Cluster Computing (2019) 22:1159–1172
https://doi.org/10.1007/s10586-018-02893-y

Evaluating Docker storage performance: from workloads to graph drivers

Vasily Tarasov[1] · Lukas Rupprecht[1] · Dimitris Skourtis[1] · Wenji Li[2] · Raju Rangaswami[3] · Ming Zhao[2]

Received: 31 March 2018 / Revised: 7 September 2018 / Accepted: 19 December 2018 / Published online: 1 January 2019
© Springer Science+Business Media, LLC, part of Springer Nature 2019

Abstract
Containers are a widely successful technology today popularized by Docker. They improve system utilization by increasing workload density and enable seamless deployment of workloads across development, test, and production environments. Docker's unique approach to data management, which involves frequent snapshot creation and removal, presents a new set of exciting challenges for storage systems. At the same time, storage management for Docker containers has remained largely unexplored with a dizzying array of solution choices and configuration options. In this paper we unravel the multi-faceted nature of Docker storage and demonstrate its impact on system and workload performance. As we uncover new properties of the popular Docker storage drivers, this is a sobering reminder that widespread use of new technologies can often precede their careful evaluation.

Keywords Containers · Docker · Storage · Performance

1 Introduction

Operating systems (OSs) use *processes* as a powerful and convenient computing abstraction. However, as the hardware capabilities of machines improved, it became cost-efficient to share the abundant hardware resources across multiple users and applications but with better isolation than native processes. Process *containers* made this transition possible. The rapid adoption of containers is fueled by cloud computing whereby multiple tenants can transparently run their workloads on the same node. According to a recent poll, 25% of enterprises are already using containers, while at least 62% are at some stage of adopting them [1].

At its core, a container is a set of processes that is isolated from other processes running in the system. Linux uses control groups (*cgroups*) [2], to limit the resource usage (e.g., memory and CPU) of containers, and *namespaces* [3] to confine process visibility. In 2013, *Docker* has emerged as a composite technology that enables and simplifies the adoption of containers in modern OSs [4]. Docker containers allow users to effectively capture runtime environments in persistent *images* and easily execute the resident software inside dynamically created containers. Docker *images* contain all information needed to run the packaged software which significantly simplifies deployment across development, testing, and production environments.

While containers adequately address CPU and memory isolation across workloads, storage isolation is more challenging. At a fundamental level, storage for containers introduces the need to deal with an abundance of duplicate data referenced within container images. In a naive

✉ Vasily Tarasov
vtarasov@us.ibm.com

Lukas Rupprecht
lukas.rupprecht@ibm.com

Dimitris Skourtis
dimitrios.skourtis@us.ibm.com

Wenji Li
wenjili@asu.edu

Raju Rangaswami
raju@cs.fiu.edu

Ming Zhao
mingzhao@asu.edu

1 IBM Research, San Jose, USA

2 Arizona State University, Tempe, USA

3 Florida International University, Miami, USA

implementation, Docker would need to create a complete copy of the image for every running container, which puts high pressure on the I/O subsystem and makes container start times unacceptably high for many workloads. As a result, Docker uses *copy-on-write* (CoW) storage and storage *snapshots* and splits images into multiple *layers*. A layer consists of a read-only set of files and layers with the same content can be shared across images. Each running container also contains a *writeable layer* which records all changes to files in the read-only layers via CoW. Sharing layers reduces the amount of storage required to run containers and container start times.

With Docker, one can choose Aufs [5], Overlay2 [6], Btrfs [7], ZFS [8] or device-mapper (dm) [9] as *storage drivers* which provide the required snapshotting and CoW capabilities for images. None of these solutions, however, were designed with Docker in mind and their effectiveness for Docker has not been systematically evaluated in the literature. With Docker, the depth of the file system software stack increases significantly. Besides, the available variety of CoW configurations and the possible diversity in workload mixes make the selection of the right storage solution challenging. A system architect must decide not only which CoW storage technology to use, but also how to configure it. For example, with Aufs and Overlay2, one needs to select the type of the underlying file system, and for device-mapper, which file system to format the thin-provisioned device with.

Unlike conventional workloads, Docker workloads induce frequent snapshot creation and destruction. The amount of data churn and the rate of image commits impact the behavior of the system significantly. Moreover, the number of layers in an image can impact performance both negatively and positively. Finer-grained images with more layers increase the amount of data that can be shared across different containers while on the other hand, more layers cause more overhead when accessing files in a container [10]. High-density containerized environments also increase the diversity of workloads and their parallelism. Resource isolation constraints imposed on individual containers can further morph workloads in non-trivial ways.

Although many questions arise when choosing storage for Docker containers, there is little to no guidance in peer-reviewed literature for selecting or designing storage for Docker [11]. In this study, we demonstrate the complexity of the Docker storage configuration and empirically demonstrate how the selection of the storage solution impacts system and workload performance. After discussing the relevant background on Docker and introducing the available storage drivers (Sect. 2), we make the following contributions:

- We present and discuss the different dimensions that influence the choice of an appropriate storage solution for Docker containers (Sect. 3).
- We conduct a preliminary evaluation of the introduced dimensions and analyze their impact on performance and stability (Sect. 4).

We found that, for example, for read-intensive workloads, Aufs and Overlay2 are a good choice, while Btrfs and ZFS are superior for workloads with many writes to large files. At the same time, for realistic workloads like code compilation and system upgrade, Btrfs and ZFS did not perform as well as expected. Device-mapper has a stable codebase but its performance is typically low and highly susceptible to the underlying storage speed.

These and other observations can serve as a starting point for future work on storage management for Docker containers. The goal of this study is not to provide a comprehensive analysis of Docker storage but rather to serve as a stepping stone towards further in-depth research of this emerging technology.

2 Docker

Docker consists of a Command Line Tool (CLI) and a daemon (sometimes called engine) which continuously runs in the background of a dockerized system (Fig. 1). The Docker daemon receives user commands from the CLI to build new or retrieve existing images, start or stop containers from images, delete containers or images, and other actions. Next, we introduce the relevant background on Docker images and its concept of storage drivers in more detail.

2.1 Docker images

Modern software relies heavily on the availability of a file system interface for its processes (after all, “Everything is a file” in UNIX-derived OSs). Files not only store binaries, configurations, and data, but also provide access to system information and configuration options (e.g. `/proc` and /

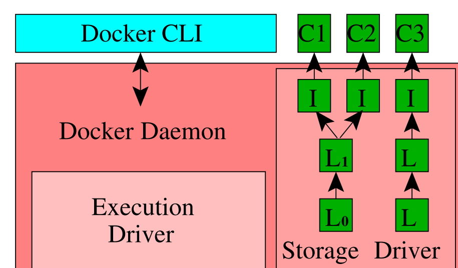

Fig. 1 Docker high-level design. *C* stands for container, *I* for image, and *L* for layer

sys file systems). Docker therefore dynamically creates a file system for a container to execute from. Docker file system images are similar to VM images, except that they consist of a series of layers where every layer is a set of files. Layers in an image are stacked on top of each other using a *union mount* and files in the upper layers supersede files in the layers below them. For example, in Fig. 1, if the same file resides in layers L_0 and L_1, then containers $C1$ and $C2$ only see the version from L_1. However, all files in L_0 that do not exist in the upper layers will be seen by containers $C1$ and $C2$. In almost all cases, a container's file system is stored on the local storage device of the node, which executes the container.

The number of layers in a single image ranges from one to several dozens [10]. Similarly to git, the layers are identified by fingerprints of their content and different images can share layers, which provides significant space and I/O transfer savings. A layer in a Docker image often represents a layer in the corresponding software stack. For example, an image could consist of a Linux distribution layer, a libraries layer, a middleware layer, and an application layer.

A container image is read-only, with changes to its file system during execution stored separately. To create a container from an image, Docker creates an additional *writeable* layer on top of the image with which the container interacts. When the container updates a file, the newly written data does not overwrite the data in-place. For example, if the Aufs storage driver is used, the file is copied to the writable layer and only the copy is updated (copy-on-write). Unless the user saves the changes as a new layer (and hence a new image), the changes are discarded when the container is removed.

To store data persistently beyond container removal, users can attach one or more file system *volumes* using a *volume driver*, which provides the container access to data on the local file system or remotely via protocols such as NFS and iSCSI. In this study, our focus is on challenges specific to configuring the local "ephemeral" file system for storing and accessing container images.

Users exchange images via a Docker *registry* service which typically runs on independent machines. For example, Docker Hub is a popular registry service storing over 400,000 images [12]. Docker clients cache images locally and therefore can start any number of containers from an image after pulling it once. In this paper we assume that images are already pulled from the registry.

Docker containers are often managed by high-level frameworks such as Docker Swarm [13], Kubernetes [14], and others. Furthermore, many products use Docker containers as a basic primitive for their workflows [15, 16]. In this paper, we generate workloads at the Docker level, not employing orchestration frameworks or complex workflows. We are planning to extend our evaluation in the future.

2.2 Storage drivers

Docker uses a variety of pluggable *storage drivers*[1] to manage the makeup and granularity of the layers and how changes to layers are saved. A storage driver is responsible for preparing the file system for a container. In this section, we first describe the storage driver interface as this allows to better understand the unique characteristics of the workload that Docker creates on the underlying storage. We then describe the available Docker storage drivers and their key features.

2.2.1 API

As Docker itself, storage drivers are typically implemented in the *Go* programming language. Each storage driver implements the `Driver` interface, which consists of two sub-interfaces: `ProtoDriver` and `DiffDriver`. `ProtoDriver` includes the minimum set of methods that a storage driver must implement. `DiffDriver` covers the methods related to the comparison of layers. `DiffDriver` methods are optional as Docker already provides a generic implementation o the interface.

The `ProtoDriver` API consists of 10 methods that typically take child and parent layer identifiers as arguments (`cid` and `pid` in our description). A layer identifier is a hex string generated by the Docker core. For brevity, we omit some less important arguments in the methods below.

1. `String()` returns the name of the driver.
2. `CreateReadWrite(cid, pid)` creates a new writable layer with a child `cid` based on the parent `pid`. The method is called when a new container starts.
3. `Create(cid, pid)` creates a new read-only layer `cid` on top of the parent layer `pid`. If a container uses a multi-layer image, this method is called multiple times during container start up.
4. `Remove(cid)` removes the layer.
5. `Get(cid)` returns the absolute path where the layer is mounted.
6. `Put(cid)` indicates to the driver that the specified layer is no longer referenced by Docker.
7. `Exists(cid)` checks if a layer exists.
8. `Status()` requests the driver's internal status as a set of arbitrary key-value string pairs.

[1] Storage drivers are sometimes also called *graphdrivers* because they maintain the graph (tree) of Docker layers and images.

9. `GetMetadata(cid)` requests driver-specific metadata information about the layer.
10. `Cleanup()` is called before Docker shuts down.

The `DiffDriver` interface consists of an additional 4 methods:

1. `Diff(cid, pid)` generates an archive of differences between layer `cid` and its parent layer `pid`.
2. `Changes(cid, pid)` produces the list of files added, deleted, or changed between layers `cid` and `pid`.
3. `ApplyDiff(cid, pid, archive)` extracts the provided archive into the layer `cid`.
4. `DiffSize(cid, pid)` returns the size of the difference between layers `cid` and `pid` in bytes.

In the following, we explain the different storage driver implementations for Docker.

2.2.2 VFS

This simple driver does not save file updates separately from an image via CoW, but instead creates a complete copy of the image for each newly started container. It can, therefore, run on top of any file system. While this driver is not recommended for production due to its inefficiency, we discuss it in our evaluation because it is very stable and offers as a good baseline.

2.2.3 Aufs

Another Union File System [5] is a union file system that takes multiple directories, referred to as *branches*, and stacks them on top of each other to provide a single unified view at a single mount point. Aufs performs file-level CoW, storing updated versions of files in upper branches. To support Docker, each branch maps to an image layer.

To find a file, Aufs searches each layer/branch from top to bottom until the required file is found. Once found, a file can be read by the container. To update a file, Aufs first creates a copy of the file in the top writeable branch, and then updates the newly copied file. Deletes are handled by placing a *whiteout file* in the top writeable layer, which obscures all versions of the file in the lower layers. Aufs performance depends on many factors—application access pattern, the number of files, distribution of files across layers, and file sizes.

2.2.4 Overlay and Overlay2

Both of these drivers rely on the same underlying file system—OverlayFS [6]—which is another implementation of a union file system. Unlike Aufs, OverlayFS is in the Linux kernel mainline and is therefore available in many Linux distributions out of the box. The Overlay driver was created for an earlier version of OverlayFS that supported only two layers: a read-only "lower" layer and a mutable "upper" layer. To merge several read-only layers the Overlay driver creates hardlinks to the shared files which can lead to inode exhaustion problems.

The Overlay2 driver relies on the newer version of OverlayFS (kernel 4.0 and higher) which already supports 128 lower branches and therefore can merge up to 128 read-only layers without using hardlinks. In this paper we do not evaluate the older Overlay driver, and instead focus on the newer Overlay2 driver.

2.2.5 Device-mapper (dm)

Unlike other storage drivers, dm operates at the block level instead of the file system level. This driver uses the Linux device-mapper subsystem to create *thin-provisioned* block devices. First, an administrator creates a *pool*, which typically stacks on top of two physical devices—one for user data and one for device-mapper metadata (e.g., block mappings). Second, when Docker creates a container, the dm driver allocates an individual *volume* for the container from the pool. To benefit from CoW, dm usually creates volumes as writable snapshots of existing volumes.

However, for a Docker container to operate, it requires a file system, rather than a raw block device. So, as a third step, dm formats volumes with a configurable file system (usually Ext4 or XFS). By far the largest advantage of dm over Aufs and Overlay2 is that it can perform CoW at a granularity finer than a single file, which is 512 KB by default but configurable by the user. On the other hand, dm is completely file system-oblivious and therefore cannot benefit from using any file system information during snapshot creation.

2.2.6 Btrfs

Btrfs [7] is a modern CoW file system based on a *CoW-friendly* version of a B-tree. Compared to Aufs and Overlay2, Btrfs is a file system that natively supports CoW and does not require an underlying file system. Btrfs implements the concept of *subvolumes* which are directory trees, represented in their own B-trees. Subvolumes can be efficiently snapshotted by creating a new root which points to the children of the existing root.

The Btrfs storage driver stores the base layer of an image as a separate subvolume and consecutive images are snapshots of their parent layer. Similar to dm, Btrfs performs CoW at the granularity of blocks which is more efficient in terms of performance and space utilization compared to file-based CoW. On the downside, Btrfs can

experience higher fragmentation due to the finer-grained CoW.

2.2.7 ZFS

ZFS [8] is a file system with native CoW support. It started as part of Solaris and was recently ported to Linux. Like Btrfs, it directly manages the raw storage without requiring an underlying file system, and supports snapshots on volumes and directories. It uses the concept of storage pools to manage physical storage, where the blocks in a storage pool form a Merkle tree. ZFS can take a snapshot efficiently on this tree by creating a new root like in Btrfs. It performs CoW at the granularity of blocks and shares the same benefits and costs as Btrfs ' CoW.

ZFS can be used in Linux using the kernel-level implementation [17], which relies on the Solaris Porting Layer (SPL) kernel module [18]. The module allows to compile Solaris kernel-level code against the Linux kernel. Another option is to use FUSE-ZFS is, the FUSE-based implementation of ZFS. However, at this point ZFS on FUSE is not recommended by Docker and therefore, we use the native kernel implementation of ZFS on Linux.

3 Dimensions of container storage

When designing a containerized environment, one has to choose from a large variety of storage options that span multiple dimensions, which we describe in this section.

3.1 Storage drivers

As discussed in Sect. 2.2, Docker supports a variety of storage drivers. The storage driver choice is driven by three main considerations [19]: (1) space efficiency, (2) I/O performance, and (3) stability. Each driver uses a different approach to represent images and hence some drivers may not be suitable for certain types of workloads. For instance, if a large file is updated in a container, Aufs and OverlayFS would have to copy the entire file, decreasing performance and disk space usage. In another example, for containers with many files and deep directories, Aufs lookup operations can be slow because Aufs looks for a file at every layer of the image one at a time. Further, in our experience, some drivers are stable while others are experimental and not production-ready. All this makes choosing the appropriate driver difficult as it depends on the workload and the environment in which it is deployed. Table 1 summarizes the positive and negative impacts of each of the five supported Docker storage drivers across efficiency, performance, and stability metrics.

Additionally, we observed that the network traffic remained unaffected by the choice of the storage driver. This demonstrates that, unlike common belief, Docker exchanges images with a registry service at the file-level granularity regardless of the storage driver. Significant improvements are possible in this area [10, 20].

3.2 Image layers

Docker's representation of images as "layers of changes" adds another dimension of complexity. Having a large number of fine-grained, generic layers allows for a higher degree of sharing between different containers. This improves performance when pulling images from a central registry and reduces storage consumption at the Docker host. On the other hand, deeply layered images increase the latency of file system operations such as `open` or `read` [10], because more layers need to be traversed to reach the data. Depending on the specific user workload, the number of layers might or might not affect system performance.

3.3 Storage devices

Another important factor for Docker deployments is which type of storage devices should be used for storing container data. Docker is especially popular in the cloud and large cloud vendors such as Amazon, Microsoft, IBM, and Google offer a wide variety of storage options. For example, Amazon offers local ephemeral and remote persistent, block storage based on either SSDs or HDDs [21]. Depending on the VM instance type, one or the other (or both) can be selected. For remote storage, a variety of options with different throughput and IOPS specifications exist.

With Docker, the density and diversity of workloads running against the same storage device increase significantly. This is due to the fact that hosts can execute a larger number of lightweight containers compared to VMs. As a result, matching containers to the variety of available

Table 1 Gross comparison of container storage drivers

Storage driver	Space efficiency	I/O perf	Driver stability
VFS	–	–	+
Aufs	~	~	–
Overlay2	~	~	~
dm	+	~	+
Btrfs	+	+	–
ZFS	+	+	+

+ positive, ~ neutral, – negative

devices becomes a more challenging task. The burden is on the user to optimize for performance while keeping the cost low and ensuring that all workloads have sufficient resources to run smoothly.

3.4 File systems

While the Btrfs and ZFS drivers are based upon Btrfs and ZFS formatted devices, the other drivers include an additional file system layer. Overlay2 and Aufs are both union file systems that require an underlying file system such as Ext4 or XFS. dm on the other hand exposes a block device and hence, has to be formatted with a file system. While recommendations exist for some drivers regarding the choice of file system [22], it is unclear as to how this choice affects performance. Furthermore, every file system has a large number of parameters and their optimal values for Docker can be different compared to traditional setups. For example, file system journaling is usually not needed as crash consistency is not required for a container's ephemeral (local) storage.

3.5 Workloads

The "right" choices made with respect to the above dimensions are all influenced by the characteristics of the intended workload. The workload itself has several dimensions in terms of read/write ratio, the type of I/O (random or sequential), and the amount of data (size and number of files) that it operates on. A read-only workload that operates on a large number of files does not incur any overhead due to CoW but may under-perform with deeply layered images as it has to perform frequent lookups. Balancing such trade-offs is difficult, especially when the workload characteristics are not known or are not clearly defined.

Looking at the different dimensions, it becomes clear that picking the optimal storage solution for a containerized workload is not straightforward. While some guidelines exist [19], they do not cover all dimensions and do not provide clear evidence as to how the different aspects of a given deployment influence one another. Additionally, in many cases, the best choices are tightly coupled with the workload properties. Our evaluation study highlights some of the complexities of running workloads within the Docker storage ecosystem.

4 Evaluation

The presented evaluation is not intended as a comprehensive study. It is aligned with the goals of the paper to demonstrate the impact (or the lack of it) of the Docker storage configuration on workload performance.

4.1 Testbed

In our experiments we used an IBM x3650 M4 server provisioned to support environments with a high density of containers per machine. Each node has two 8-core Intel Xeon 2.6 GHz CPUs, 96 GB of RAM, and a single 128 GB SSD fully dedicated to Docker. The node runs a CentOS Linux 7.3.1611 (released in December 2016) with an updated kernel version `4.11.6` (released in June 2017) to enable support for Overlay2 and Aufs. For the same reason we updated Docker to version `17.03.1-ce` (released in March 2017). We used Ext4 as the underlying file system for the storage drivers (except for the Btrfs and ZFS drivers). While we also ran experiments with XFS, our preliminary results indicate no significant difference caused by the file system and hence, we do not show the XFS results.

In our earlier experiments (not presented in here) we used an older kernel version (4.10) and Aufs hung under write-intensive workloads. Furthermore, when we switched to the newer 4.11 kernel, Aufs was returning errors on writes to the internal portions of large files on XFS. We reported this issue to the Aufs maintainers and it was recently fixed [23]. This contributed to our decision to mark Aufs stability as negative.

We used kernel-level ZFS for Linux version `0.7.6-1` prepackaged for CentOS 7.3 [24].

4.2 Microbenchmark workloads

We created several different Docker images for the experiments with micro workloads (experiments 1–3). Each image is kept to the minimum size and only contains a Filebench [25] installation along with the required libraries. The total size of the image *without* the dataset is 2 MB. Depending on the workload, we created two images from the Filebench base image: a single-file image *Singlef* and a multi-file image *Multif*. The Singlef image contains a single 1 GB file while Multif includes about 16,000 64 KB files in a directory tree with an average depth of 3, similar to Filebench's Web-server workload. The total size of the Multif dataset is therefore also 1 GB.

For the microbenchmark experiments we used the same high-level workload: we simultaneously started 64 Docker containers and then waited for all of them to finish. The idea behind using such a workload was to resemble a

parallel job, such as a Spark job, which consists of a large number of equal small tasks. Each task is launched in its own container and operates on a different partition of the input data. We selected 64 containers assuming that every available core services 4 containers. We did not simulate memory usage.

In each experiment, all containers were created from the same image and each container ran until it performed 600 I/O operations. The I/O speed was rate-limited to 10 synchronous IOPS per container, reflecting a scenario when a containerized application is not particularly I/O intensive. Though per-container I/O accesses are infrequent, all containers together produce a significant load on the I/O subsystem. In the case of no I/O congestion, every container executes for exactly 60 seconds. However, in reality, every container took longer to finish (up to 576 seconds) because 64 containers competed for the limited storage bandwidth.

For the Singlef-image-based containers we experimented with two workloads: (1) random 4 KB reads from the 1 GB file (*sf-reads*) and (2) random 4 KB updates to the file (*sf-writes*). The corresponding two workloads for the Multif-image-based containers are: (1) whole-file reads from randomly selected 64 KB files (*mf reads*) and (2) whole-file overwrites of random 64 KB files in the dataset (*mf writes*).

4.3 Realistic workloads

In addition to the microbenchmarks, we also compared the different storage drivers for two realistic workloads (Experiment 4). The first workload compiles a recent Linux kernel (4.15.7) while the second one simulates a Ubuntu system upgrade by installing a large number of packages using:

```
apt-get -y -f install ubuntu-standard freeglut3-dev
```

The goal of these two workloads is to asses the behavior of the different storage drivers under more realistic conditions, in which the system experiences a mix of disk I/O, network I/O, and CPU load. The images for both workloads are based on the official `ubuntu` image and the image for the kernel compilation also contains the necessary build tools and kernel sources.

Below we present four experiments, each exploring a different dimension: storage driver, number of layers, storage device speed, and the behavior under realistic workloads. Results for these experiments are presented in Figs. 2, 3, 4, and 5 respectively.

We repeat every experiment enough times to make the measurements statistically sound. We plot the average completion time and the confidence interval with a confidence level of 95% as error bars for Figs. 3 and 4. Figure 2 contains a time-series graph where standard deviation is not applicable. We, however, verified that there was no significant variation between the results of different runs. The upgrade workload shown in Fig. 5b depends on the Internet connectivity to the public Ubuntu package repositories and, therefore, its performance variation is higher. For reproducibility, before starting an experiment, we deleted all old containers, trimmed and reformatted the SSD, restarted the Docker daemon, and cleared file system caches.

4.4 Experiment 1—drivers

In this experiment we monitored how, for different storage drivers, the total number of *running* containers changed over time (see Fig. 2). As is expected, for all drivers, the number of running containers first increased and then decreased again to zero by the end of the experiment, as all containers eventually finished. However, only Btrfs and ZFS started containers fast enough to run all 64 containers at a high speed for *every* workload. Aufs and Overlay2 were as fast as Btrfs and ZFS for all workloads except sf-writes as they need to perform expensive file-based CoW of the large 1 GB file in this workload. The latency of the very first write in a container in this case reached up to hundreds of seconds due to the high I/O pressure. Interestingly, the starting time of containers for this workload was not impacted in the Aufs configuration, but significantly increased in the Overlay2 configuration. This resulted in a faster completion of the experiment with the Aufs driver.

In terms of completion time, the dm driver under-performed for the sf-reads, mf-reads, and mf-writes workloads by about 20%. The graphs indicate that this was mainly caused by the slower setup and destruction of the writable layer by device mapper. However, for the sf-writes workload, dm showed better completion times than Aufs and Overlay2. This is expected because dm 's CoW granularity is a block. We also observe that dm 's performance took a more severe hit in sf-writes than in other workloads. We believe this is because dm has to first fetch 512 KB blocks before updating them due to *read-before-write* [26] requirements.

Probably because of their similar designs, ZFS and Btrfs performed equally well. However, due to the implementations differences (which we did not investigate) ZFS started the Singlef containers up to 30% slower.

For comparison, we also included the VFS driver in this experiment, which incurs unacceptably low performance. In fact, to avoid Docker timeouts and to be able to fit the resulting dataset on the SSD we had to limit the VFS driver experiments to 24 containers. When creating a writable layer for a container, the VFS driver copies all

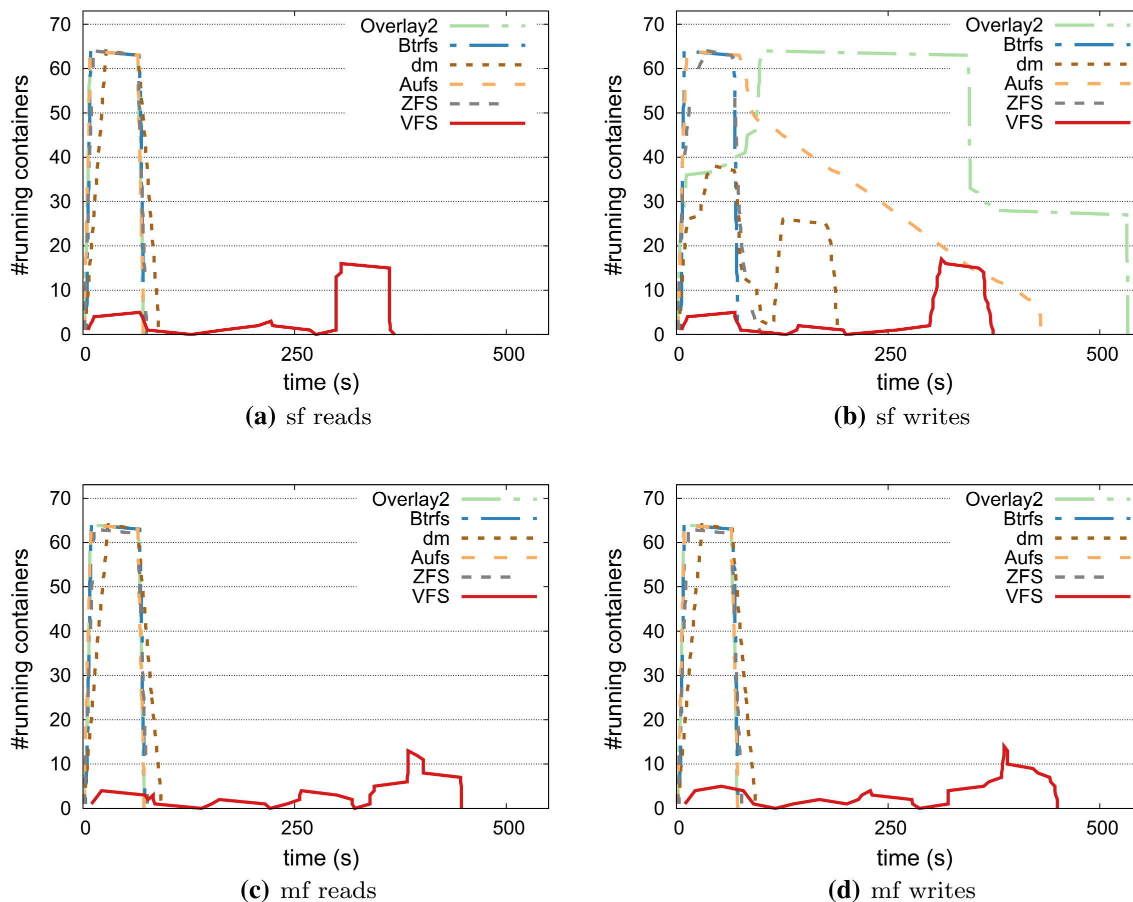

Fig. 2 Driver dimension: the number of running containers vs. time for different drivers and workloads

layers that belong to the corresponding image. The start of a Docker container requires the creation of two layers: one with configuration files specific to the container (e.g., /etc/hosts), and another one for the containerized application itself. Therefore, the start of a container results in copying 2 GB of data, irrespective of the workload. This clearly indicates that storage CoW capability is crucial for Docker's feasibility in practice. Because of its low performance, we do not evaluate the VFS driver further.

An interesting observation we made when running this experiment was that Overlay2 and Aufs copy a file when an application opens it with a O_RDWR flag, not when the first write is issued. By default, Filebench was opening files in O_RDWR mode, even for read-only workloads, which initially resulted in low performance for the read workloads with Overlay2 and Aufs. We modified Filebench to open files in O_RDONLY mode for read-only workloads. This experience shows that, first, applications have to be more careful when picking an open flag within containers. Second, if a containerized application performs both reads and writes, it is beneficial to segregate read-only data to one set of files, and write-only data to a different set of files, if possible. This will reduce the amount of data copied during the CoW process.

4.5 Experiment 2—layers

Next, we evaluate the impact of the number of layers within an image on performance. For this experiment, we created layered versions of the Singlef and Multif images. For Singlef, each layer overwrites 1% of the 1 GB file from a randomly chosen offset with random bytes. For Multif, each layer overwrites 1% of the 64 KB files with random bytes. We varied the number of layers from 1 to 30 and measured the completion time for each of the four workloads (see Fig. 3).

Previous work has shown that the number of layers does influence file access latencies [10]. Surprisingly, our results did not reveal significant dependency between the performance and the number of layers. For example, for ZFS under mf-writes workload, we observed that the completion times reproducibly increase from 79 seconds to 81 seconds (about 2.5%) between 10 and 30 layers. We experimented with higher percentages of inter-layer overwrites—5%, 10% and 20%—and still did not see

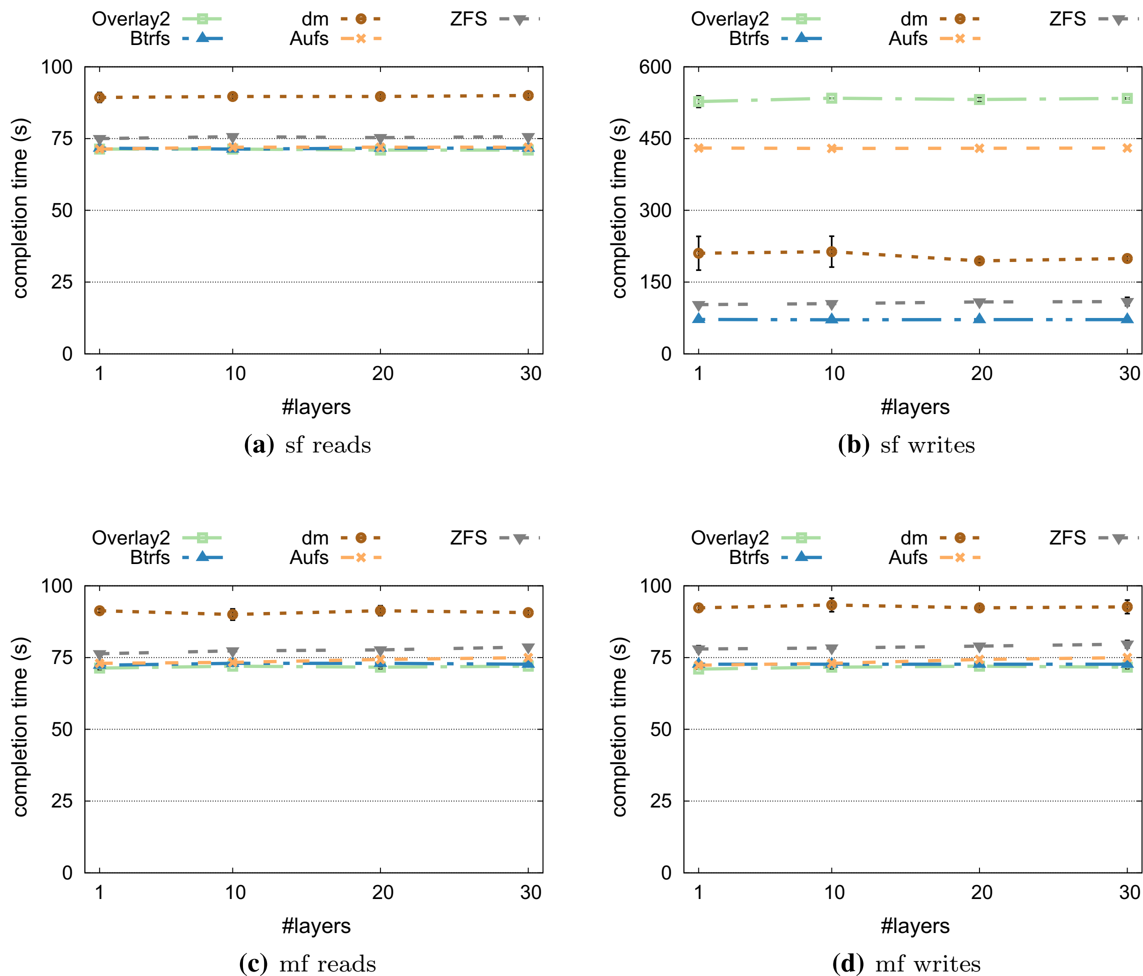

Fig. 3 Layer dimension: completion time versus the number of layers for different drivers and workloads

significant impact of the number of layers on performance. We believe that this is due to two reasons: (1) the overhead reported by [10] is only on the order of milliseconds and hence, does not have a significant impact on longer running workloads; (2) the overhead is only incurred for the first (uncached) access to a file which reduces its impact even further.

While this initial exploration does not indicate an impact of layers on performance, other interesting aspects of layering can be investigated such as its impact on storage utilization and caches. For example, we found that when a layered image is created on a client using a Dockerfile [27], the client does not fully benefit from the block-level CoW (in Btrfs, ZFS, and dm). E.g., the Singlef image with 10 layers was of 10 GB size despite the fact that only 10 MB of the file were changed in each layer. This happens because the image build process commits every layer, and every commit generates a layer at *file granularity* to exchange it with the Docker registry.

4.6 Experiment 3—devices

To experiment with a wide range of devices that cloud providers offer to a consumer, we used our previously developed dm-delay tool to create virtual devices with arbitrary latencies [28, 29]. We extended dm-delay to support configurable queue depths and sub-millisecond latencies; we present here the results for 0, 2, 4, and 6 millisecond latencies when using a queue depth of 10. We deployed dm-delay on top of the SSD used in the previous experiments. Fig. 4 highlights that in the majority of the cases, all drivers operate slower on slow devices. However, the impact when using the dm and ZFS drivers is much more severe than with other file-system-based drivers. For dm we think this is due to the fact that it does not benefit from the page cache of the underlying file system, as Aufs and Overlay2 do. For ZFS we speculate that the reason can be in that it does not utilize Linux page cache but rather uses its own ARC cache.

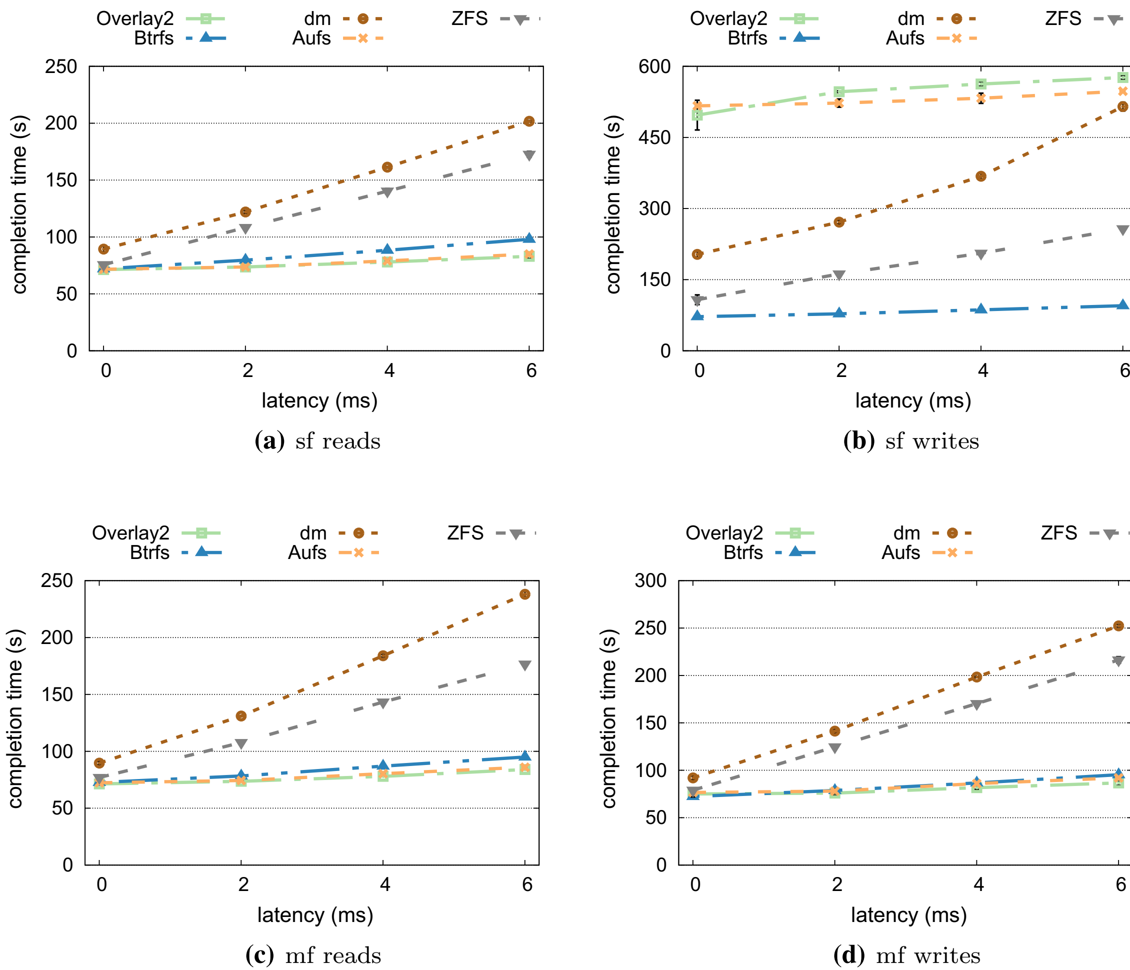

Fig. 4 Device dimension: completion time versus device latency for different drivers and workloads

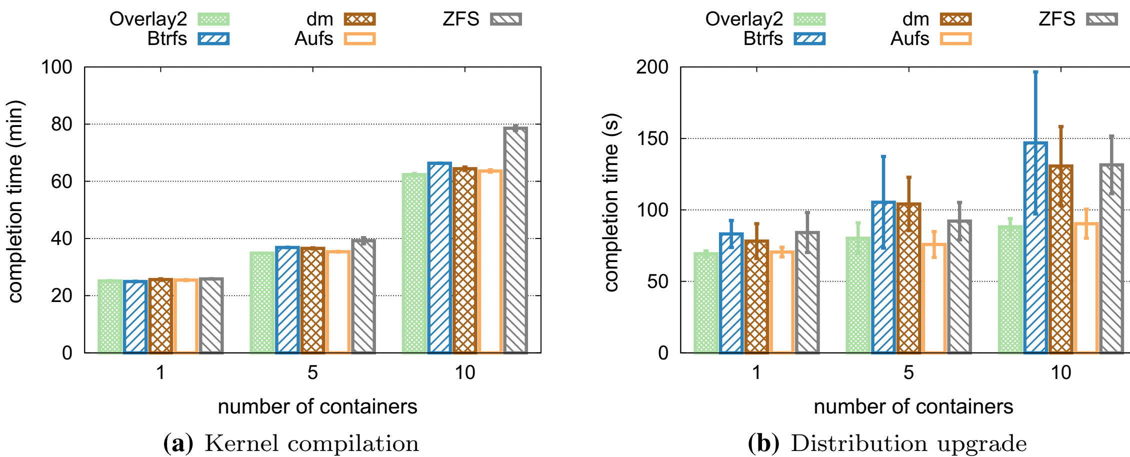

Fig. 5 Realistic workloads: the completion time of all tasks versus the number of concurrently running containers

In this experiment we also occasionally ran into a problem with Btrfs. At times, under write-intensive workloads, Btrfs reported out-of-space errors even though there still was available space on the device. There are multiple reports on the web related to this issue [30–32]. This contributed to our decision to mark Btrfs stability as low in Table 1.

4.7 Experiment 4—realistic workloads

Finally, we studied the behavior of the storage drivers for two realistic workloads: kernel compile and system upgrade. We assigned 4 cores to a single kernel compilation task. For each workload, we varied the number of concurrently running containers and plot the completion times in Fig. 5a, b.

For the kernel compile workload, we did not observe a significant difference between the storage drivers for a small number of containers. For 1 container, all drivers performed nearly identically. Interestingly, with an increasing number of concurrently running compilation tasks, we see that Btrfs has a slightly higher completion time compared to Overlay2, dm, and Aufs. ZFS gets even worse and experiences a slowdown of 1.23× compared to Aufs.

We can also see a large overall increase in completion time when moving from 5 to 10 containers. The reason is that for 5 containers, i.e. 5 compilation tasks, the machine is not CPU bound as all tasks together only utilize 20 out of 32 cores. For 10 containers, the machine now becomes CPU bound and has to time-slice CPU time between containers.

The system upgrade workload draws a similar picture. For 1 container, the difference between all storage drivers is small but increases for a larger number of concurrent containers. Additionally, Overlay2 and Aufs always perform the best, similar to the kernel compile workload. However, we observed an even bigger relative discrepancy of up to 1.66 × for Btrfs, ZFS, and dm compared to Overlay2 and Aufs. Additionally, these three storage drivers show a high variance.

The experiments reveal several interesting findings. Contrary to our microbenchmarks, Btrfs and ZFS seem to perform worse compared to other storage drivers under realistic workloads. This difference is smaller for kernel compile and becomes more pronounced for the upgrade workload. At the same time, these drivers also show a higher variance. There are several possible explanations for this behavior, for example, as Btrfs and ZFS are native file systems, they benefit less from the Linux page cache which may cause problems during a mixed read/write workload. It is also possible that Aufs and Overlay2 handle `sync()` operations (which are abundant during package upgrades) more efficiently. However, determining the exact reason for this behavior is left for future work.

4.8 CPU utilization

In experiments 1, 2, and 3, CPU utilization of the system was less than 5%, supporting the fact that the storage drivers generate I/O bound workloads. Therefore, the speed of CPU is not critical for the drivers' performance. As expected, kernel compilation and unpacking Ubuntu packages in Experiment 4 caused significant CPU utilization (up to 95%).

5 Related work

Docker provides some guidance on selecting a storage driver [19], but the recommendations are coarse grained and not supported with experimental results. There are also numerous non peer-reviewed postings online regarding container storage performance, but they tend to either not focus on storage [33] or they do not take the dimensions discussed in this paper into account [34–37].

Studies exist that compare the performance of containers to virtual machines [38] or investigate container provisioning times [39]. However, they do not consider the different storage options for Docker containers.

BetrFS improves the performance of copy-on-write file systems [40], but these optimizations have not been adapted to container workloads. While some additional work focuses on optimizing copy-on-write with containers [41, 42], it lacks a comprehensive evaluation and does not discuss the presented storage dimensions.

Some researchers suggested to replace local storage drivers with distributed storage drivers [10, 43]. Slacker mixes NFS with server-side cloning to improve performance [10]. Shifter combines all storage layers into a single file stored in a parallel file system [43]. Our work applies to these approaches as well, since the storage and file system are simply two dimensions that factor into container storage performance.

Finally, approaches exist to improve the management of container images. Exo-clones alter container image snapshots to be usable across different deployment environments [20]. CoMICon introduces a distributed, cooperative registry that allows Docker hosts to share images between each other [44]. This is complementary to our work, since our focus is on performance and not management or storage efficiency.

6 Conclusions

Docker containers have become one of the major execution environments for many different applications, producing a unique workload blend that exercises storage's writable snapshot capabilities like never before. There is a variety of generic solutions that support CoW snapshots, but their effectiveness for Docker containers is not well understood. First, it is not trivial to understand which

solution works best for Docker and in which situation. Second, the generic solutions were not designed with Docker in mind and may deliver suboptimal performance due to its unique I/O workload.

We demonstrated in this paper that Docker storage systems have many design dimensions and the design choices can impact Docker performance profoundly. At the same time, some dimensions that would, in theory, be expected to have a significant effect on performance (e.g., the number of layers in an image) do not have much impact in practice. We hope that the information and experimental results presented here will draw the deserved attention to this new, exciting, and largely unexplored area.

Acknowledgements This manuscript is an extended version of the paper titled “In Search of the Ideal Storage Configuration for Docker Containers” published in Proceedings of the Workshop on Autonomic Management of Large Scale Container-based Systems (AMLCS) in 2017. We thank the co-authors of the workshop paper Amit Warke, Dean Hildebrand, Mohamed Mohamed, and Nagapramod Mandagere for the contributions to the project. We also thank the AMLCS reviewers for their valuable comments. This work was supported in part by the NSF via Grants CNS-1563883, CNS-1320426, CNS-1562837, and CNS-1619653.

References

1. 451 Research: Application Containers Will Be a $2.7Bn Market by 2020. https://451research.com/images/Marketing/press_releases/Application-container-market-will-reach-2-7bn-in-2020_final_graphic.pdf. Accessed March 2018
2. Control Group v2. https://www.kernel.org/doc/Documentation/cgroup-v2.txt. Accessed March 2018
3. Biederman, E.: Multiple instances of the global Linux namespaces. In: Linux Symposium (2006)
4. Docker. https://www.docker.com/. Accessed March 2018
5. AUFS—Another Union Filesystem. http://aufs.sourceforge.net. Accessed March 2018
6. Overlay Filesystem. https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt. Accessed March 2018
7. Rodeh, O., Bacik, J., Mason, C.: BTRFS: The Linux B-Tree Filesystem. ACM Trans. Storage **9**(3), 9 (2013)
8. Bonwick, J., Ahrens, M., Henson, V., Maybee, M., Shellenbaum, M.: The Zettabyte File System. In: Proceedings of the 2nd Usenix Conference on File and Storage Technologies (FAST) (2003)
9. Device Mapper Thin Provisioning Targets. https://www.kernel.org/doc/Documentation/device-mapper/thin-provisioning.txt. Accessed March 2018
10. Harter, T., Salmon, B., Liu, R., Arpaci-Dusseau, A., Arpaci-Dusseau, R.: Slacker: Fast Distribution with Lazy Docker Containers. In: Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST) (2016)
11. Lamourine, M.: Storage Options for Software Containers. ;login: The USENIX Magazine **40**(1) (2015)
12. Docker Hub. https://hub.docker.com/. Accessed March 2018
13. Docker Swarm Mode. https://docs.docker.com/engine/swarm/. Accessed March 2018
14. Kubernetes. https://www.kubernetes.io/. Accessed March 2018
15. Juju. https://jujucharms.com/. Accessed March 2018
16. fabric8. https://fabric8.io/. Accessed March 2018
17. ZFS on Linux. http://zfsonlinux.org/. Accessed March 2018
18. Solaris Porting Layer. https://github.com/zfsonlinux/spl. Accessed March 2018
19. Select a Storage Driver. https://docs.docker.com/engine/userguide/storagedriver/selectadriver/. Accessed March 2018
20. Spillane, R.P., Wang, W., Lu, L., Austruy, M., Rivera, R., Karamanolis, C.: Exo-clones: Better Container Runtime Image Management Across the Clouds. In: Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage) (2016)
21. Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/. Accessed March 2018
22. Docker and OverlayFS in Practice. https://docs.docker.com/engine/userguide/storagedriver/overlayfs-driver/. Accessed March 2018
23. Aufs Returns ENOSUP when Writing in the Middle of a File on XFS. https://sourceforge.net/p/aufs/mailman/aufs-users/thread/27641.1496877655%40jrobl/. Accessed March 2018
24. ZFS on Linux for RHEL and CentOS. https://github.com/zfsonlinux/zfs/wiki/RHEL-and-CentOS. Accessed March 2018
25. Tarasov, V., Zadok, E., Shepler, S.: Filebench: A Flexible Framework for File System Benchmarking. ;login: The USENIX Magazine **41**(1) (2016)
26. Campello, D., Lopez, H., Useche, L., Koller, R., Rangaswami, R.: Non-blocking Writes to Files. In: Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST) (2015)
27. Dockerfile Reference. https://docs.docker.com/engine/reference/builder/. Accessed March 2018
28. Santana, R., Rangaswami, R., Tarasov, V., Hildebrand, D.: A fast and slippery slope for file systems. ACM SIGOPS Oper. Syst. Rev. **49**(2), 27–34 (2016)
29. Dm-delay. https://www.kernel.org/doc/Documentation/device-mapper/delay.txt. Accessed March 2018
30. Ubuntu Thinks Btrfs Disk Is Full but It Is Not. https://askubuntu.com/questions/464074/ubuntu-thinks-btrfs-disk-is-full-but-its-not. Accessed March 2018
31. Btrfs Out of Space, Even Though There Should Be 10% Left. https://superuser.com/questions/1096658/btrfs-out-of-space-even-though-there-should-be-10-left. Accessed March 2018
32. Btrfs: No Space Left on Device. https://nathantypanski.com/blog/2014-07-14-no-space-left.html. Accessed March 2018
33. Eder, J., Murphy, C.: Performance Analysis of Docker on Red Hat Enterprise Linux 7 (2014). https://developers.redhat.com/blog/2014/08/19/performance-analysis-docker-red-hat-enterprise-linux-7/. Accessed March 2018
34. Eder, J.: Comprehensive overview of storage scalability in Docker (2014). https://developers.redhat.com/blog/2014/09/30/overview-storage-scalability-docker/. Accessed March 2018
35. Estes, P.: Storage Drivers in Docker: A Deep Dive (2016). https://integratedcode.us/2016/08/30/storage-drivers-in-docker-a-deep-dive/. Accessed March 2018
36. Simon, R., von Eicken, T.: Perspectives on Docker (2014). https://www.slideshare.net/rightscale/docker-meetup-40826948. Accessed March 2018
37. Tyczynski, W.: 1000 Nodes and Beyond: Updates to Kubernetes Performance and Scalability in 1.2 (2016). http://blog.kubernetes.io/2016/03/1000-nodes-and-beyond-updates-to-Kubernetes-performance-and-scalability-in-12.html. Accessed March 2018
38. Sharma, P., Chaufournier, L., Shenoy, P.J., Tay, Y.: Containers and Virtual Machines at Scale: A Comparative Study. In: Proceedings of the 17th International Middleware Conference (Middleware) (2016)
39. Hegde, A., Ghosh, R., Mukherjee, T., Sharma, V.: SCoPe: A Decision System for Large Scale Container Provisioning Management. In: Proceedings of the 9th IEEE International Conference on Cloud Computing (CLOUD) (2016)

40. Jannen, W., Yuan, J., Zhan, Y., Akshintala, A., Esmet, J., Jiao, Y., Mittal, A., Pandey, P., Reddy, P., Walsh, L., Bender, M., Farach-Colton, M., Johnson, R., Kuszmaul, B.C., Porter, D.E.: BetrFS: A Right-Optimized Write-Optimized File System. In: Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST) (2015)
41. Zhao, F., Xu, K., Shain, R.: Improving Copy-on-Write Performance in Container Storage Drivers. In: Storage Developers Conference (2016)
42. Wu, X., Wang, W., Jiang, S.: TotalCOW: Unleash the Power of Copy-on-Write for Thin-Provisioned Containers. In: Proceedings of the 6th Asia-Pacific Workshop on Systems (APsys) (2015)
43. Canon, R.S., Jacobsen, D.: Shifter: Containers for HPC. In: Cray User Group (2016)
44. Nathan, S., Ghosh, R., Mukherjee, T., Narayanan, K.: CoMICon: A Co-Operative Management System for Docker Container Images. In: Proceedings of the 5th IEEE International Conference on Cloud Engineering (IC2E) (2017)

Vasily Tarasov is a Research Staff Member at IBM Research—Almaden. He currently works on data storage solutions for Cloud and high-performance computing, file-system-as-a-service methodology, performance analysis of complex systems, and distributed storage for Linux containers. Dr. Tarasov received his Ph.D. degree from Stony Brook University, M.S. degree from Moscow Institute of Physics and Technolgo, and worked as a kernel developer at Parallels Inc. prior to that.

Lukas Rupprecht is a Postdoctoral Researcher in the Storage Systems Group at IBM Research Almaden. His research interests are broadly related to distributed systems for data management including scalability, performance, fault tolerance, and manageability aspects. He is currently working on metadata management and supporting containerized workloads in distributed storage systems. Before joining IBM, he obtained a PhD degree from Imperial College London as part of the Large-Scale Data & Systems Group, where he worked on optimizing the interaction of data-parallel frameworks with the network by making applications network-aware. He holds an MSc and BSc degree in Computer Science from Technical University Munich (TUM).

Dimitris Skourtis is a Research Staff Member in the Storage Systems Research group at IBM Almaden Research Center. Prior to joining IBM, Dimitris worked on resource management and scheduling at VMware. Dimitris received his Ph.D. focusing on flash storage and predictable performance at UC Santa Cruz, where he was a member of the Systems Research Lab at the Department of Computer Science. Dimitris holds a Master in Mathematics (M.Math) from the University of St. Andrews in Scotland.

Wenji Li is a Ph.D. candidate in the VISA Laboratory of Arizona State University. Wenji's research focuses on Flash cache management and I/O scheduler optimizations.

Raju Rangaswami received a B.Tech. degree in Computer Science from the Indian Institute of Technology, Kharagpur, India. He obtained M.S. and Ph.D. degrees in Computer Science from the University of California at Santa Barbara where he was the recipient of the Dean's Fellowship and the Dissertation Fellowship. Raju is currently an Associate Professor of Computer Science at Florida International University where he directs the Systems Research Laboratory. His research interests include operating systems, storage systems, persistent memory, virtualization, and security.

Ming Zhao is an associate professor of the Arizona State University (ASU) School of Computing, Informatics, and Decision Systems Engineering (CIDSE), where he directs the research laboratory for Virtualized Infrastructures, Systems, and Applications (VISA, http://visa.lab.asu.edu). His research is in the areas of experimental computer systems, including cloud/edge, big-data, and high-performance systems as well as operating systems and storage in general. He is also interested in the interdisciplinary studies that bridge computer systems research with other domains. He received his bachelor's and master's degrees from Tsinghua University, and his Ph.D. from University of Florida.