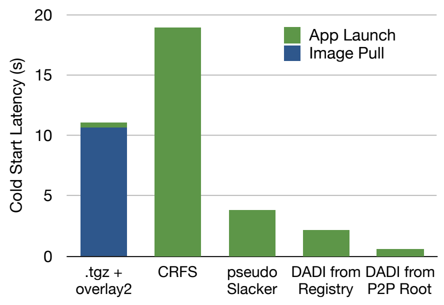

Figure 14: Cold Startup Latency.

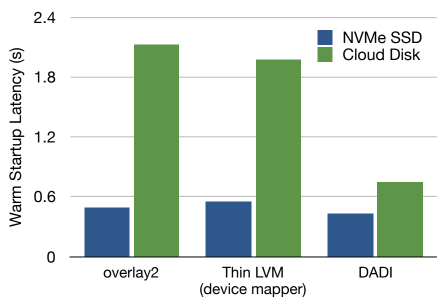

Figure 15: Warm Startup Latency.

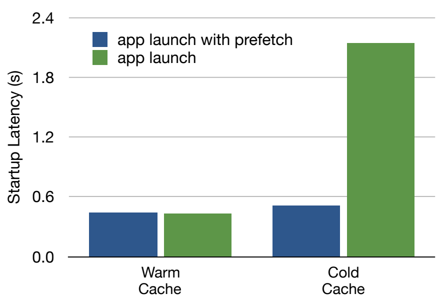

Figure 16: Startup Latency with Trace-Based Prefetch.

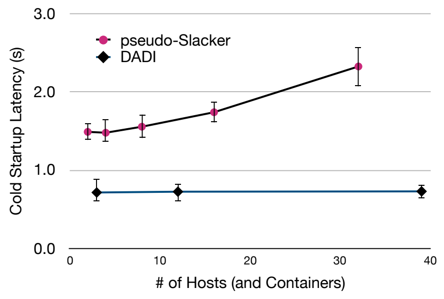

Figure 17: Batch Cold Startup Latency. Bars indicate 10 and 90 percentiles.

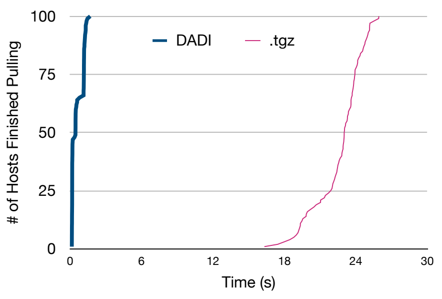

Figure 18: Time to Pull Image in Production Environment.

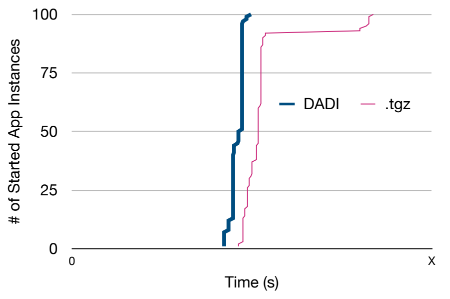

Figure 19: Time to Launch Application in Production Environment.

VMs are hosted on our public cloud. Each VM is equipped with 4 CPU cores and 8 GBs of memory. The vNICs are capable of a burst bandwidth of 5 Gbps and sustained bandwidth of 1.5 Gbps.

5.2 Startup Latency

To evaluate container startup latency, we use the application image `WordPress` from DockerHub.com. WordPress is the most popular content management system powering about one third of the Web in 2019 [7]. The image consists of 21 layers in .tgz format with a total size of 165MB. When unpacked, the image size is 501MB. In DADI compressed format with lz4 compression, the image occupies 274MB. The tarball of DADI-specific metadata that is downloaded on image pull is only 9KB in size.

Cold start of a single instance. We test startup latencies of a single container instance running WordPress when the layer blobs are stored in the Registry (.tgz, DADI, CRFS) and on remote storage servers (DADI, pseudo-Slacker). All the servers are located in the same datacenter as the container host. The results, as summarized in Figure 14, show that container cold startup time is markedly reduced with DADI.

Warm start of a single instance. Once the layer blobs are stored or cached on local disk, the containers can be started and run without a remote data source. In this case, any difference in startup time can be attributed to the relative efficiency of the I/O paths. As indicated in Figure 15, DADI performs 15%~25% better than overlayfs and LVM on NVMe SSD, and more than 2 times better on cloud disk.

Cold startup with trace-based prefetching. We first make use of `blktrace` to record an I/O trace when starting a container. On another host, we use `fio` to replay only the read operations in the trace while starting a new container instance of the same image. We set fio to replay with a relatively large I/O depth of 32 so as to fetch data blocks before they are actually read by the application. Figure 16 shows the results. Observe that trace-based prefetching can reduce 95% of the difference between cold and warm startup times.

Batch cold startup. In practice, many applications are large and require multiple instances to be started at the same time. For this batch startup scenario, we compare only pseudo-Slacker and DADI because the .tgz image and CRFS are bottlenecked by the Registry. The results are presented in Figure 17. Note that the startup time with pseudo-Slacker begins at 1.5s for one instance and increases to 2.3s for 32 instances. On the other hand, the startup time with DADI remains largely constant at 0.7s as the number of instances increases.

Startup in our Production Environment. We selected typical deployment tasks for an application in our production environment and analyzed its timing data. As shown in Figure 18, pulling the DADI metadata tarball takes no more than 0.2s for nearly half of the hosts and around 1s for the rest of the hosts. This compares very favorably with pulling the equivalent .tgz image which takes more than 20s for most

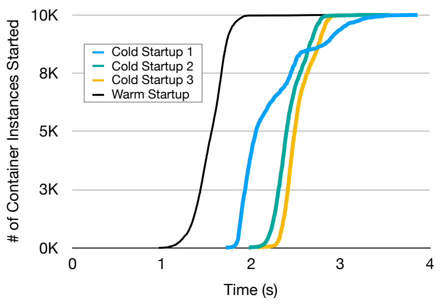

Figure 20: Startup Latency using DADI (Large-Scale Startup).

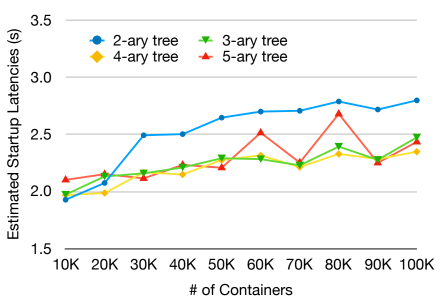

Figure 21: Projected Startup Latency using DADI (Hyper-Scale Startup).

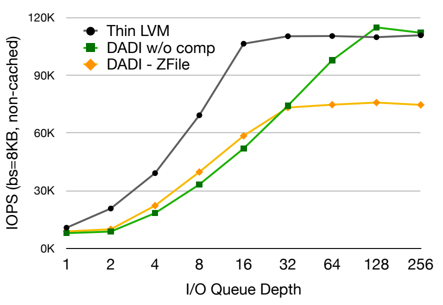

Figure 22: Uncached Random Read Performance.

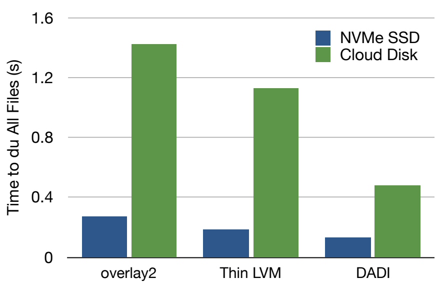

Figure 23: Time to du All Files.

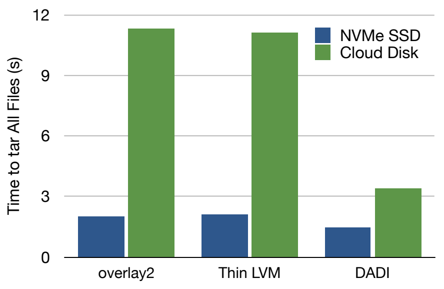

Figure 24: Time to tar All Files.

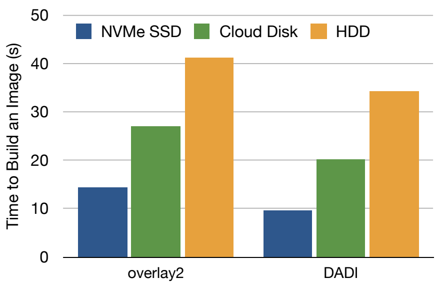

Figure 25: Time to Build Image.

of the hosts. Note that in this case, the .tgz image pull only needs to download the application layers as the much larger dependencies and OS layers already exist on the hosts. If all the layers have to be downloaded, the time needed will be even higher.

As shown in Figure 19, applications start faster using DADI remote image and P2P data transfer than with the .tgz image stored on local SSD. This result surprised us initially but it turned out to be a common occurrence for a couple of reasons. First, overlayBD performs better than OverlayFS (See Section 5.4). Second, with the tree-structured P2P data transfer, hosts effectively read from their parents' page cache, and this is faster than reading from their local disks.

5.3 Scalability

For the scalability analysis, we use a lightweight application called `Agility`. Agility is a Python application based on CentOS 7.6. Its image consists of 16 layers with a total size of 575MB in ZFile format and 894MB uncompressed. When Agility starts, it accesses a specified HTTP server which records the time stamps of all the accesses. We use Agility instead of WordPress for our scalability test because it provides a means to collect timings of a large number of container instances. Agility also consumes fewer resources, allowing us to create many more containers in our testbed.

Large-scale startup with DADI. We create 1,000 VMs on our public cloud platform and use them as hosts for containers. A large and increasing portion of our production environment is VM-based so this test reflects our real world situation. We start 10 containers running Agility on each host for a total of 10,000 containers. As shown in Figure 20, the cold start latency with DADI is within a second or two of that for warm start. The experimental environment is not dedicated and some noise is apparent in one of the runs (Cold Startup 1). Note that other than for ramp-up and long-tail effects, the time taken to start additional containers is relatively constant.

Hyper-scale startup with DADI. We deliberately construct a special P2P topology with tens of hosts and use it to project the behavior for a full tree with tens of thousands of hosts. The special topology models a single root-to-leaf path where each interior node has the maximum number of children. Each host again runs 10 instances of Agility. As shown in Figure 21, the startup time is largely flat as the number of containers increases to 100,000. Notice also that a binary tree for P2P is best when there are fewer than 20,000 participating hosts. A 3-ary or 4-ary tree works better beyond that scale.

5.4 I/O Performance

We perform micro benchmarks with `fio` to compare uncached random read performance. The results are summarized in Figure 22. At an I/O queue depth of 1, DADI offers comparable performance to LVM despite its user-space implementation. DADI's performance ramps up slower as the queue depth is increased but it catches up and tops LVM to achieve the highest IOPS at an I/O queue depth of 128 and without compression. This behavior suggests that DADI's index is more efficient than that of LVM, but there is room to optimize our queueing and batching implementation. Observe that DADI with compression performs 10%~20% better than without compression when the I/O queue depth is less than 32. This is because compression, by reducing the amount of data transferred, increases effective I/O throughput provided that the CPU is not bottlenecked. In our experimental setup, the CPU becomes bottlenecked for ZFile beyond a queue depth of 32.

We also test I/O performance with `du` and `tar` to scan the entire image from inside the container. These tests respectively emphasize small random read and large sequential read. The output of these commands are ignored by redirecting to `/dev/null`. As shown in Figure 23 and 24, DADI outperforms both overlayfs and LVM in all cases especially on the cloud disk. This is again primarily due to the effect of compression in reducing the amount of data transferred.

5.5 Image Building Speed

Image building speed is driven primarily by write performance and the time needed to setup an image. We evaluate image building performance with a typical dockerfile from our production environment. The dockerfile creates 15 new layers comprising 7,944 files with a total size of 545MB, and includes a few `chmod` operations that trigger copy-ups in overlayfs-backed images. As shown in Figure 25, the image is built 20%~40% faster on DADI than on overlayfs. Note that the time to commit or compress the image is not included in this measurement.

6 Discussion and Future Work

With overlayfs, containers that share layers are able to share the host page cache when they access the same files in those shared layers. Because DADI realizes each layered image as a separate virtual block device, when multiple containers access the same file in a shared layer, the accesses appear to the host to be for distinct pages. In other words, the host page cache is not shared, potentially reducing its efficiency.

One way to address this issue is to introduce a shared block pool for all the virtual block devices corresponding to the different containers on a host. The basic idea is to use the device mapper to map segments from the pool to the virtual block devices such that accesses by different containers to the same file in a shared layer appear to be for the same segment in the pool. The pool is backed by the page cache while the virtual block device and file system on top will need to support Direct Access (DAX) to avoid double caching. This solution can be further improved by performing block-level deduplication in the pool.

With the emergence of virtualized runtimes, container is becoming a new type of virtual machine and vice versa. The runtimes of container and VM may also begin to converge. By being based on the widely supported block device, DADI image is compatible with both containers and VMs, and is naturally a converged image service. Such a converged infrastructure will bring the convenience and efficiency of layered image to VM users on the cloud today. It will also provide users with increased flexibility and enable applications to evolve gradually from cloud-based to cloud-native.

A key part of realizing the potential of DADI is to standardize its image format and facilitate its adoption. We are working to contribute core parts of DADI to the container community.

7 Conclusions

We have designed and implemented DADI, a block-level remote image service for containers. DADI is based on the observation that incremental image can be realized with block-based layers where each layer corresponds to a set of file changes but is physically the set of changes at the block level underneath a given file system. Such a design allows the image service to be file system and platform agnostic, enabling applications to be elastically deployed in different environments. The relative simplicity of block-based layers further facilitates optimizations to increase agility. These include fine-grained on-demand data transfer of remote images, online decompression with efficient codecs, trace-based prefetching, peer-to-peer transfer to handle burst workload, easy integration with the container ecosystem. Our experience with DADI in the production environment of one of the world's largest ecommerce platforms show that DADI is very effective at increasing agility and elasticity in deploying applications.

Acknowledgments

This paper would not have been possible without the close collaboration of our storage, container, serverless, kernel and virtualization teams. We are especially grateful to Haobo Xu, Liang Han, Qianbo Huai and Jiwei Lu for delivering the initial version of the P2P Data Transfer capability. We are also grateful to our shepherd, Vasily Tarasov, and anonymous reviewers for helping us improve the paper.

References

[1] 9p virtio - KVM. https://www.linux-kvm.org/page/9p_virtio. Accessed: 2020-01-15.

[2] Btrfs. https://btrfs.wiki.kernel.org/index.php/Main_Page. Accessed: 2020-01-15.

[3] Everything You Need to Know about Linux Containers, Part I: Linux Control Groups and Process Isolation. https://www.linuxjournal.com/content/everything-you-need-know-about-

linux-containers-part-i-linux-control-groups-and-process. Accessed: 2020-01-15.

[4] File Allocation Table. https://en.wikipedia.org/wiki/File_Allocation_Table. Accessed: 2020-01-15.

[5] gVisor. https://gvisor.dev/. Accessed: 2020-01-15.

[6] Minify and Secure Your Docker Containers. Frictionless! https://dockersl.im/. Accessed: 2020-01-15.

[7] One-third of the web! https://wordpress.org/news/2019/03/one-third-of-the-web/. Accessed: 2020-01-15.

[8] Overlay Filesystem. https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt. Accessed: 2020-01-15.

[9] Project Teleport. https://azure.microsoft.com/en-gb/resources/videos/azure-friday-how-to-expedite-container-startup-with-project-teleport-and-azure-container-registry/. Accessed: 2020-01-15.

[10] virtio-fs. https://virtio-fs.gitlab.io/. Accessed: 2020-06-05.

[11] XFS. http://xfs.org/. Accessed: 2020-01-15.

[12] ZFS: The Last Word in Filesystems. https://blogs.oracle.com/bonwick/zfs:-the-last-word-in-filesystems/. Accessed: 2020-01-15.

[13] Amazon. Secure and fast microVMs for serverless computing. https://firecracker-microvm.github.io/. Accessed: 2020-01-15.

[14] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless Computing: Current Trends and Open Problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.

[15] Bram Cohen. Incentives Build Robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72, 2003.

[16] Lian Du, Tianyu Wo, Renyu Yang, and Chunming Hu. Cider: A Rapid Docker Container Deployment System through Sharing Network Storage. In *2017 IEEE 19th International Conference on High Performance Computing and Communications (HPCC'17)*, pages 332–339. IEEE, 2017.

[17] Gideon Glass, Arjun Gopalan, Dattatraya Koujalagi, Abhinand Palicherla, and Sumedh Sakdeo. Logical Synchronous Replication in the Tintri VMstore File System. In *16th USENIX Conference on File and Storage Technologies (FAST'18)*, pages 295–308, 2018.

[18] N Hardi, J Blomer, G Ganis, and R Popescu. Making Containers Lazy with Docker and CernVM-FS. In *Journal of Physics: Conference Series*, volume 1085, page 032019. IOP Publishing, 2018.

[19] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Slacker: Fast Distribution with Lazy Docker Containers. In *14th USENIX Conference on File and Storage Technologies (FAST'16)*, pages 181–195, 2016.

[20] Alibaba Inc. Dragonfly: An Open-source P2P-based Image and File Distribution System. https://d7y.io/en-us/. Accessed: 2020-01-15.

[21] Google Inc. CRFS: Container Registry Filesystem. https://github.com/google/crfs. Accessed: 2020-01-15.

[22] Uber Inc. Introducing Kraken, an Open Source Peer-to-Peer Docker Registry. https://eng.uber.com/introducing-kraken/. Accessed: 2020-01-15.

[23] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv preprint arXiv:1902.03383*, 2019.

[24] Wang Kangjin, Yang Yong, Li Ying, Luo Hanmei, and Ma Lin. FID: A Faster Image Distribution System for Docker Platform. In *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 191–198. IEEE, 2017.

[25] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSv–Optimizing the Operating System for Virtual Machines. In *2014 USENIX Annual Technical Conference (USENIX ATC'14)*, pages 61–72, 2014.

[26] Huiba Li, Yiming Zhang, Dongsheng Li, Zhiming Zhang, Shengyun Liu, Peng Huang, Zheng Qin, Kai Chen, and Yongqiang Xiong. URSA: Hybrid Block Storage for Cloud-Scale Virtual Disks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, page 15. ACM, 2019.

[27] Haifeng Liu, Wei Ding, Yuan Chen, Weilong Guo, Shuoran Liu, Tianpeng Li, Mofei Zhang, Jianxing Zhao, Hongyin Zhu, and Zhengyi Zhu. CFS: A Distributed

File System for Large Scale Container Platforms. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1729–1742. ACM, 2019.

[28] James Mickens, Edmund B Nightingale, Jeremy Elson, Darren Gehring, Bin Fan, Asim Kadav, Vijay Chidambaram, Osama Khan, and Krishna Nareddy. Blizzard: Fast, Cloud-Scale Block Storage for Cloud-Oblivious Applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, pages 257–273, 2014.

[29] Kazutaka Morita. Sheepdog: Distributed Storage System for QEMU/KVM. *LCA 2010 DS&R miniconf*, 2010.

[30] Aravind Narayanan. Tupperware: Containerized Deployment at Facebook, 2014.

[31] OCI. OCI Artifacts. `https://github.com/opencontainers/artifacts`. Accessed: 2020-01-15.

[32] Mendel Rosenblum and John K Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[33] Richard P Spillane, Wenguang Wang, Luke Lu, Maxime Austruy, Rawlinson Rivera, and Christos Karamanolis. Exo-clones: Better Container Runtime Image Management across the Clouds. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'16)*, 2016.

[34] Vasily Tarasov, Lukas Rupprecht, Dimitris Skourtis, Amit Warke, Dean Hildebrand, Mohamed Mohamed, Nagapramod Mandagere, Wenji Li, Raju Rangaswami, and Ming Zhao. In Search of the Ideal Storage Configuration for Docker Containers. In *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 199–206. IEEE, 2017.

[35] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. CNTR: Lightweight OS Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC'18)*, pages 199–212, 2018.

[36] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *15th USENIX Conference on File and Storage Technologies FAST'17)*, pages 59–72, 2017.

[37] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

[38] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[39] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. KylinX: A Dynamic Library Operating System for Simplified and Efficient Cloud Virtualization. In *2018 USENIX Annual Technical Conference (USENIX ATC'18)*, pages 173–186, 2018.

[40] Zhaoning Zhang, Ziyang Li, Kui Wu, Dongsheng Li, Huiba Li, Yuxing Peng, and Xicheng Lu. VMThunder: Fast Provisioning of Large-Scale Virtual Machine Clusters. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3328–3338, 2014.

[41] Chao Zheng, Lukas Rupprecht, Vasily Tarasov, Douglas Thain, Mohamed Mohamed, Dimitrios Skourtis, Amit S Warke, and Dean Hildebrand. Wharf: Sharing Docker Images in a Distributed File System. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 174–185. ACM, 2018.