Electronic Preprint for Journal of Information Processing Vol.28

Regular Paper

When the Virtual Machine Wins over the Container: DBMS Performance and Isolation in Virtualized Environments

Asraa Abdulrazak Ali Mardan[1,a) Kenji Kono[1,b)

Received: September 10, 2019, Accepted: April 8, 2020

Abstract: Database management systems (DBMSs) are a common service in the cloud. Cloud providers employ virtualization to consolidate DBMSs for efficient resource utilization and to isolate collocated DBMS workloads. To guarantee the service-level agreement (SLA), the disk I/O performance and its isolation are critical in DBMSs because they are inherently disk I/O intensive. This paper investigates the disk I/O performance and its isolation of MySQL on two major virtualization platforms, KVM and LXC. KVM provides the hypervisor-based virtualization (virtual machines) while LXC provides the operating-system-level virtualization (containers). Containers are widely believed to outperform virtual machines because of negligible virtualization overheads, while virtual machines are to provide stronger performance isolation. Contrary to the general belief, our results show that KVM outperforms LXC by up to 95% in MySQL throughput without compromising the isolation. Furthermore, LXC violates the performance isolation; a container given 30% share of the disk bandwidth consumes the 70% share. Our analysis reveals that the file system journaling, which is mandatory to maintain file system consistency, has negative impact on both the performance and its isolation in the container. Because of the sharing of the journaling module in containers, the journaling activities are serialized and bundled with each other, resulting in poor performance and violation of the performance isolation.

Keywords: virtualization, container, DBMS, journaling file system.

1. Introduction

Virtualization technologies are the building block of cloud datacenters. They are widely used in today's datacenters to achieve server consolidation for efficient resource usage, less power conusumption, and to isolate collocated applications on a single machine. There are two virtualization approaches: *operating system (OS)*- and *hypervisor*-based virtualization. Hypervisor-based virtualization virtualizes hardware resources to provide the abstraction of virtual machines (VMs). This allows multiple OSes to co-exist on a single machine. OS-based virtualization also known as *containerization* creates multiple virtual units in the userspace, known as *containers* in which processes run directly on the host OS. Containers share the same host kernel in opposite of VMs but are isolated from each other through abstraction mechanisms of Linux namespaces and Cgroup resource control at the OS level.

Containers are lightweight and bring the performance advantage over VMs. However, these advantages have some shortcomings. Since containers share the same kernel components like "buffer caches" and other the data structures, the performance isolation among containers becomes weak and hard to accomplish. For example, if one container accesses many files to increase the pressure on a shared buffer cache, other containers suffer from performance degradation because less cache is allocated for them. On the other hand, VMs have stronger performance isolation because no kernel components are shared among VMs.

The performance and isolation are important in virtualized environments, where multiple users' workloads are consolidated on the same machine. Users expect their workloads run in isolated manner and to get the performance they pay for. Failing to achieve isolation results in performance degradation. This leads to SLA violation which ultimately results in financial loss for the cloud providers [1], [2].

This work investigates disk I/O performance and isolation in LXC [3] and KVM [4], which are representatives of OS- and hypervisor-based virtualization, respectively. Disk I/O performance and isolation are critical in database management systems (DBMSs) which are a common service in the cloud. Many cloud-based services such as Dropbox [5], Facebook [6] and Saleforce [7] make use of DBMS. Microsoft's SQL Azure [8] and Google Cloud SQL [9] provide DBMS as a cloud service. DBMS makes excessive use of disk resource by orchestrating a large number of I/O operations and causing journaling updates to ensure consistency. Journaling is indispensable for recovering from unexpected crashes in file systems, and can not be turned off in DBMS.

This paper presents an updated and extended DBMS performance and isolation comparison of LXC and KVM. Our previous work [10] examines MySQL performance and isolation under the old function of `Cgroup` disk I/O control [11]. This work adopts

[1] Keio University, Yokohama, Kanagawa 223–8522, Japan
[a) asraaiteng@sslab.ics.keio.ac.jp
[b) kono@sslab.ics.keio.ac.jp

© 2020 Information Processing Society of Japan

the newer fuction of `Cgroupv2` [12] for disk I/O control, and considers the up-to-date software version of KVM, LXC, and Linux kernel.

We have extended the previous paper by first, examining how the `Cgroup` in general achieves the disk I/O control in a separate experiment. Second, invistigating DBMS performance and isolation under `Cgroupv2`, a newer version of `Cgroup` with new I/O controllers, new unified groups, and new implementation that overcome many of `Cgroup` implemetation problems [13]. One of the most important difference between the new and the old `Cgroup`, is the support of buffered write along side non-buffered disk write when applying I/O control [12]. This new feature supposes to make the I/O control stronger and may affects the previous results. Regardless of a newer developed linux kernel, KVM, and LXC software version, it was important to compare DBMS performance isolation under these new circumstances. Finally, considering the latency metric in our experiments for a better explanation and details of the underlying problem.

The key finding of our work is that VMs outperform containers in both I/O performance and isolation in DBMS. This finding is contrary to the general belief. The container is widely believed to beat VM in performance because container does not suffer from virtualization overheads. However, according to our results, KVM outperforms LXC with up to 94% in MySQL throughput. Furthermore, LXC fails to achieve isolation; a container given 30% share of disk bandwidth consumes 70% share although the resource control mechanism, *cgroupv2*, enforces disk I/O limits.

Our contributions consist of analyzing DBMS performance and performance isolation in the container. The sharing of a journaling module among containers degrades I/O performance in DBMS. To guarantee the consistency, existing file systems typically use journaling with transactions. A journaling module batches updates from multiple containers into a transaction and commits the transaction to disk periodically or when `fsync` is invoked. DBMS application invokes a lot of `fsync()` to ensure consistency and to write all updates to disk. If a single transaction contains updates from multiple containers, each container has to wait until the data belonging to other containers is flushed. Journaling also interferes with performance isolation among containers. When an update-intensive container commits a transaction, it adversely affects the performance of other containers. An update from the container is likely to be batched together with updates from other containers. Even if each transaction contains updates solely from one container, the transactions are serialized in the journaling module and cannot be committed in parallel. In addition, the journaling interferes with disk I/O control of `Cgroup`. Since a journaling module is running outside of containers, I/O requests from the module are not accounted for containers that initiate the commits.

The rest of this paper is organized as follows. Section 2 describes the background of virtual machine and container architectures and the overview of disk I/O control in them. Section 3 highlights the motivation by showing that containers are not suitable for DBMS consolidation despite outperform VM in I/O throughput. The DBMS performance and its isolation are analyzed in Sections 4. Section 5 discusses our findings, while Section 6 presents the work related to ours. Finally Section 7 concludes the paper.

2. Background

Hypervisors and containers have different architectures and properties. Section 2.1 overviews the internal design of hypervisor-based and container-based virtualization using KVM and LXC, with highlighting the I/O performance and performance isolation. Section 2.2 presents the disk I/O control by `cgroup` in KVM and LXC, and analyzes the drawbacks with it.

2.1 Virtual machine and Container

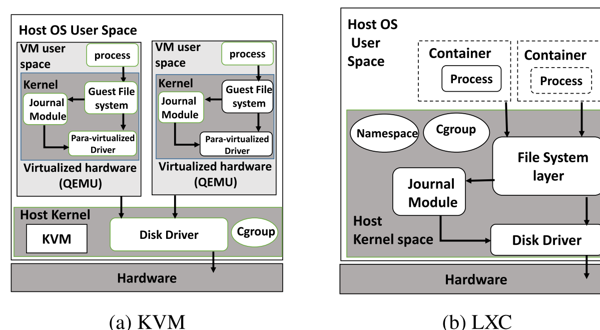
In hypervisor-based virtualization, a hypervisor runs on top of a physical machine and virtualizes the underlying hardware resources onto which a guest OS is installed, resulting in multiple virtual machines running on the same physical machine. On the other hand, containers shift the layer of virtualization from the hardware level to the OS level. Containers rely on mechanisms provided by the host OS; namespaces to separate containers logically and `Cgroup` to control resources among containers.

Figure 1 (a) and (b) show the architecture of KVM and LXC, respectively. In KVM, a guest OS runs on top of a host OS. Hardware devices such as disk drives are virtualized with QEMU device emulator [14]. A disk I/O request is handled in a guest OS, passed to a device emulator, and then processed by a host OS. In LXC, user processes run directly on a host OS without any virtualization overheads. A disk I/O request from a container is handled in the same way as ordinary processes on the host.

In exchange for the performance advantage, LXC provides weaker isolation of performance. Since containers share buffer caches and other data structures at the OS-level, the activity inside one container is likely to affect the performance of other containers through the shared kernel components. Even though the resource controlling mechanism *Cgroupv2* enforces resource limits on containers, it can be bypassed as it will be shown in Section 4.1.

2.2 Disk I/O Control by Cgroup

Both of LXC and KVM rely on `Cgroup` to enforce disk I/O control. `Cgroup` supports two policies for controlling disk I/O: 1) I/O throttling and 2) proportional-weight. *I/O throttling* caps the maximum usage of I/O bandwidth or I/O request rates. In I/O throttling, a container/VM cannot make use of an idle resource



(a) KVM (b) LXC

Fig. 1 Architectures and I/O pathes of KVM virtual machine and LXC container.

© 2020 Information Processing Society of Japan

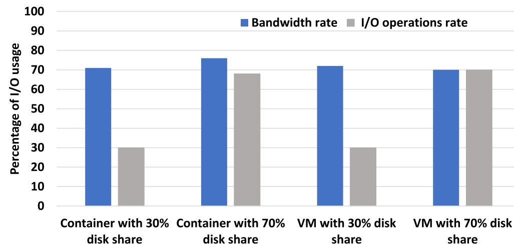


Fig. 2 Proportional-weight policy in `cgroup` fails to control disk I/O bandwidth. High-priority VM/container (I/O request rate is 70%) gets less bandwidth than a low-priority VM/container (I/O request rate is 30%).

even if there is no contention over the resource; it wastes the idle resource. *Proportional-weight* enforces the resource limits only when there is contention over the resource. For example, a container given 30% share of disk time can consume as much I/O bandwidth as possible if there is no contention over disk resource. More precisely, the `Cgroup` controls "disk time". This leads to the control over disk I/O bandwidth and thus we use the term "I/O bandwidth". Needless to say, if there is contention over disk I/O bandwidth, the container can use up to 30% share of disk I/O bandwidth.

Similarly to old `Cgroup`, the new version of control group `Cgroupv2` uses disk time to apply the proportional-weights I/O control. These time-based weight controls the I/O requests but do not consider the size of I/O each request takes. In the case, the amount of disk I/O that each I/O request incurs is not taken into consideration when applying the desired share. If a container/VM issues a single I/O request that incurs the huge amount of disk I/O, it can monopolize the disk bandwidth. This problem is amplified in DBMS in which *direct* I/O is employed. Direct I/O has been introduced for DBMS. It allows DBMS to manage file cache by itself because direct I/O bypasses the buffer cache inside the OS.

We perform an experiment consisting of running two collocated containers/VMs and we used Cgroup proportional-weights policy for disk I/O control. The experimental environment is described in Section 3.1. One container/VM is given given 30% share of disk I/O and the other container/VM is given 70% share of disk I/O. Both of these two container/VM are performing sequential writes using FIO workload but with different I/O sizes. Here, the low-priority container/VM (given 30% share) issues 64 KB-disk I/Os while the high-priority container/VM (given 70% share) issues 16 KB-disk I/Os. **Figure 2** shows a container/VM with a low disk I/O priority (given 30% share of I/O request rate) can consume more bandwidth than a container/VM with a high disk I/O priority (given 70% share of I/O request rate). The graph shows that despite of high-priority container/VM is given 70% share of disk I/O request rates, the low-priority container/VM consumes more disk bandwidth than the high-priority container/VM.

3. Motivation

This section examines DBMS performance and the performance isolation in containers and reveals that despite showing better performance than KVM in disk I/O benchmarks, LXC is worse in performance and isolation in DBMS benchmarks. Section 3.1 describes the experimental setup and the testbed. Section 3.2 shows the benchmark results of the disk I/O performance and isolation in KVM and LXC. Section 3.3 shows the DBMS performance and the isolation in LXC and KVM.

3.1 Experimental Environment

The testbed consists of dell powerEdge T610 with Xeon 2.8 GHz CPU, 4 cores and 32 GB RAM as a host machine. Ubuntu 18.04.1 LTS 64 bit Linux distribution with 4.18.0-25-generic kernel is installed. SAS hard disk of 1 TB is formatted with ext4 file system with the default journaling mode; i.e., only the metadata are journaled. Disk resource control is enforced through the kernel new version of control group "`Cgroupv2`" which is lately supported by LXC and can be with KVM. LXC 3.0.4 is used for Linux containers. KVM-qemu 2.11.1 is installed on the host and the guest environments are exactly the same as the host. Each VM is allocated one virtual CPU that pins to a one CPU core and 1 GB RAM with a raw disk partition allocated as secondary storage. The same configuration is applied to LXC containers.

To examine the performance and isolation in DBMS, MySQL ver. 5.7.27 is installed in each container/VM with InnoDB as a storage engine. MySQL is configured to use direct I/O since it is the common setting in DBMS to avoid the well-known problem of double caching. The transaction model is the default `autocommit`, in which MySQL performs a commit after each SQL statement. Sysbench OLTP benchmark [15] generates workloads, which runs in a separated machine connected via Cisco 1 Gbit Ethernet switch. Sysbench is configured to use the non-transactional mode so that each query is automatically committed. The workload generates INSERT queries to 10 database tables each with 100,000 rows of records. The number of clients is increased until I/O operations are saturated.

3.2 Disk I/O Performance and Isolation

Container are supposed to outperform VMs since KVM causes additional overhead due to an extra layer of virtualization. Past studies [14], [16], [17], [18], [19], [20], [21] have shown that traditional hypervisors such as Xen [22], VMware [23], and KVM [4] have high overheads. From the viewpoint of performance isolation, VM promises to provide better isolation because each VM runs a stand-alone OS without sharing kernel data structures.

To confirm this general belief, we compare I/O performance and performance isolation of LXC and KVM. **Figure 3** shows the I/O bandwidth of KVM and LXC. In this experiment, one instance of a container or a VM is running Flexible I/O (FIO) benchmark [24] is used to generate four types of I/O workloads: 16 KB random read/write and 64 KB sequential read/write. Direct I/O mode is turned on. As we can see from Fig. 3, LXC outperforms KVM in all types of I/O workloads.

LXC shows better performance than KVM in consolidation case, when two VMs/containers are consolidated on a single physical machine. The one VM/container is given 30% share of I/O requests and the other is given 70% share. **Figure 4** shows the throughput of one container/VM (the one with 70%

© 2020 Information Processing Society of Japan

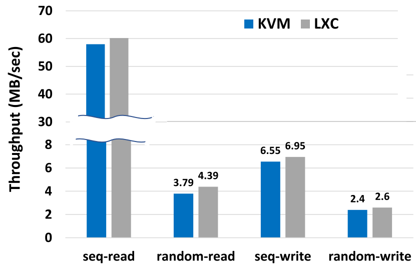
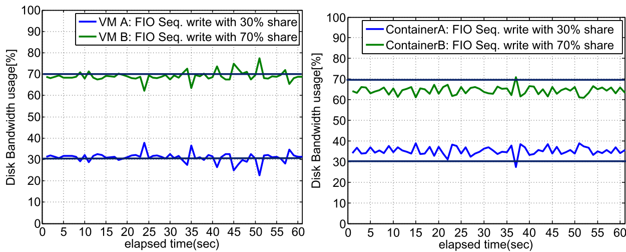


Fig. 3 Disk I/O throughput in KVM and LXC. LXC beats KVM in all workloads.

Fig. 4 Disk I/O throughput in KVM and LXC in consolidation case when two I/O workloads are collocated togather. LXC outperform KVM in all workloads.



(a) Disk I/O bandwidth in KVM (b) Disk I/O bandwidth in LXC

Fig. 5 Performance isolation in KVM and LXC where the one VM/container is given 30% share of disk I/O and the other is given 70%. Both KVM and LXC respect the given shares of disk I/O.

disk share) when the same I/O workloads in Fig. 3 are run in two VMs/containers. As this figure shows LXC beats KVM in all cases.

Figure 5 (a) and (b) show the performance isolation of KVM and LXC, respectively. The one VM/container is given 30% share of I/O requests and the other is given 70% share. The workload used is the sequential write from FIO benchmark. Figure 5 shows both KVM and LXC respect the shares of I/O bandwidth gracefully. The VM/container with 30% share consumes around 30% share, and the VM/container with 70% share consumes around 70% share. LXC enforces the resource limit successfully and shows comparable performance isolation to VM.

3.3 DBMS Performance and Isolation

From the results obtained in Section 3.2, it is expected that LXC is more appropriate than KVM for consolidating DBMSes. LXC outperforms KVM in disk I/O throughput and shows isolation comparable with KVM.

To confirm these expectations, we compared the MySQL performance and isolation in LXC and KVM. **Figure 6** shows MySQL throughput in KVM and LXC. In this experiment, two VMs/containers are launched. The one VM/container executes MySQL and the other executes the sequential write workload of the FIO benchmark. The VM/container running MySQL is given 30% share of disk I/O and the other is given 70% share.

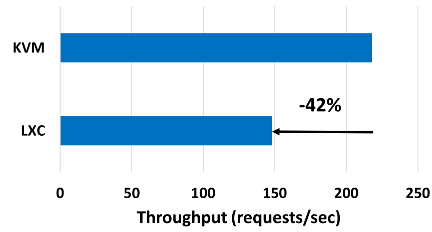
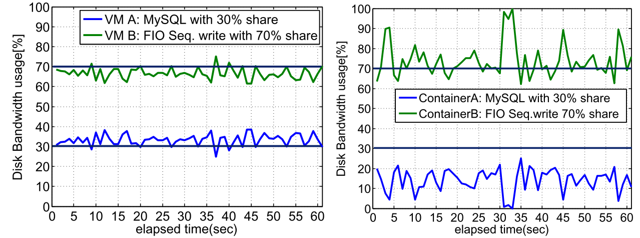


Fig. 6 MySQL throughput in KVM and LXC. MySQL is consolidated with the FIO sequential-write workload. KVM outperforms LXC.



(a) Disk I/O bandwidth in KVM (b) Disk I/O bandwidth in LXC

Fig. 7 Performance Isolation in LXC and KVM. MySQL is consolidated with the FIO sequential-write workload.

As seen from Fig. 6, KVM beats LXC in MySQL throughput. Container performance is 42% worse than VM performance. **Figure 7** compares the performance isolation between VMs and containers in the MySQL workload. The X-axis shows the elapsed time and the Y-axis shows the percentage of the disk I/O bandwidth consumed by each VM/container. Figure 7 (a) indicates that KVM respects the resource limit because the VM with 30% share consumes 30–38% of disk bandwidth and the other VM with 70% share does 62–70%. On the other hand, Fig. 7 (b) indicates that LXC violates the resource limit; the bandwidth consumption of the container with 30% share fluctuates and ranges from 5% to 20%. The other container with 70% share consumes 70–95% bandwidth.

4. Analyzing Performance and Isolation Anomaly in DBMS

To understand the performance results of DBMS that are contrary to our expectations, this section investigates the performance and the performance isolation of DBMS. Section 4.1 suggests `fsync` calls issued at a high rate can slow down disk I/O in container and that really matters in MySQL performance. In Section 4.2, the analysis of how a journaling module interferes with the DBMS performance isolation in containers is presented.

4.1 Impact of `fsync` on MySQL Performance

Although LXC outperforms KVM in disk I/O performance as shown in Section 3.2, MySQL performance is better in KVM than LXC. A major difference in the MySQL and FIO workloads is that the MySQL issues `fsync` frequently to ensure that all updates are written to the final destination on disk. In the MySQL workload, `fsync` is issued at the rate of 27 times/sec in KVM and 14 times/sec in LXC. On the other hand, the FIO workload

© 2020 Information Processing Society of Japan

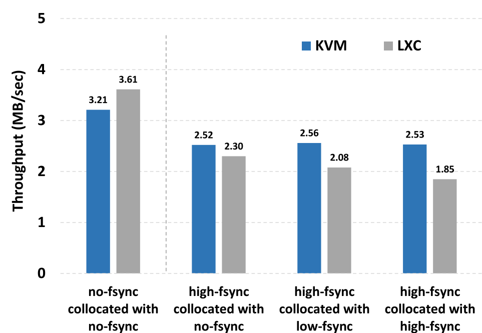


Fig. 8 Throughput of disk I/O in KVM and LXC. A container/VM is running high-fsync workload and is collocated with either 1) no-, 2) low-, or 3) high-fsync workload. LXC performance degrades as fsync intensity increases in the collocated workload.

does not issue `fsync` explicitly. In this case, updates are flushed to disk at the rate of 0.2 times/sec (every 5 seconds).

To confirm that the high rate of `fsync` has significant impact on I/O performance, we have prepared three I/O workloads: 1) no-, 2) low-, and 3) high-fsync workload. The no-fsync workload is the same as the FIO sequential-write workload. The low- and high-fsync benchmarks are based on the no-fsync workload but set to issue `fsync` calls more frequently. The low- and high-fsync workload issue `fsync` every 20 I/O operations (at the rate of 3–5 times/sec) and 5 I/O operations (at the rate of 10–15 times/sec), respectively.

Figure 8 shows the throughput of the high-fsync workload when it is collocated with a VM/container running either 1) no-, 2) low-, or 3) high-fsync workload. For comparison, the figure shows the throughput of the no-fsync collocated with the no-fsync workload. LXC outperforms KVM only when both containers/VMs are running the no-fsync workload. If one workload is changed to the high-fsync, KVM always beats LXC. The throughput in LXC degrades when the collocated workload issues `fsync` more frequently. While KVM shows a stable and constant throughput (around 2.5 MB/sec) regardless of collocated workload. This result suggests that a high rate of `fsync` calls is relevant in disk I/O performance in LXC but not in KVM.

The `fsync` is closely related to file system journaling. A standard Linux file system, Ext4, uses journaling to guarantee the file system consistency. Containers share the same journaling module of the host OS. A journaling module batches updates from multiple containers into a transaction and commits the transaction periodically or when `fsync` is invoked. If a single transaction contains updates from multiple containers, each container has to wait until the updates coming from other containers to be flushed to disk. Ideally, calling `fsync` in a container should flush only the dirty data belonging to that particular container. Unfortunately, calling `fsync` causes unrelated data to be flushed as well. Even if a single transaction contains updates solely from one container, transactions are serialized and cannot be committed in parallel because two different transactions may update a global shared data structure on disk (for instance, inode bitmap).

To verify the above observation, we measure the latency of `fsync` in KVM and LXC in the previous experiment. **Table 1** shows the fsync latency of the high-fsync workload when it collocated with either 1) no-, 2) low-, or 3) high-fsync workload.

Table 1 Average fsync latency of the high-fsync workload when collocated with either 1) no-, 2) low-, or 3) high-fsync workload in KVM and LXC.

Collocated workload	KVM	LXC
No-fsync	18.58 ms	33.81 ms
Low-fsync	18.94 ms	38.86 ms
High-fsync	18.27 ms	44.09 ms

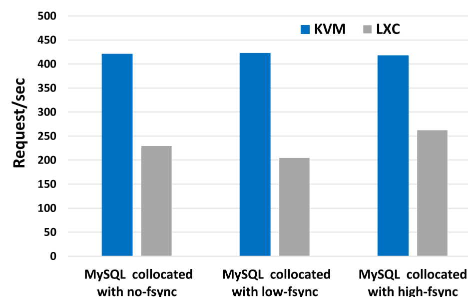


Fig. 9 MySQL throughput in LXC and KVM with either 1) no-, 2) low-, or 3) high-fsync workloads. MySQL is given 70% share of disk I/O.

Table 2 Average fsync latency of MySQL when collocated with no-, low-, high-fsync workload. MySQL VM/container is given 70% share.

Collocated workload	KVM	LXC
No-fsync	26.87 ms	72.51 ms
Low-fsync	27.28 ms	85.09 ms
High-fsync	26.12 ms	74.08 ms

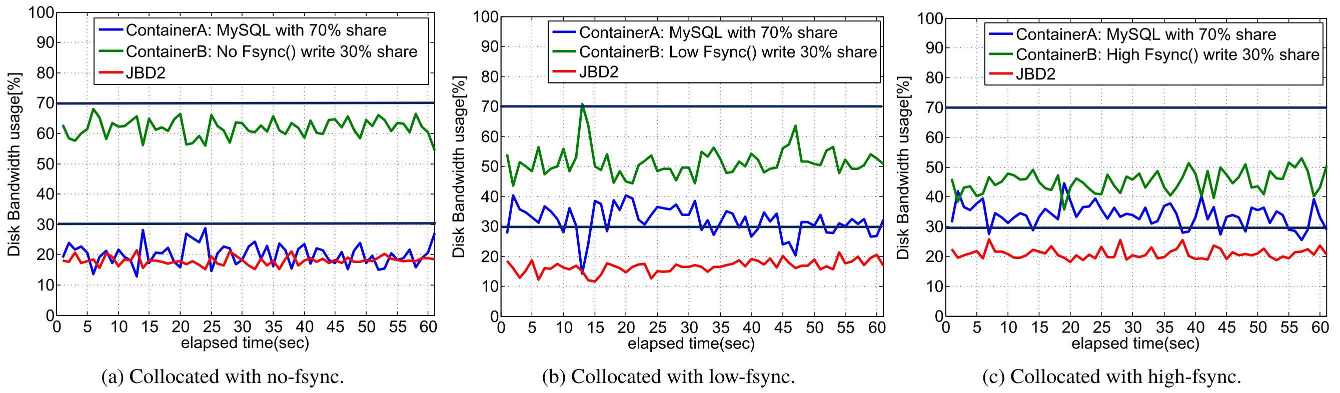
For KVM, the average latency is about 18.5 ms regardless of the collocated workloads. While the fsync latency increases in LXC as fsync intensity increases in the collocated workload. This increase of fsync latency degrades the I/O performance of the high-fsync workload in LXC.

Figure 9 confirms that MySQL performance is largely affected by `fsync` calls. The throughput of MySQL is measured when container/VM is collocated with either 1) no-, 2) low-, or 3) high-fsync workload. A container/VM running MySQL is given 70% share of disk I/O while the other is given 30% share. Since each VM has its standalone OS with its own journaling module, it gets rid of all the journal-related problems. KVM always shows better performance than LXC. KVM outperfoms LXC up to 95%. The MySQL throughput in KVM is almost constant (around 420 requests/sec) in all the cases. On the other hand, the MySQL throughput in LXC degrades from 230 to 209 req/sec when the collocated workload is changed from no-fsync to low-fsync. Since the low-fsync workload issues `fsync` more frequently than no-fsync, the `fsync` latency increases in MySQL.

Table 2 shows the `fsync` latency of MySQL increases from 72.51 ms to 85.09 ms in LXC when the collocated workload is changed from no-fsync to low-fsync.

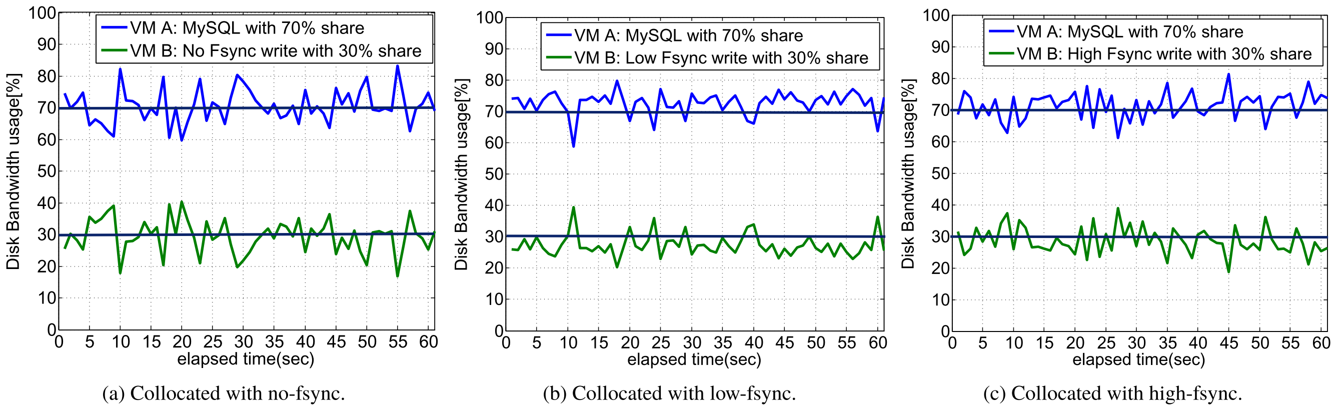
The same behavior is observed in **Fig. 10** which shows MySQL throughput when VM/container is given 30% instead of 70% share. LXC outperforms KVM in the no-fsync and low-fsync workloads although it is beaten in the high-fsync workload. This improvement in MySQL performance happens for two reasons. First, the performance isolation is violated in LXC and thus MySQL is allocated more disk I/Os than its given share. The details are discussed in Section 4.2. Second, as the collocated workload issues more `fsync`, it causes more journaling updates. Thus, some of MySQL updates are flushed as well due to sharing

© 2020 Information Processing Society of Japan



(a) Collocated with no-fsync. (b) Collocated with low-fsync. (c) Collocated with high-fsync.

Fig. 11 Disk I/O usage in MySQL in LXC. Collocated with no-, low-, high-fsync workloads. MySQL is given 70% share.



(a) Collocated with no-fsync. (b) Collocated with low-fsync. (c) Collocated with high-fsync.

Fig. 12 Disk I/O usage in MySQL in KVM. Collocated with no-, low-, high-fsync workloads. MySQL is given 70% share.

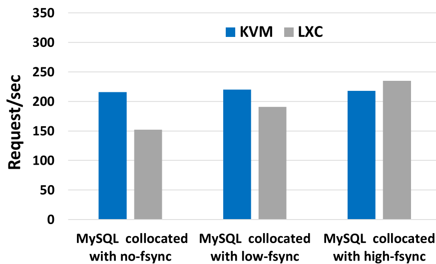


Fig. 10 MySQL throughput in LXC and KVM with either 1) no-, 2) low-, or 3) high-fsync workloads. MySQL is given 30% share of disk I/O.

of the same journaling transaction among containers.

4.2 Journaling Influence on MySQL Isolation

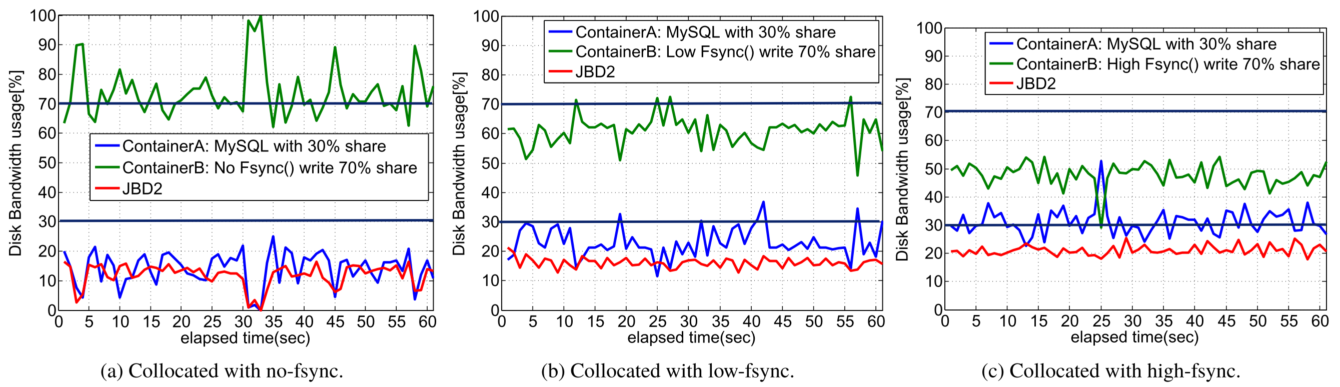
MySQL performance isolation in containers is terrible as shown in Fig. 7. The sharing of a journaling module in containers has a negative impact on disk I/O isolation. The journaling module causes performance dependencies across collocated containers. This happens as updates from multiple containers are bundled in a single journaling transaction. Hence, one container can affect another because each container has to wait until the data belonging to other containers are flushed. Even if a transaction consists solely of updates from one container, the transactions are serialized in the journaling module and cannot be committed in parallel. Thus, `fsync` calls from other containers are suspended causing a longer blocking time of further I/O operations. In this case, the disk isolation mechanism of `Cgroup` judges the container as not issuing I/O operations and hands its disk share to other containers. Also, I/O operations from the journaling module are overlooked by `Cgroup` and not accounted for containers that initiate them as the journaling module is running outside of controlled containers.

To confirm that isolation in containers is violated due to the journaling, the disk I/O usage of MySQL is compared when it is collocated with no-, low- and high-fsync workload. **Figures 11** shows the disk I/O usage of MySQL container when it is collocated with no-, low-, high-fsync workload. **Figure 12** shows that of MySQL VM.

MySQL is given 70% disk share while the other collocated VM/container is given 30%. Since MySQL is update-intensive, the I/O usage of the journaling is around 18% in LXC even when it is collocated with no-fsync workload. But `cgroup` overlooks these journaling I/Os and judges the MySQL container as not I/O-intensive. As a result, `cgroup` allocates more disk I/O to the collocating container while the MySQL container is given around 20% disk share instead of 70% share.

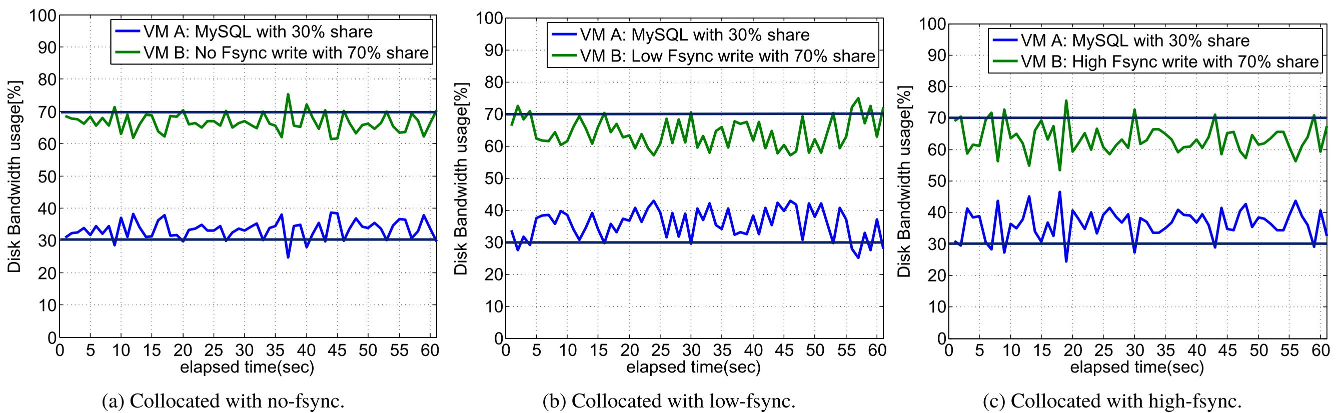
When the collocated workload is changed to the low- or high-fsync workload, the MySQL container consumes more share of the disk I/O. This is because the low- or high-fsync container issues more `fsync` calls which compete with `fsync` from the MySQL container and are suspended in the journaling module due to the contention. Hence, the I/Os from low- or high-fsync container are reduced and `cgroup` allocates more disk I/Os to the MySQL container. This results in the improved performance of

© 2020 Information Processing Society of Japan



(a) Collocated with no-fsync. (b) Collocated with low-fsync. (c) Collocated with high-fsync.

Fig. 13 Disk I/O usage in MySQL in LXC. Collocated with no-, low-, high-fsync workloads. MySQL is given 30% share.



(a) Collocated with no-fsync. (b) Collocated with low-fsync. (c) Collocated with high-fsync.

Fig. 14 Disk I/O usage in MySQL in KVM. Collocated with no-, low-, high-fsync workloads. MySQL is given 30% share.

MySQL container in Fig. 9.

Figures 13 and **14** shows the percentages of disk I/O in LXC and KVM where MySQL is given 30% share. As shown in the figure, the MySQL container is given less than 30% share when collocated with the no-fsync, but it is given around 30% share when collocated with the high-fsync. When fsync-intensive container is collocated with MySQL, it slows down due to severe contention over the journaling with MySQL and hands over its disk I/O share to MySQL. This explains the performance mystery in Fig. 10 when the performance of MySQL container improves if it is collocated with more fsync-intensive workloads. But this improvement happens with sacrificing the performance isolation. KVM gracefully respects the I/O control of `cgroup` and divides the disk I/O into 30% and 70% share between VMs.

5. Discussion

There is a trade-off between performance and isolation in the VM and the container. Our experimental results suggests the use of VM is preferable in update-intensive application like DBMS at least with current implementation of LXC. The file-system journaling, which is mandatory to guarantee the consistency of file-system, is the root cause of the poor I/O performance and its isolation in containers. Since the journaling module is shared among containers, it becomes a bottleneck in performance and interferes with disk I/O control of `Cgroup` in containers. Even with the new `Cgroupv2` disk I/O control, journaling I/O is still being overlooked. Disabling the file system journaling to overcome journaling-related problems in containers is unpractical and not acceptable in DBMS because it risks loosing user data and file system consistency. The kernel `Cgroup` should provide a higher isolation for disk I/O. Journaling I/O should be taken into consideration when applying the disk I/O control among containers. It may be possible to estimate the amount of journaling I/Os of each container by observing non-journaling I/O behaviors of each container. This estimate could be used to adjust the weight of disk I/Os for each container by the `Cgroup`. Another possible solution is to present each container with its own block device to put it's container file system in it so that each container can have its own journaling module. This will avoid the journaling related problems like the KVM does.

6. Related Work

The performance of OS- and hypervisor-based virtualization is compared in Refs. [25], [26], [27], [28], [29]. According to them, the container and VM achieve near-native performance in CPU-intensive workloads, but KVM incurs the overheads larger than Xen or containers in I/O-intensive workloads. Sharma et al. [30] and Zhang et al. [31] show that VMs are not as scalable and resource-efficient as the container. Surya et al. [32] compare the performance of KVM and the docker container in E-commerce applications. They show that the container has a better CPU and disk utilization than VM, but suffers from contention in memory and network I/O. All of these previous works focus only on the performance; the performance isolation is not addressed.

© 2020 Information Processing Society of Japan

Performance isolation is discussed in Xavier et al. [33]. It compares Xen with Linux VServer [34], OpenVZ [35] and LXC containers in the context of high performance computing. Regarding the performance, all the containers show near-native performance. Regarding the performance isolation, VServer and OpenVZ show better memory isolation than LXC. For disk I/O, Xen shows better isolation of the performance than any of the container implementations but the underlying causes are not investigated. Matthews et al. [36] study performance isolation among VMware, Xen, Solaris containers and OpenVZ. VMware imposes stronger isolation than Xen and the containers. Both of OpenVZ and Solaris containers show good isolation in CPU performance while suffering from poor isolation in memory and I/O benchmarks. However, the underlying mechanism that causes performance interference in containers is not investigated.

Soltesz et al. [37] compare performance isolation of VServer with Xen, using database applications as target. A VServer container running a database application severely suffers from other containers running I/O-intensive workloads. Their analysis shows that I/O-intensive containers monopolize the buffer cache to degrade the database applications. Our investigation reveals that the file-system journaling disturbs I/O operations in the container even if the buffer cache is not shared. In our experiments, the direct I/O is used to bypass the buffer cache. Mizusawa et al. [38] present an evaluation of file operations of OverlayFS which is a widely recognized method for improving I/O performance in docker. According to their results, the performance of file writing is severely low because of the synchronization of data in the memory and the storage. They suggest disabling this synchronization to improve the I/O performance which is not acceptable for an application like DBMS. Xavier et al. [39] compare LXC and KVM in terms of the performance interference in I/O-intensive workloads. According to their study, LXC suffers more severely from interference than KVM when a database is collocated with I/O-intensive workloads. However, no analysis is conducted to understand how performance interference occurs in LXC. Our investigation shows that KVM beats LXC not only in isolation but also in database performance as we consider the case of journaling-intensive workloads.

Some works propose new file systems such as IceFS [40] and SpanFS [41] to provide logically separated units for independent journaling among containers. MultiLanes [42] provides a virtualized storage device for each container on top of which an isolated I/O stack is built. These novel file systems can overcome the problems of the shared journaling among containers, but the existing file systems or I/O stacks must be replaced to utilize them. Park et al. [43] propose a new journaling technique called *iJournaling*, which limits the journaling updates on the metadata of fsynched file. These techniques improve the performance of update-intensive containers but do not overcome all the journaling-related problems. For example, fsync call serialization and uncounted journaling I/Os are not addressed.

7. Conclusion

This paper investigates the DBMS performance and performance isolation in KVM and LXC. Our key finding is that KVM outperforms LXC in both I/O performance and its isolation in DBMS. This finding is contrary to the general belief that the container is better than the VM in performance because of no virtualization overheads. Furthermore, LXC fails to achieve the performance isolation among containers although a resource-control mechanism called *cgroup* enforces disk I/O limits.

Our analysis shows that the file-system journaling, which is mandatory to guarantee the consistency of file-system especially in database applications, is the root cause of the poor I/O performance and its isolation in LXC. Since the journaling module is shared among containers, it becomes a bottleneck in performance and causes a serious problem on update-intensive applications such as DBMS. The paper identifies journaling-related problems that cause performance dependencies among containers and the violation of performance isolation. Journaling interferes with disk I/O control of cgroup in containers. Since the journaling module is running outside of the container, I/O operations from the module are not accounted for containers that initiate them.

In contrast, KVM avoids the journal-related problems because each VM has its own journaling module due to the complete separation of kernel components. We conclude that VM is a better choice than the container for DBMS consolidation.

Acknowledgments This work is partially supported by JST, CREST, JPMJCR19F3 and Keio Gijuku Academic Development Funds.

References

[1] Verghese, B., Gupta, A. and Rosenblum, M.: Performance isolation: sharing and isolation in shared-memory multiprocessors, *ACM SIGPLAN Notices*, Vol.33, No.11, pp.181–192, ACM (1998).

[2] Gupta, D., Cherkasova, L., Gardner, R. and Vahdat, A.: Enforcing performance isolation across virtual machines in Xen, *Middleware 2006*, pp.342–362, Springer (2006).

[3] Linux container (LXC): Infrastructure for container projects, A WEB page, available from ⟨https://linuxcontainers.org/⟩ (accessed 2019-10-07).

[4] Kivity, A., Kamay, Y., Laor, D., Lublin, U. and Liguori, A.: KVM: The Linux virtual machine monitor, *Proc. Linux Symposium*, Vol.1, pp.225–230 (2007).

[5] SQLite: Well-Known Users of SQLite, A WEB page, available from ⟨https://www.sqlite.org/famous.html⟩ (accessed 2019-10-07).

[6] Facebook: The Facebook Data Center, A WEB page, available from ⟨http://www.datacenterknowledge.com/the-facebook-data-center-faq-page-2/⟩ (accessed 2019-10-07).

[7] Salesforce: The salesforce.com Multitenant Architecture, A WEB page, available from ⟨https://developer.salesforce.com/page/Multi-Tenant-Architecture⟩ (accessed 2019-10-07).

[8] AzureSQL: Microsoft Azure SQL Databaser, A WEB page, available from ⟨https://azure.microsoft.com/en-us/services/sql-database⟩ (accessed 2019-10-07).

[9] CloudSQL: Google Cloud SQL, A WEB page, available from ⟨https://cloud.google.com/sql/⟩ (accessed 2019-10-07).

[10] Abdulrazak, A.M.A. and Kenji, K.: Containers or Hypervisors: Which Is Better for Database Consolidation?, *Proc. International Conference on Cloud Computing Technology and Science* (*CloudCom*), pp.564–571, IEEE (2016).

[11] Menage, P.: Linux Cgroup resource managment, A WEB page, available from ⟨https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt⟩ (accessed 2020-10-01).

[12] Cgroup-v2: New version of Cgroup, A WEB page, available from ⟨https://www.kernel.org/doc/Documentation/-cgroup-v2.txt⟩ (accessed 2020-10-01).

[13] CGROUP: Linux Programmer's Manual, A WEB page, http://man7.org/linux/man-pages/man7/cgroups.7.html (accessed 2020-10-01).

[14] Zhang, B., Wang, X., Lai, R., Yang, L., Wang, Z., Luo, Y. and Li, X.: Evaluating and optimizing I/O virtualization in kernel-based virtual machine (KVM), *Network and Parallel Computing*, Lecture Notes in Computer Science, Vol.6289, pp.220–231, Springer (2010).

© 2020 Information Processing Society of Japan

[15] SQLite: Sysbench benchmark suite, A WEB page, available from ⟨https://github.com/akopytov/sysbench⟩ (accessed 2019-10-07).

[16] Menon, A., Santos, J.R., Turner, Y., Janakiraman, G. and Zwaenepoel, W.: Diagnosing Performance Overheads in the Xen VirtualMachine Environment, *Proc. 1st ACM/USENIX International Conference on Virtual Execution Environments* (*VEE*), pp.13–23, ACM (2005).

[17] Caglar, F., Shekhar, S. and Gokhale, A.: Towards a Performance Interferenceaware Virtual Machine Placement Strategy for Supporting Soft Realtime Applications in the Cloud, *Proc. 3rd International Workshop on Real-time and Distributed Computing in Emerging Applications* (*REACTION*), pp.15–20, Universidad Carlos III de Madrid (2014).

[18] Jinho, H., Sai, Z., F.Y., W. and Timothy, W.: A component-based performance comparison of four hypervisors, *Proc. IFIP/IEEE International Symposium on Integrated Network Management* (*IM*), pp.269–276, IEEE (2013).

[19] Rasmusson, L. and Diarmuid, C.: Performance overhead of KVM on Linux 3.9 on ARM Cortex-A15, *ACM SIGBED Review*, Vol.11, No.2, pp.32–38 (2014).

[20] Xing, P., Ling, L., Yiduo, M., Sankaran, S., Koh, Y. and Calton, P.: Understanding performance interference of i/o workload in virtualized cloud environments, *Proc. 3rd International Conference on Cloud Computing* (*CLOUD*), pp.51–58, IEEE (2010).

[21] Malensek, M., Pallickara, S.L. and Pallickara, S.: Alleviation of Disk I/O Contention in Virtualized Settings for Data-Intensive Computing, *BDC*, pp.1–10, IEEE Computer Society (2015).

[22] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and the art of virtualization, *ACM SIGOPS Operating Systems Review*, Vol.37, No.5, pp.164–177 (2003).

[23] Walters, B.: VMware virtual platform, *Linux journal*, Vol.1999, No.63es, p.6 (1999).

[24] FIO: The Flexible I/O (FIO) benchmark, A WEB page, available from ⟨https://github.com/axboe/fio⟩ (accessed 2019-10-07).

[25] Felter, W., Ferreira, A., Rajamony, R. and Rubio, J.: An Updated Performance Comparison of Virtual Machines and Linux Containers, Technical Report RC25482 (AUS1407-001), IBM Research Division (2014).

[26] Regola, N. and Ducom, J.: Recommendations for Virtualization Technologies in High Performance Computing, *Proc. International Conference on Cloud Computing Technology and Science* (*CloudCom*), pp.409–416, IEEE (2010).

[27] Che, J., ans C. Shi, Y.Y. and Lin, W.: A synthetical performance evaluation of openVZ, Xen and KVM, *Proc. Asia-Pacific Services computing Conference*, pp.587–594, IEEE (2010).

[28] Morabito, R., Kjallman, J. and Komu, M.: Hypervisors vs. Lightweight Virtualization: A Performance Comparison, *Proc. International Conference on Cloud Engineering* (*IC2E*), pp.368–374, IEEE (2015).

[29] Raho, M., Spyridakis, A., Paolino, M. and Raho, D.: KVM, Xen and Docker: A performance analysis for ARM based NFV and Cloud computing, *Proc. 3rd Workshop on Advances in Information, Electronic and Electrical Engineering* (*AIEEE*), pp.1–8, IEEE (2015).

[30] Sharma, P., Chaufournier, L., Shenoy, P. and Tay, Y.C.: Containers and Virtual Machines at Scale: A Comparative Study, *Proc. 17th International Middleware Conference*, pp.1:1–1:13, ACM (2016).

[31] Zhang, Q., Liu, L., Pu, C., Dou, Q., Wu, L. and Zhou, W.: A Comparative Study of Containers and Virtual Machines in Big Data Environment, *2018 IEEE 11th International Conference on Cloud Computing* (*CLOUD*), pp.178–185 (2018).

[32] Garg, S.K., Lakshmi, J. and Johny, J.: Migrating VM Workloads to Containers: Issues and Challenges, *11th IEEE International Conference on Cloud Computing, CLOUD 2018*, pp.778–785 (2018).

[33] Xavier, M., Neves, M.V., Rossi, F.D., Ferreto, T.C., Lange, T. and Rose, C.F.D.: Performance evaluation of container-based virtualization for high performance computing environments, *Proc. 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing* (*PDP*), pp.233–240, IEEE (2013).

[34] Linux-VServer: Welcome to Linux-VServer, A WEB page, available from ⟨http://linux-vserver.org/⟩ (accessed 2019-10-07).

[35] OpenVZ: Open source container-based virtualization for Linux, A WEB page, available from ⟨https://openvz.org/⟩ (accessed 2019-10-07).

[36] Matthews, J., Hu, W., Hapuarachchi, M., Deshane, T. and Hamilton, D.G.: Quantifying the Performance Isolation Properties of Virtualization Systems, *Proc. 2007 Workshop on Experimental Computer Science*, (*ExpCS*), p.6, ACM (2007).

[37] Soltesz, S., Potzl, H., Fiuczynski, M., Bavier, A. and Peterson, L.: Container based operating system virtualization: A scalable, high-performance alternative to hypervisors, *SIGOPS Operating System Review*, Vol.41, No.3, pp.275–287 (2007).

[38] Mizusawa, N., Kon, J., Seki, Y., Tao, J. and Yamaguchi, S.: Performance Improvement of File Operations on OverlayFS for Containers, *Proc. IEEE International Conference on Smart Computing*, pp.297–302, IEEE (2018).

[39] Xavier, M., Neves, M.V., Rossi, F.D., Ferreto, T.C., Lange, T. and Rose, C.F.D.: A Performance Isolation Analysis of Disk-intensive Workloads on Container-based Clouds, *Proc. 23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing* (*PDP*), pp.253–260, IEEE (2015).

[40] Lu, L., Zhang, Y., Do, T., AI-Kiswany, S. and Arpaci-Dusseau, R.: Physical Disentanglement in a Container-Based File System, *Proc. 11th USENIX Symposium on Operating Systems Design and Implementation* (*OSDI*), pp.81–96 (2014).

[41] Kang, J., Zhang, B., Wo, T., Yu, W., Du, L., Ma, S. and Huai, J.: SpanFS: A scalable file system on fast storage devices, *Proc. USENIX Annual Technical Conference* (*ATC*), pp.249–261 (2015).

[42] Kang, J., Zhang, B., Wo, T., Hu, C., and Huai, J.: Multilanes: providing virtualized storage for OS-level virtualization on many cores, *Proc. 12th USENIX Conference on File and Storage Technologies* (*FAST*), pp.317–329 (2014).

[43] Park, D. and Shin, D.: iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call, *2017 USENIX Annual Technical Conference* (*USENIX ATC 17*), pp.787–798, USENIX Association (2017).

Asraa Abdulrazak Ali Mardan received her B.Sc. and M.Sc. in Information Engineering from AL-Narain University, Iraq in 2010 and 2014 respectively. Currently she is a Ph.D. student in Keio University, graduate school of Science and Technology. Her research interests are Cloud Computing, Virtualization technologies, Container, and File systems.

Kenji Kono received his B.Sc. degree in 1993, M.Sc. degree in 1995, and Ph.D. degree in 2000, all in computer science from the University of Tokyo. He is a professor in the Department of Information and Computer Science at Keio University. He received the IPSJ Yamashita-Memorial Award in 2000, IPSJ Annual Best Paper Awards in 1999, 2008, 2009, and 2012, JSSST Software Paper Award in 2014, IBM Faculty Award in 2015, and JSSST Basic Research Award in 2016. He served as a PC member of top conferences such as ICDCS and DSN. He also organized ACM SIGOPS APSys in 2015. His research interests include operating systems, system software, and computer security. He is a member of the IEEE, ACM and USENIX.

© 2020 Information Processing Society of Japan