Docker Characterization on High Performance SSDs

Qiumin Xu[1], Manu Awasthi[4], Krishna T. Malladi[2], Janki Bhimani[3], Jingpei Yang[2], Murali Annavaram[1]
[1]University of Southern California, [2]Samsung, [3]Northeastern University, [4]IIT-Gandhinagar

I. Introduction

Docker containers [2] are becoming the mainstay for deploying applications in cloud platforms, having many desirable features like ease of deployment, developer friendliness and lightweight virtualization. Meanwhile, solid state disks (SSDs) have witnessed tremendous performance boost through recent innovations in industry such as Non-Volatile Memory Express (NVMe) standards [3], [4]. However, the performance of containerized applications on these high speed contemporary SSDs has not yet been investigated. In this paper, we present a characterization of the performance impact among a wide variety of the available storage options for deploying Docker containers and provide the configuration options to best utilize the high performance SSDs.

II. Docker container storage options

The primary "executable" unit in Docker terminology is an *image*. Images are immutable files that are snapshots of the container states, including all the necessary application binaries, input files and configuration parameters to run an application within the container. Any changes to the container state during the execution are stored in a separate *diff* image. If a future container run requires to use the modified state, then both the base image and the image *diff* have to be read and combined to create the required state. The storage driver is responsible for managing these different images and providing a unified view of the system state.

There are three different approaches to store and manage the image and data in Docker.

a) Option 1: Through Docker Filesystem: In the first approach, Docker stores the data within the container using Union File System (UnionFS) and Copy-on-Write (CoW) mechanisms. Different storage drivers of path ❶ and ❷ in Figure 1 differ in the manner that they implement the unionizing file system and the CoW mechanism. Docker provides a variety of pluggable storage drivers that are based on Linux filesystem or volume manager, including Aufs, Btrfs, and Overlayfs. **Aufs** is a fast reliable unification file system with some new features like writable branch balancing. **Btrfs** (B-tree file system) is a modern CoW file system which implements many advanced features for fault tolerance, repair and easy administration. **Overlayfs** is another modern union file system which has a simpler design and is potentially faster than Aufs.

b) Option 2: Through Docker Volume Manager: **Devicemapper** is an alternate approach to unionizing file systems where the storage driver leverages the thin provisioning and snapshotting capabilities of the kernel-based Device Mapper framework. There are two different configuration modes for Docker host running devicemapper storage driver. The default configuration mode is known as **"loop-lvm"** shown in path 2.a of Figure 1. This uses sparse files to build the thin-provisioned pools used by storing images and snapshots. However, this mode is not recommended for production, and using block device to directly create the thin pool is preferred. The latter is known as **"direct-lvm"**, shown in path 2.b.

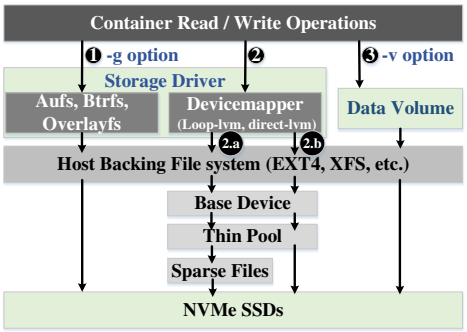


Fig. 1: Illustration of Storage paths in Docker

c) Option 3: Through Docker Data Volume: Data stored on Docker's data volume persists beyond the lifetime of the container, and can also be shared and accessed from other containers. This data volume provides better disk bandwidth and latency characteristics by bypassing the Docker file system, and is shown as the **-v** option in path ❸ in Figure 1. It also does not incur the overheads of storage drivers and thin provisioning. The main reason for using data volumes is data persistence. The rest of the paper explores the performance implications of these options.

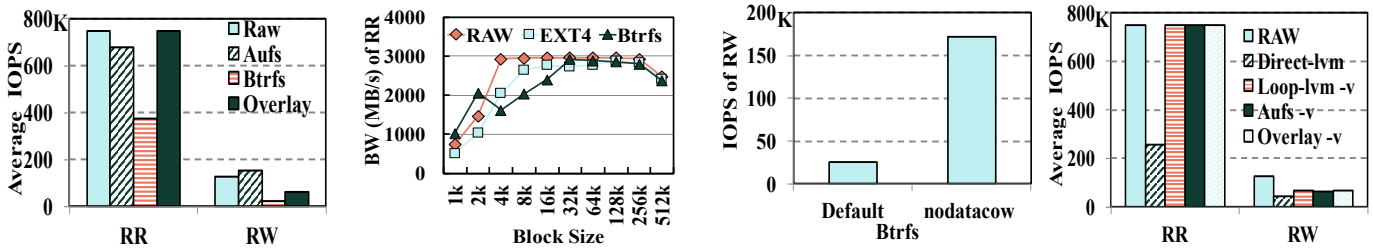
III. Container Performance Analysis

We provide characterization of multiple components of the file system stack using the flexible I/O tester (fio) tool available on Github. We use 1.5TB Samsung XS1715 NVMe SSD and Docker version 1.11.2. I/O traffic is synthetically generated using fio's asynchronous I/O engine, `libaio`. To quantify the performance impact of different file systems and Docker storage options, we run the same set of fio experiments from within the container as well as natively on the host, and then report the *steady state* performance. Due to space limitations, we will focus on analyzing our results on random reads/writes (RR/RW) under high load (32 jobs, 32 queue depth configured within the fio tool). 4 KB is used as the default block size and we report the IOPS for each run in isolation.

A. Performance Comparison of Docker Storage Drivers

We compare the performance implications of the different storage drivers offered by the Docker ecosystem. Data stored

978-1-5386-3890-3/17/$31.00 ©2017 IEEE



(a) IOPS comparison between storage drivers for RR and RW

(b) Sensitivity of Btrfs RR Bandwidth to block size of I/O requests

(c) Compare Btrfs RW IOPS w/ and w/o CoW feature

(d) Performance of using NVMe SSD as data volume: RR and RW

Fig. 2: Characterization of the throughput of Docker storage system using assorted combinations of request types (random reads or writes), file systems (ext4, xfs, btrfs), storage drivers (aufs, btrfs, overlayfs, loop-lvm, direct-lvm) and block sizes

though Docker storage drivers lives and dies with the containers. Therefore, this ephemeral storage volume is usually used in stateless applications that does not record data generated in one session for use in the next session with that user. For these experiments we created a Docker container that is configured with a specific storage driver used in that experiment. We then carried out file read and write operations from within the container. In these experiments, we vary the storage driver in use while keeping the backend device the same – Samsung XS1715 NVMe SSD. The NVMe device was configured with xfs file system (except for the btrfs driver) along with the specific Docker storage driver. Figure 2a present the results of aufs, btrfs and overlayfs compared to the raw block device performance. We find that aufs and overlay drivers can achieve performance similar to the raw block device for random read operations. Aufs even performs slightly better than raw block device for random write operations, because of its internal buffering. Btrfs on the other hand performs much worse than all the other options for random workloads.

In fact, the performance of btrfs is much lower for small block size read and write operations. Figure 2b shows a comparison among btrfs, ext4 and raw performance when changing the block size. We observed that btrfs achieves maximal random read performance when block size is increased to 32KB. Typically, btrfs operations have to read file extents before reading file data, adding another level of indirection and hence leading to higher performance loss for smaller block size. For random writes, the performance degradation is mostly due to the CoW overheads. As shown in Figure 2c, when we disable the CoW feature of btrfs, the random write performance of btrfs increased by 5X.

Our second important observation is the fact that in order to achieve performance closer to the rated device performance, *deterministically*, it is better to use the data volume rather than the storage driver. As shown in Figure 2a, the performance experienced by an application is dependent on the choice of the storage driver. However, in the case of the Docker volume, since the I/O operations are independent of the choice of the storage driver, the performance is consistently closer to that of raw block device. However, such an approach can potentially compromise repeatability and portability since the writes are persistent across container invocations.

B. Data Volume Performance

In order to test the performance implications of the data volumes, which allow for data persistence (by default), we tried a number of different experiments. The data file used as the target for fio is stored on the NVMe drive and is accessed using the default data volume, while the images are stored on the boot drive using the specified storage backend. In these experiments, we again vary the storage backend option while carrying out I/O intensive operations on the Docker volume. Using these experiments, we wanted to gauge the effects of the choice of the storage driver on the performance of operations for persistent data through the data volume.

Figure 2d presents the results of these experiments. For the most part, the performance of the persistent storage operations through the data volume are independent of the choice of the storage driver. An important observation is that the performance of I/O operations using data volumes matches closely with raw block device performance. This indicates that Docker containers only bring minimal performance overhead when using data volume for I/O operations. Direct-lvm, one of the recommended setups in Docker guidelines [1], however, performs poorly. The lvm, device mapper and the dm-thinp kernel modules which Direct-lvm are based upon, introduce additional code path and overheads that are not suitable for I/O intensive applications.

IV. Conclusion

In this paper, we provide an extensive characterization of Docker storage subsystem in the presence of high performance SSDs. We demonstrate that from a performance perspective, all storage backends for Docker are not created equal. In order to utilize the device to near saturation, it is better to use overlay or aufs backends. We finally show that the Docker volume is able to utilize these SSDs as efficiently as natively running applications.

References

[1] Docker, "Select a storage driver," https://docs.docker.com/engine/userguide/storagedriver/selectadriver/#future-proofing, 2017.

[2] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux J.*, vol. 2014, no. 239, 2014.

[3] NVM Express, "NVM Express – scalable, efficient, and industry standard," www.nvmexpress.org, 2016.

[4] Q. Xu, H. Siyamwala, M. Ghosh, T. Suri, M. Awasthi, Z. Guz, A. Shayesteh, and V. Balakrishnan, "Performance Analysis of NVMe SSDs and their Implication on Real World Databases," in *Proc. of SYSTOR*, 2015.