A Dockers Storage Performance Evaluation: Impact of Backing File Systems

Amer Ramadan[1]

[1] Faculty of Computer Sciences, Megatrend University, Belgrade, Serbia
Email: amer.cce@hotmail.com

Abstract
This paper reports on an in-depth examination of the impact of the backing filesystems to Docker performance in the context of Linux container-based virtualization. The experimental design was a 3x3x4 arrangement, i.e., we considered three different numbers of Docker containers, three filesystems (Ext4, XFS and Btrfs), and four application workloads related to Web server I/O activity, e-mail server I/O activity, file server I/O activity and random file access I/O activity, respectively. The experimental results indicate that Ext4 is the most optimal filesystem, among the considered filesystems, for the considered experimental settings. In addition, the XFS filesystem is not suitable for workloads that are dominated by synchronous random write components (e.g., characteristical for mail workload), while the Btrfs filesystem is not suitable for workloads dominated by random write and sequential write components (e.g., file server workload).

Keywords: docker, storage, filesystem, virtualization, containers, Linux

1. Introduction

The modern virtualization approaches can be classified as hypervisor-based or container-based. The latter approach, which is in practice mostly based on the Docker technology, introduces a whole new strategy in virtual machine usage [1]. Docker containers do not represent classical virtual machines, but can be defined as shrunk and changed operating system images, containing additional dependencies, applications and software needed for a proper functioning (e.g., code, system libraries and tools), cf. Figure 1. The name is completely in accordance to the purpose of supporting rapid development of applications in the context of a minimal OS images (so-called containers).

The research presented in this paper targets the container virtualization environment and container storage performances. A large body of related literature for hypervisor-based virtualization provides useful overviews of the performance testing for the common test environments [2-4]. In contrast to the hypervisor-based virtualization that has been relatively well elaborated, Docker containers as a relatively new technology still undergo research and development pressure. Thus, a number of recent studies provide comparison analyses of the Docker and frequently used hypervisors (e.g., KVM, Xen, ESXi). These studies

DOI: 10.5281/zenodo.5202497
Received: March 09, 2021 Accepted: July 19, 2021

primarily relate to evaluation of computational capacities, network and storage performances, by applying a set of benchmark scenarios (generated with, e.g., LinPack, Sysbench, OpenFOAM) [5-12]. In order to evaluate storage drivers performances, the research efforts are devoted to building images, and generating and managing containers, along with a number of specific file operations [13-15]. However, due to the specific Docker filesystem organization, it is not always possible to apply the available filesystem benchmarks. Therefore, there is a need for an alternative methodology for a comprehensive performance evaluation.

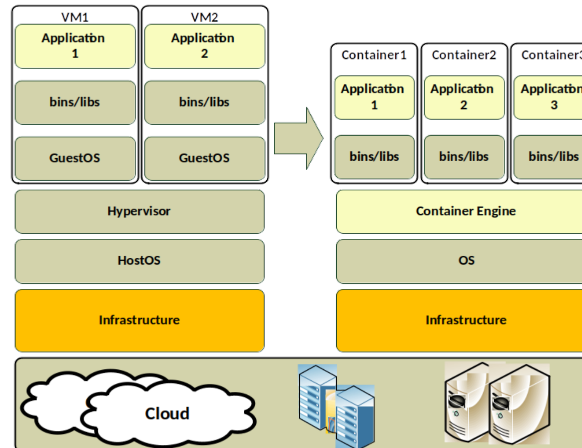

Figure 1 Hypervisor-based versus container-based virtualization

The paper addresses this desideratum. Its main contribution is a comprehensive examination of the impact of the backing filesystems to Docker performance in the context of Linux container-based virtualization – which differentiates our approach from other storage driver performance-based studies [13-15]. From the pool of storage drivers that support the Linux CentOS environment, we select the following, rather specific, storage drivers: (1) overlay, a union filesystem with file-based copy-on-write (CoW) method, and (2) Btrfs, a snapshot filesystem with block-based CoW method. Related to backing filesystems, we consider the following: Ext4, XFS and Btrfs. To evaluate the file system performance, we use the Filebench tool, and generate the following test scenarios: (1) web server workload scenario dominated by random read components; (2) e-mail scenario dominated by random read and random write components; (3) file server scenario in which both random and sequential components are equally present, and (4) random file access scenario dominated by random read and random write components.

2. Data Management in Docker Environment

All files generated or modified inside a container are stored in a writable container layer, which is strongly coupled to the host machine. The writing operation is performed by a storage driver. To achieve data persistency, the Docker technology provides two possibilities of storage to a host machine: Volumes and Bind Mounts. In both approaches, data are stored as directories of a container filesystem (cf. Figure 2). Bind Mounts may be stored anywhere on a host system, and rely on the host machine's filesystem, allowing thus for specific directory structures. In contrast to Bind Mounts, Volumes are created and managed by Docker, which makes them more appropriate for achieving data persistency.

DOI: 10.5281/zenodo.5202497
Received: March 09, 2021 Accepted: July 19, 2021

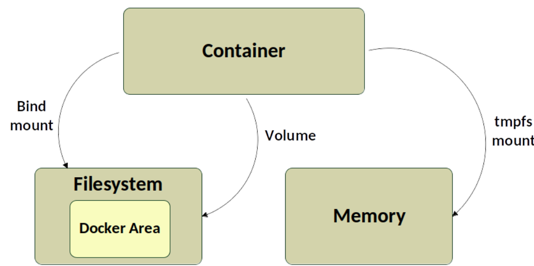

Figure 2 Volumes and bind mounts

For the purpose of managing contents of writable containers and image layers, specific storage drivers are used. Although there are differences between them, every storage driver type relies on stackable image layers and the CoW technique. This technique is characteristically used for sharing and copying files which are intended for further modification in a container layer, whereas the sequence of steps depends on a storage driver type. For instance, in the aufs, overlay and overlay2 drivers, the CoW technique is performed in the following steps: image layers are searched through in order to find a file to be updated, and when the file is found, it is copied-up onto the container writable layer. Further modifications (i.e., initial write request) are carried over the created copy of the file. The ZFS, Btrfs, and other driver types perform the CoW technique in a different way [16]. In the case of writing a large amount of data, containers are consuming more space, which leads to a significant performance overhead of the copy-up operation.

This implies a need for a special type of Docker pluggable architecture. The decision on the architecture is based on the following considerations:

(1) Does the kernel support single or multiple storage drivers? This consideration is important for the aufs, devicemapper, vfs, overlay, overlay2, Btrfs, and ZFS drivers.

(2) What are the inherited limitations of the Docker edition, operating system, and distribution?

(3) Does the storage driver use a particular format for the backing filesystem? For example, the overlay, overlay2 and aufs drivers support the Ext4 and XFS filesystem; the devicemapper driver support direct-lvm filesystem, while the Btrfs and ZFS drivers rely on their respective proprietary formats.

(4) What is the expected workload? In general, the aufs, overlay, and overlay2 drivers are suitable for workload operations at the file level rather than at the block level. These drivers allow for efficient memory consumption, with a potentially large growth of the container writable layer in scenarios with write-heavy workloads. To address the issue of write-heavy workloads, it is recommended to consider block-level storage drivers, among which the mostly used are devicemapper, Btrfs, and ZFS.

In this research, we use the CentOS operating system. It supports the overlay, devicemapper and Btrfs storage drivers, but in we consider the performances of the overlay and Btrfs drivers.

The overlay storage driver uses OverlayFS, a filesystem that provides a unified view of two different directories, i.e., of an image layer and a container layer. In this architecture when two layers contains the same files, the files in the container layer are considered more dominant. A container mount is created by combining a read-only image layer and writable container layer, cf. Figure 3. [17].

DOI: 10.5281/zenodo.5202497
Received: March 09, 2021 Accepted: July 19, 2021

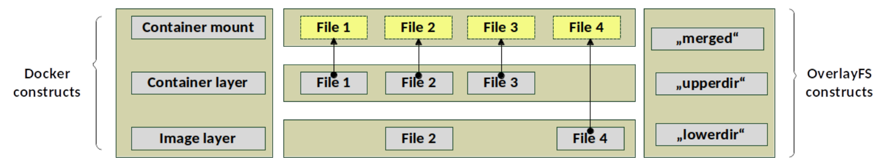

Figure 3 Docker process of constructing the map to OverlayFS

In the Btrfs filesystem, the only true subvolume is the base layer of an image. The other layers, stored as space-efficient snapshots, contain only the differences with respect to the base layer. The CoW technique is applied in order to fully exploit the storage efficiency capacities and to minimize the layer size. The writes are managed at the block level [18].

Linux has a rich support for UNIX-like filesystems (e.g. Ext2, Ext3, Ext4, ReiserFS, JFS, XFS, WAFL, Btrfs, ZFS, etc.). In our research, we focus on three 64-bit filesystems: Ext4, XFS, and Btrfs. The main reason underlying this decision is that all of them are modern, extent-based and commonly used in Linux environment. In addition, these filesystems are the only candidates for backing FS in our Docker storage driver environment.

Ext4 is a native Linux filesystem developed to resolve the capability and scalability issues of its predecessor (ext3 FS) [19], caused by double and triple indirect block mappings. Ext4 uses extents, which stand for a number of continuous physical blocks that can enhance a large file performance and reduce fragmentation. It relies on an efficient tree-based index, and represents files and directories in the form of H-trees. This filesystem applies a write-ahead journal to ensure operation atomicity. The checksumming operation is performed on the journal data only, while skipping the user data [20].

XFS is ported to Linux in 2001, as an originally developed native Silicon Graphics IRIX filesystem [21, 22]. It allocates the space by means of extents, with data structures stored in B+ trees. The free extents are located by a dual indexing feature, i.e., they are indexed by the size and the starting block of the free extent, respectively. The feature of delayed allocation efficiently prevents the filesystem fragmentation. The meta-data journaling and write barriers ensures the data consistency, even though the snapshot feature is not supported.

The B-tree filesystem (Btrfs) represents a native CoW Linux filesystem, based on a forest of CoW friendly B-trees [23]. It is designed to offer better data integrity characteristics along with the efficient storage management [24]. Btrfs is proposed as a solution to a number of scalability problems, mainly for larger and faster storage, lack of pooling, snapshots, checksums, integral multi-device spanning and built-in RAID support [25]. The CoW friendly B-trees uses the standard B+tree construction. It relies on the top-down update procedure, removes leaf-chaining, and uses lazy reference-counting for space management [24]. Due to the CoW feature, XFS may provide the self-healing functionalities in some configurations. Disk blocks are managed in extents, with checksumming for integrity, and reference counting for space reclamation [16].

3. Performance Costs

3.1 Data Path in Docker Virtualized Environment

Performance characteristics of filesystems in Docker environments depend on a large number of factors. Figure 4 provides an overview of the overall data path of the workload. A benchmark running in a Docker container generates workload which can be denoted as a function of requests generated by a benchmark software and the Docker container. The output workload generated by container is redirected to the Docker

DOI: 10.5281/zenodo.5202497
Received: March 09, 2021 Accepted: July 19, 2021

engine, and mapped onto the Bind Mount (BM) or Volume directory requests, which serve as input workload of the host operating system that generates requests to disk drivers.

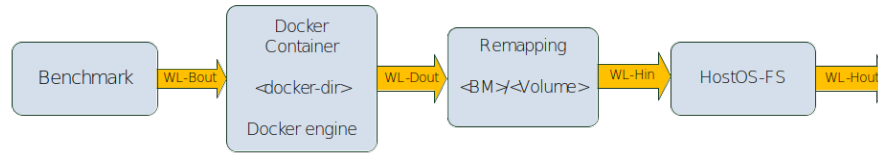

Figure 4 Data path in Docker virtualized environment

A benchmark provides a framework for the generation of input workload, specifying the values of the following parameters: the total number of workload files, the mean tree depth, the average file size, the read/write block size, and the mean size of appending block. It takes a set of input files, and applies a sequence of different file operations – such as opening, creation, deletion, reading, writing, and appending – in order to generate benchmark output workload. This workload is applied to the Docker, thus generating the Docker output workload, WL-Dout, consisting of the following: file block, i-node, extent, free-list and directory read/write operations within the Docker directory. WL-Dout is mapped onto a Bind Mount or Volume directory, denoted as WL-Hin, which is further applied to the hostOS filesystem, generating a final sequence, WL-Hout, consisting of: file block, i-node, extent, free-list and directory read/write operations in the host filesystem.

3.2 Modeling the Workload and Filesystem Performance

Workload *Tw* can be conceptualized in three different manners. In equation (1), Tw represents the total time in a Docker environment:

$$T_W = D_T(D_P, E_R) + HostOS_{FS_P}, \quad (1)$$

where the D_T represents the Docker time component, and $HostOS_{FSp}$ represents the Host OS filesystem processing time component. The first component D_T includes two sub-components, the first of which represents Docker processing component (D_P), and the second represents the engine remapping component (E_R). The latter component refers to the remapping of the objects from a Docker directory onto a Volume or Bind Mounts directory. This mapping represents a one-to-one correspondence between the shared directories. The $HostOS_{FSp}$ component refers to the final processing on the HostOS filesystem directory. In our research, the benchmark framework does not use files stored in container image layers, which makes the impact of storage drivers negligible. Consequently, in all the observed competitive experimental settings, the values of D_P and E_R are very similar, primarily due to the aforementioned one-to-one correspondence between the shared directories.

In Equation (2), the workload is observed at the access level, targeting different access types: read/write and random/sequential. The performance characteristics of a workload are evaluated with respect to the total time required to complete random and sequential read and write operations [18], where time components are denoted as T_{RR}, T_{SR}, T_{RW} and T_{SW}, respectively:

$$T_W = T_{RR} + T_{SR} + T_{RW} + T_{SW}. \quad (2)$$

In Equation (3), term T_W is interpreted through the impact of different data structures. The performance characteristics of a workload are evaluated with respect to the total time required to complete all directory related operations, all meta-data operations, all free lists operations, direct file block operations, journaling

DOI: 10.5281/zenodo.5202497
Received: March 09, 2021 Accepted: July 19, 2021

operations and housekeeping in the filesystem, where time components are denoted as T_W, T_{DIR}, T_{META}, T_{FL}, T_{FB}, T_J and T_{HK}, respectively.

$$T_W = T_{DIR} + T_{META} + T_{FL} + T_{FB} + T_J + T_{HK} \quad (3)$$

Table 1 provides a summary decomposition of performance costs. For the purpose of easier referencing in the rest of the paper, we introduce labels of a general form *Cx.y* to denote performance costs, where *x* relates to an operation, and *y* to a filesystem.

Table 1 Performance Costs Summarized

Operation / FS	Ext4	Cost	XFS	Cost	Btrfs	Cost
Update method (writing)	Overwrite	C1.1	Overwrite	C2.1	CoW	C3.1
Directory operations	H-tree	C1.2	B+ tree	C2.2	B+ tree	C3.2
Meta-data operations	Linear i-node table	C1.3	B+ tree	C2.3	B+ tree	C3.3
Free lists operations	Linear bitmap	C1.4	B+ tree	C2.4	B+ tree	C3.4
Direct file block access	H-extent-tree	C1.5	B+ tree	C2.5	B+ tree	C3.5
Housekeeping	Journaling CRC	C1.6	Journaling CRC	C2.6	Data, meta-data and journaling CRC	C3.6
Small file embedding	None	C1.7	Moderate performance	C2.7	Good performance	C3.7

4. Experimental Evaluation

The experiments reported in this paper were conducted on a dual core Intel Xeon E3110 @ 3.00GHz processor, with 8GB RAM and 500GB SATA-3 hard disk (7200 rpm, 6Gb/s). Centos 7.4 with Linux kernel 3.10.0-862.6.3.el7.x86_64 was selected as a native host for Docker containers.

The experimental design was a 3x3x4 arrangement, i.e., we considered three different numbers of Docker containers, three filesystems, and four application workloads. Due to the relatively small amount of available RAM (8GB), the experiments included one, two or three Docker containers. As mentioned in Section 2, we focus on three 64-bit filesystems: Ext4, XFS, and Btrfs. Finally, Docker random and sequential performances were tested in the Filebench benchmark environment, with respect to four different application workloads [19]:

- Web-workload emulates a simple Web server I/O activity. It produces a sequence of open-read-close operations over multiple files in a directory tree, finalizing with an appended log file.
- Mail-workload emulates an I/O activity of a simple e-mail server that stores each e-mail in a separate file. The workload consists of a multi-threaded set of operations in a single directory.
- Fileserver-workload emulates a simple file server I/O activity. This workload generates a large number of random and sequential reads/writes of files, as well as random writes for the appending operations.
- RandomFileAccess-workload emulates a random I/O activity.

DOI: 10.5281/zenodo.5202497
Received: March 09, 2021 Accepted: July 19, 2021

The experiments were conducted as follows. The Filebench environment runs in the HostOS directory, through a Docker container. The Docker area is mounted on a specific location, i.e., /var/lib/docker, which was also and this is the location for Volume and Bind Mounts directories. BM can be anywhere, while in our case it is also in /var/lib/docker (Docker area). Therefore, the possible combinations for Docker combined with BM or Volume test routines are: (1) Ext4 and Ext4, (2) XFS and XFS, (3) Btrfs and Btrfs.

The obtained results are given in Tables 2-5, each of which corresponds to an application workload. Data in these tables refer to MB/s and KB/s.

The Web-workload experimental results are provided Table 2. Although the Ext4 filesystem provides the best performance, the Btrfs filesystem is only slightly weaker. In addition, there are relatively small differences between Volume and Bind Mounts. In the settings with 1- and 2-Docker containers, Volume performs slightly better than Bind Mounts, while in the settings with 3-Docker containers, Bind Mounts perform better than Volume. Related to the direct file block access performance costs (C1.5, C2.5, C3.5, cf. Table 1), it should be noted that H-trees perform better than B+trees, contrary to the expectations.

Table 2 Web-workload results

No. of Dockers	**Ext4**		**XFS**		**Btrfs**	
	Volume	BM	Volume	BM	Volume	BM
1	108.9	108.6	107.3	108.23	101.67	101
2	55.05	55.08	55.01	54.2	53.43	53.73
3	37.38	37.24	35.63	37.06	35.06	37.04

The Mail-workload experimental results are provided in Table 3. The results indicate that the Ext4 filesystem provides the best performance, while the XFS filesystem provides the worst performance. For each filesystem separately, Volume and Bind Mounts perform similarly. Related to Docker containers, in the settings with 1-Docker and 3-Docker containers, Bind Mounts performs slightly better than Volume, while in the settings with 2-Docker containers, Volume performs slightly better. In addition, it can be observed that although B+trees accelerate read operations, the Ext4 filesystem, with its simple methods and the file caching feature, provides better results, primarily due to the presence of the synchronous random write component. Also, the Btrfs filesystem suffers from an inconvenient CoW method and intensive Housekeeping operations, while synchronous random writes cause the worst performance of the XFS filesystem.

Table 3 Mail-workload results

No. of Dockers	**Ext4**		**XFS**		**Btrfs**	
	Volume	BM	Volume	BM	Volume	BM
1	3.6	3.83	2.03	2.1	3.33	3.47
2	2.95	2.43	1.15	1.13	2.32	2.33
3	1.79	1.83	0.83	0.82	1.7	1.79

The Fileserver-workload experimental results are provided in Table 4. The results show that the Ext4 filesystem provides the best performances, due to the inherently simple methods, use of H-tree and specific file cashing features. The XFS filesystem provides moderate performance, while the Btrfs filesystem

DOI: 10.5281/zenodo.5202497
Received: March 09, 2021 Accepted: July 19, 2021

provides the worst performance, due to the CoW method (C3.1) and intensive Housekeeping operations (C3.6).

Table 4 Fileserver-workload results

No. of Dockers	Ext4		XFS		Btrfs	
	Volume	BM	Volume	BM	Volume	BM
1	168.03	166.13	108.23	110.93	79.7	82
2	87.21	90.57	54.2	66.02	33.02	41.87
3	62.66	64.16	46.06	56.21	18.9	19.31

Again, the results show that the Ext4 filesystem provides the best performances, the XFS filesystem provides moderate performance, and the Btrfs filesystem provides the worst performance. The differences in their performances is mostly obvious in the experimental settings with 1-Docker container, and becoming less emphasized with the increase of the number of containers. Bind Mounts performs slightly better than Volume, independently of the number of Docker containers.

Table 5 RandomFileAccess-workload test results

No. of Dockers	Ext4		XFS		Btrfs	
	Volume	BM	Volume	BM	Volume	BM
1	4356.63	4356.53	4131.8	4162.43	4044.93	4137.97
2	2268.53	2372.85	2099.6	2292.2	2072.67	2208.88
3	1600.62	1682.12	1539.73	1604.67	1315.64	1391.633

Figure 5 depicts the experimental results. The Web-workload experimental results are given in the upper left part of the figure, the Mail-workload test results are given in the upper right part of the figure, the Fileserver-workload experimental results are given in the lower left part of the figure, and the RandomFileAccess-workload experimental results are given in the lower right part of the figure.

DOI: 10.5281/zenodo.5202497
Received: March 09, 2021 Accepted: July 19, 2021

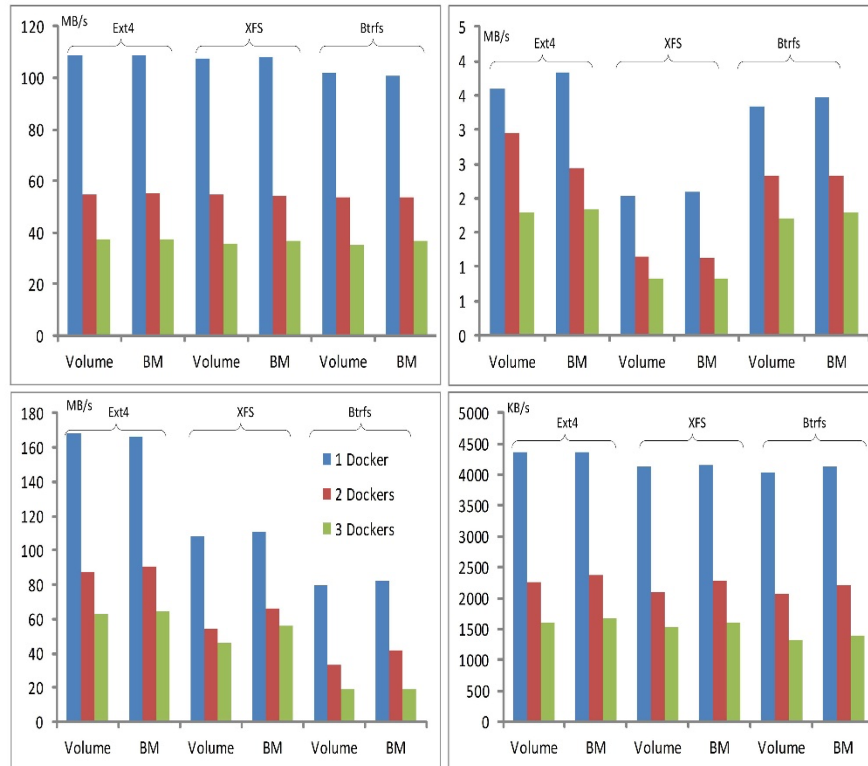

Figure 5 Optimal results on the validation set

5. Conclusions

In this paper, we reported on an comprehensive examination of the impact of the backing filesystems to Docker performance in the context of Linux container-based virtualization. The experimental design was a 3x3x4 arrangement, i.e., we considered three different numbers of Docker containers (one, two and three containers), three filesystems (Ext4, XFS and Btrfs), and four application workloads related to Web server I/O activity, e-mail server I/O activity, file server I/O activity and random file access I/O activity, respectively. The experimental results were provided and discussed. In general, Ext4 is the most optimal filesystem for the considered experimental settings. The XFS filesystem is not suitable for workloads that are dominated by synchronous random write components (e.g., mail workload), while the Btrfs filesystem is not suitable for workloads dominated by random write and sequential write components (e.g. file server workload).

References

[1] *Docker documentation*. https://docs.docker.com/. Accessed: 2016-02-25.

[2] T. Deshane, Z. Shepherd, J. Matthews, M. BenYehuda, A. Shah, B. Rao: *Quantitative Comparison of Xen and KVM*. 2008 Xen Summit, Berkeley, CA, USA, USENIX Association, 2008.

[3] B. Đorđević, N. Maček, V. Timčenko: *Performance Issues in Cloud Computing: KVM Hypervisor's Cache Modes Evaluation*. Acta Polytechnica Hungarica 12, 4 (2015), pp. 147-165.

[4] Tom Collins: *Hyper-V vs. VMware: Which Is Best?*, https://www.atlantech.net/blog/hyper-v-vs.-vmware-which-is-best

[5] D. Armstrong, K. Djemame: *Performance Issues in Clouds: An Evaluation of Virtual Image Propagation and I/O Paravirtualization*. The Computer Journal 54, 6, (2011), pp. 836-849.

[6] J. Bacik. 2012. Btrfs: *The Swiss Army Knife of Storage*. Usenix Login 37, 1, (2012), 7-15.

DOI: 10.5281/zenodo.5202497
Received: March 09, 2021 Accepted: July 19, 2021

[7] J. Che, Q. He, Q. Gao, D. Huang: *Measuring and Comparing of Virtual Machine Monitors. In Proceedings of the 5th International Conference Embedded and Ubiquitous Computing*. EUC2008, Vol. 2, Piscataway, NJ, USA, 2008, pp. 381–386.

[8] D. Comer: *Ubiquitous B-tree*. ACM Computing Surveys (CSUR) 3, 4 (2008), 2. 1979.

[9] T. Cormen, C. Leiserson, R. Rivest, C. Stein: *Introduction to Algorithms* (2nd. ed.). MIT Press and McGraw-Hill, Chapter 18: B-Trees, 2001.

[10] T. V. Do: *Comparison of Allocation Schemes for Virtual Machines in Energy-Aware Server Farms*. The Computer Journal 54, 11, (2011), pp. 1790-1797.

[11] B. Đorđević, N. Maček, V. Timčenko: *Performance Issues in Cloud Computing: KVM Hypervisor's Cache Modes Evaluation*. Acta Polytechnica Hungarica 12, 4 (2015), pp. 147-165.

[12] *Filebench*, SourceForge, https://sourceforge.net/projects/filebench/.

[13] S. Kumpf: *Docker storage drivers overview*, 2017, https://community.hortonworks.com/articles/87949/docker-storage-drivers-overview.html

[14] C. Kuehl: *Docker storage driver benchmarks*, 2017, https://github.com/chriskuehl/docker-storage-benchmark

[15] Q. Xu, M. Awasthi, K. T. Malladi, J. Bhimani, J. Yang, M. Annavaram: *Performance Analysis of Containerized Applications on Local and Remote Storage*, 2017. http://storageconference.us/2017/Presentations/PerformanceAnalysisOfContainerizedApplications-slides.pdf

[16] IBM Coorporation. 2012. *Kernel Virtual Machines (KVM): Best practices for KVM (second edition)*.

[17] *Docker and overlayfs in practice*. Docker Inc. 2018. https://docs.docker.com/engine/userguide/storagedriver/overlayfs-driver/

[18] *Use the BTRFS storage driver*. Docker Inc. 2019, https://docs.docker.com/storage/storagedriver/btrfs-driver/

[19] K.V. Kumar, A. M. Cao, J.R. Santos, A. Dilger: *Ext4 block and inode allocator improvements*. In Linux Symposium, 1, 2008.

[20] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Dilger, and L. Vivier: *The new Ext4 filesystem: current status and future plans*. In Proceedings of the Linux Symposium, l, 2, 2007, pp. 21-33.

[21] M. Holton, and R. Das: *XFS: a next generation journalled 64-Bit filesystem with guaranteed rate I*. SGI Corp. 1995.

[22] Silicon Graphics Inc. 2006. *XFS Filesystem Structure, Documentation of the XFS filesystem on-disk structures*.

[23] O. Rodeh: *Deferred Reference Counters for Copy-On-Write B-trees*. Technical Report rj10464, IBM Corporation, 2010.

[24] O. Rodeh: *B-trees, shadowing, and clones*. ACM Transactions on Storage (TOS) 3, 4 (2008), 2.

[25] O. Rodeh, J. Bacik, and C. Mason. 2013. *BTRFS: The Linux B-tree filesystem*. ACM Transactions on Storage (TOS) 9, 3 (2013), 9.

DOI: 10.5281/zenodo.5202497
Received: March 09, 2021 Accepted: July 19, 2021