PAPER

Alleviating File System Journaling Problem in Containers for DBMS Consolidation

Asraa ABDULRAZAK ALI MARDAN[†a)], *Nonmember and* **Kenji KONO**[†], *Member*

SUMMARY Containers offer a lightweight alternative over virtual machines and become a preferable choice for application consolidation in the clouds. However, the sharing of kernel components can violate the I/O performance and isolation in containers. It is widely recognized that file system journaling has terrible performance side effects in containers, especially when consolidating database management systems (DBMSs). The sharing of journaling modules among containers causes performance dependency among them. This dependency violates resource consumption enforced by the resource controller, and degrades I/O performance due to the contention of the journaling module. The operating system developers have been working on novel designs of file systems or new journaling mechanisms to solve the journaling problems. This paper shows that it is possible to overcome journaling problems without re-designing file systems or implementing a new journaling method. A careful configuration of containers in existing file systems can gracefully solve the problems. Our recommended configuration consists of 1) per-container journaling by presenting each container with a virtual block device to have its own journaling module, and 2) accounting journaling I/Os separately for each container. Our experimental results show that our configuration resolves journaling-related problems, improves MySQL performance by 3.4x, and achieves reasonable performance isolation among containers.

key words: *virtualization, container, DBMS, journaling file system*

1. Introduction

Database management systems (DBMSs) are a common service in the cloud. Many cloud-based services such as Dropbox [1], Salesforce [2], and Facebook [3] make use of DBMS. Microsoft's SQL Azure [4] and Google Cloud SQL [5] provide DBMS as a cloud service. Cloud-services providers employ virtualization to consolidate DBMSs for efficient resource utilization and to isolate collocated DBMS workloads [6], [7]. *Containerization* or operating system (OS) based virtualization has become widely used for deploying and consolidating applications in clouds [8]. Containers are lightweight, deploy applications efficiently at a large scale and bring the performance advantage over virtual machines with almost no virtualization overhead.

Despite the advantages, containers have some shortcomings because of the sharing of OS kernel's components such as the file system. It is widely recognized that file system journaling has negative side-effects in containers [9]–[11]. OpenVZ [12] developers show that the journal is a serious bottleneck in containers and try to address this problem [10]. Mizusawa et al. [11] show that the performance of file writing in docker is severely low because of journaling. Our previous work [13], [14] examines DBMS performance and reveals that the sharing of the journaling module among containers has a negative impact on performance of DBMS in containers. On the other hand, journaling is very important to guarantee consistency and for crash recovery in file systems [10], [15]. The Journaling records updates not yet committed to the file system and provides backup and recovery capabilities. Hence, journaling should not be turned off especially in update-intensive applications like DBMS.

The journaling problems in containers are caused mainly for the following reasons. Since a journaling module is shared inherently among containers, it causes performance dependency between containers. A journaling module batches updates from multiple containers into a transaction and commits the transaction to disk periodically or when fsync is invoked. If a single transaction contains updates from multiple containers, each container has to wait until the data belonging to other containers is flushed. Even if each transaction contains updates solely from one container, the transactions are serialized in a journaling module and cannot be committed in parallel. It takes a long time to commit the transaction and fsync from other containers are suspended because of the lack of parallelism. Also, journaling interferes with disk I/O control of the kernel resource control mechanism known as `cgroup` [16]. Since the journaling module is running outside of containers, I/O operations from the module are overlooked by the `cgroup` and not accounted for the container that initiates the journaling I/Os.

To solve these journaling problems, OS researchers work on designing a completely new file systems [9], [17], [18] or developing novel journaling mechanisms [19] to overcome the journaling issues. However, all of these works involve non-negligible modifications to the kernel. Hence, these solutions are difficult to deploy on current cloud platforms.

In this paper, we quest for possible solutions to alleviate the file system journaling problems in containers. We show that these problems can be overcome without redesigning the file systems or modifying the existing kernel. The careful configuration of containers can gracefully solve the journaling-related problems. First, our configuration provides each container with a virtual disk so that each container can have its own file system. Hence, each container has its own journaling module to eliminate the bottle-

Manuscript received September 7, 2020.
Manuscript revised December 23, 2020.
Manuscript publicized April 1, 2021.

[†]The authors are with Keio University, Yokohama-shi, 223–8522 Japan.
a) E-mail: asraaiteng@sslab.ics.keio.ac.jp

DOI: 10.1587/transinf.2020EDP7178

neck of the shared journaling module and its performance dependencies.

Second, these per-container journaling modules are still running outside of controlled containers and their I/Os are still overlooked. Hence, a proper configuration of kernel processes that handle the journaling operations is needed. To account journaling I/Os, these processes should belong to the same `cgroup` to which the container controlled by each kernel process belongs. Our results show that DBMS performance improves up to 1.3x with the virtual block device configuration, and improves more up to 3.4x with the proper configuration of per-container journaling processes. Eventually, containers show performance isolation comparable to that of virtual machines.

This configuration is not widely adopted in the clouds because most container implementations do not allow this configuration. To the best of our knowledge, our recommended configuration is possible only in OpenVZ [12] with Ploop [20]. Most container implementations use `chroot` [21] to provide a container with its own root directory, which is a sub-tree of the host file system [22], [23]. Even if a virtual block device is enabled, it lacks the support of direct I/O that bypasses the kernel cache, and causes the double-caching problem. Unfortunately, major cloud services like Azue [24] and Google cloud [25], neither use our configuration nor support the OpenVZ with Ploop [26], [27]. We advocate the need of per-container virtual disks with its own journaling module, and per-container journaling accounting. These features should be supported in container implementations other than OpenVZ with Ploop.

The rest of this paper is organized as follows: Sect. 2 describes the journaling-related problems in containers and shows that containers are not suitable for DBMS consolidation. Section 3 proposes a configuration model for containers to overcome journaling-related problems. Section 4 shows the experiments' results that confirm the the feasibility of our recommended configuration. Section 5 discusses our findings, while Sect. 6 presents the work related to ours. Section 7 concludes the paper.

2. Journal-Related Problem in Containers

The sharing of kernel components among containers results in violating the isolation among them. This section demonstrates how the sharing of the journaling module violates I/O isolation and degrades the performance of containers. Section 2.1 gives a background on containers. Section 2.2 explains file system journaling in containers while section 2.3 explains the journaling-related problem. Section 2.4 shows that the journaling has negative effects on I/O-intensive applications like DBMS.

2.1 Container-Based Virtualization

Containers refer to multiple virtual units that are created in the user space. In containers, user processes run directly on the host OS without any virtualization overhead. These containers share the same OS kernel but are isolated from each other through private namespaces [28] and the resource control groups, called `cgroup` [16]. Namespaces allow creating separate instances of the global resources. Linux implements filesystem, PID, network, user, IPC, and hostname namespaces. For example, each filesystem namespace has its own root directory and mount table [29].

The `cgroup` is a group of processes that are bound by the same resource limits [30]. The `cgroup` is largely composed of two parts, the core and controllers. The core is responsible for hierarchically organizing processes while the controller is responsible for distributing a specific type of system resource along the hierarchy [16]. The OS kernel provides access to multiple *controllers* (also called subsystems) that limit the resource usage for each container; for example, the I/O controller limits disk usage, the CPU controller limits CPU usage, etc. The I/O controller implements two policies to set limits on input/output access to and from block devices. The first policy is I/O throttling which caps the maximum usage of I/O bandwidth or request rates. The second policy is the proportional I/O weight which assigns a share of disk I/Os [16]. It enforces resource limits only when the resource contention actually occurs.

Containers rely on `chroot` [21] to provide a per-container view of a file system, which is a sub-tree of the host file system [22], [23]. The `chroot` (change root) changes the root directory for the current running process and their children to the sub-tree. A process that is running in such a modified environment cannot access files that are outside the sub-tree. This modified environment is populated with all required configuration files, device nodes, and shared libraries to be run successfully.

2.2 Journaling in Containers

Modern file systems use journaling [10], [15], [19], [31] to keep the file system consistent even after unexpected system crashes or power failures. Updating files or directories usually requires multiple write operations on on-disk data structures. If a power failure or system crash happens between the writes, the on-disk data structures become inconsistent. For example, when a file is removed, the disk blocks it occupies must be returned to the free block list. This operation involves the updates on multiple on-disk data structures such as inode and bitmap. If the on-disk structures are partially updated, the file system becomes inconsistent. In the worst case, the user cannot access to it anymore.

Journaling is write-ahead logging. Before updating the file system, it logs the write operations to an on-disk region called *journal*.

A kernel component responsible for journaling activities is called a *journal module*. Only a single journaling module can run at a time [32]. If there are multiple journaling modules, they cause a race condition and the file system become inconsistent. For efficiency, several updates on files or directories are bundled into a single *transaction* which logs the write operations. In the Linux file system (ext4),

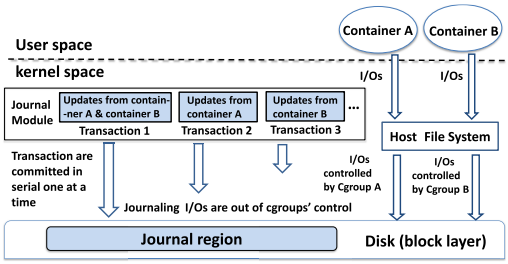


Fig. 1 File system journaling in container virtualization.

JDB2 (Journaling Block Device) is responsible for journaling. The JBD2 groups the updates from multiple containers in that single compound transaction. After logging the updates, the transaction are committed to the file system and then removed from the journal.

In case of a system crash or power failure, the file system recovers from inconsistency by re-doing the logs in the journal. Under the normal operation, transactions are committed periodically (5 sec by default) or every time `fsync` is invoked.

2.3 Journaling Issues in Containers

The sharing of a journaling module in containers has a negative impact on disk I/O performance and isolation. The journaling module causes performance dependencies across collocated containers, which result in violation of performance isolation and degrade I/O performance in containers.

In Fig. 1, two containers share the single journaling module and thus a single transaction bundles updates from the two containers. When one container issues `fsync`, the journaling module commits all the updates in the transaction and thus the one container has to wait until the updates from the other container are committed. This dependency can violate the performance isolation because one container can degrade the performance of another by issuing many `fsync` calls.

A performance dependency can be caused even if a single transaction solely contains updates from one container. Suppose transactions 2 and 3 contain updates solely from container *A* and *B*, respectively. Transactions *A* and *B* cannot be committed in parallel because transactions are serialized in the journaling module to keep the order of updates. Therefore, if two containers issue `fsync` at the same time, container *A*, for example, has to wait until transaction *B* is committed.

The sharing of journaling module among containers interferes with the disk I/O control of `cgroup` as well. As shown in Fig. 1, the journaling module is running outside of controlled containers. The journaling I/Os are overlooked by `cgroup` and not accounted for the container that initiated the updates. Suppose that `cgroup` divides the disk I/Os of container *A* and *B* into 70% and 30% share, respectively. If container *B* issues `fsync` calls frequently, its corresponding journaling I/Os are not accounted for the 30% disk I/O share. This results in violation of performance isolation between containers *A* and *B*; container *B* gets higher disk I/O share than 30%.

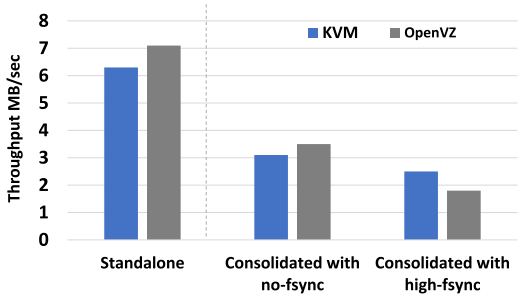


Fig. 2 Disk I/O throughput of FIO sequential-write benchmark. The graph shows 1) standalone, 2) collocated with no-fsync and 3) collocated with high-fsync cases.

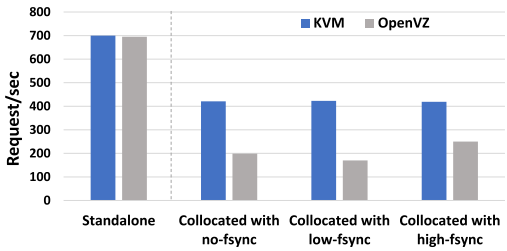


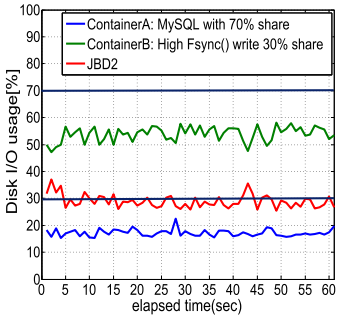
Fig. 3 MySQL throughput in KVM and openVZ. The graph shows 1) standalone, 2) collocated with no-fsync and 3) collocated with high-fsync cases.

2.4 DBMS Performance and Isolation in Containers

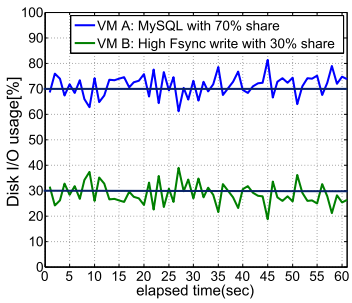
To confirm that the journaling has negative effects on DBMS performance and isolation, we performed a set of experiments. We consolidated a MySQL container with a container running either with no-fsync, low-fsync, or high-fsync workloads. The detailed experimental setup is shown in Sect. 4.1. We consider OpenVZ as a representative of containers, and compare the results with KVM in which each VM has its own journaling module and avoids all the journaling-related problems.

First, we conduct a preliminary experiment to examine the I/O performance in KVM and OpenVZ. Figure 2 shows the disk I/O throughput of the FIO sequential-write benchmark in three different cases: 1) stand-alone, 2) consolidation with no-fsync; where the collocated VM/container runs the no-fsync workload, and 3) consolidation with high-fsync; where the collocated VM/container runs the high-fsync workload. In the standalone case, OpenVZ outperforms KVM because of the lightweight nature of containers. When collocated with no-fsync, OpenVZ keeps its advantage over KVM, but when collocated with high-fsync, KVM outperforms OpenVZ because of the contention in the shared journaling module in OpenVZ.

Figure 3 shows MySQL throughput in OpenVZ and KVM. KVM performance of the standalone MySQL is competitive with OpenVZ because the number of disk I/Os in MySQL is smaller than in the FIO sequential-write



(a) Collocated with high-fsync in OpenVZ.



(b) Collocated with high-fsync in KVM.

Fig. 4 Disk I/O usage of MySQL in OpenVZ when collocating with high-fsync workloads. MySQL is given 70% share of disk I/O.

benchmark (30% less in MySQL than FIO benchmark). This diminishes the performance advantage of OpenVZ. When MySQL is collocated with no-fsync, low-fsync, and high-fsync benchmark, the performance of OpenVZ degrades because of the shared journaling module; `fsync` takes (120 ms) and (160 ms) in the standalone and the collocation with low-fsync, respectively.

The shared journaling module destroys performance isolation between containers. Figure 4 shows the disk I/O shares of the container/VM running MySQL and that running the high-fsync benchmark. Using the disk I/O control of `cgroup`, the container/VM running MySQL is given 70% share of disk I/O while the container/VM running the high-fsync is given 30% share. As can be seen from Fig. 4 a), the container running MySQL consumes only 20% share of disk I/O while the container running the high-fsync does 50 to 60% share. On the other hand, Fig. 3 b) indicates the VM running MySQL consumes 70% share of disk I/O and the VM running the high-fsync does 20% share in KVM.

Our previous work [13], [14] shows the in-depth analysis of this behavior. Based on the analysis, this current paper shows a mitigation of the journal-related problems in containers. Note that the problems are not peculiar to OpenVZ; our previous work focused on LXC containers. Our new experimental results on OpenVZ, shown in this paper, demonstrate OpenVZ cannot be a solution to the journal-related problems in containers.

3. A Quest for Best Configuration

Designing new file systems or developing novel journaling mechanisms require heavy implementation and involve non-negligible modifications to the kernel. Also, these solutions can be difficult to apply to existing cloud platforms. In this section, we present journaling-related problems can be gracefully mitigated by careful configuration of existing container platforms. Unfortunately, this configuration is not available on all container platforms and not provided on major cloud platforms even if the underlying container platform supports the configuration. Our configuration can be applied to the mainstream Linux and existing file systems without any modification. Section 3.1 shows a configuration that avoids bundled transactions in journaling modules. Section 3.2 shows the per-container accounting of journaling I/Os is possible. The combination of these two configurations solves the journal-related problems in containers.

3.1 Separating Journaling Module

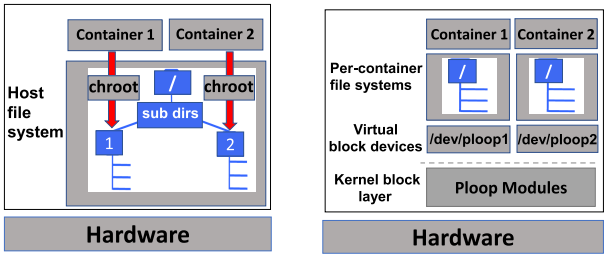
File system journaling, which is mandatory to guarantee the file system consistency, is the root cause of the poor I/O performance and the weak isolation in containers. The journaling problems are caused in particular due to bundled transactions. Since a journaling module is shared among containers, it batches updates from multiple containers into a transaction and commits that transaction to disk periodically or when `fsync` is invoked. If a single transaction contains updates from multiple containers, each container has to wait until the data belonging to other containers is flushed. Even if each transaction contains updates solely from one container, the transactions are serialized in the journaling module and cannot be committed in parallel.

A possible approach to mitigating the journaling problems is to provide a per-container journaling module. It gets rid of bundled transactions and avoids the performance dependency between containers.

By assigning a virtual block device to each container, each container is served by a separate journaling module and has its own journaling transactions that contain updates solely from the corresponding container. Since journaling modules in different containers can run in parallel, one container no longer has to wait for other containers to commit their updates.

A simple way to implement a virtual block device is to use a pseudo-device known as “loopback device”. A loopback device is a pseudo-device that makes an ordinary file accessible as a block device. However, the Linux loopback device has some limitations if used as a virtual disk for containers. First, the container file system suffers from the problem of double caching. Since a container file system is created on an ordinary file (a loopback device), both the host and the container file systems cache file contents, which leads to the well-known problem of double caching.

Second, direct I/O is not supported. Direct I/O is a way



(a) Normal approach (b) Ploop approach

Fig. 5 The architecture of "Normal approach" and "Ploop approach" of container implementation.

to bypass caching layer in the kernel. Direct I/O is important in DBMS because DBMS manages their own caches to avoid the double caching problem. Also, the direct I/O ensures that data is written immediately to disk instead of the kernel buffers first then later being written to the disk. This can aid in minimizing the data loss in DBMS. Direct I/O is supported by many databases like MySQL and Oracle.

Aside from these limitations, the Linux loopback device lacks relevant features in the clouds such as dynamic allocation, snapshot, and migration. These features are indispensable in managing containers in the data-centers for re-sizing the container to accommodate bursty workloads, the load balancing among servers, and for the backup and data protection.

Fortunately, OpenVZ supports "Ploop", a special implementation of the loopback device which overcomes all the limitations of the Linux loopback device. Figure 5 illustrates the Ploop virtual disk approach and the normal approach of providing file systems in containers. The normal approach uses `chroot` to provide a per-container sub-tree of the system-wide file system. Although this approach gives each container its own view of the host file system and semi-isolation from other containers, they still share the same host file system. This results in the sharing of kernel resources for managing files. All the containers share the same file system type, properties, total number of inode, and most importantly the cache layer and the journaling module.

In the Ploop approach, the Ploop module in the kernel block layer is responsible for presenting a virtual disk for each container. Each container has its own file system of different types and properties. Ploop has the I/O module that supports the direct I/O and avoids the double caching problem.

3.2 Journaling I/O Accounting

Providing each container with a virtual block device is not enough to overcome all journaling-related problems. Although each container has its own journaling module and transactions are separated, the journaling I/Os are not accounted because the `cgroup` overlooks the journaling I/Os from these per-container journaling modules. The journaling modules are managed by the kernel and are running outside the controlled containers.

The kernel process known as "jbd2" is responsible for performing the journaling I/Os. In case of per-container journaling, this kernel process is prepared for each container. Since they are outside of `cgroup` control, their I/Os are not accounted for the corresponding containers. For example, if two containers are performing I/Os and one container is given 70% disk share while the other is given 30% share, their corresponding journaling I/Os by jbd2 are not included in the disk share. This results in violating the performance isolation of disk I/O.

To solve this issue, the kernel jbd2 process should be included in the `cgroup` of its corresponding container. Both the journaling and the container's I/Os can be controlled and accounted together by the `cgroup` disk I/O controller. This can be implemented through `cgclassify` [33] functionality. It changes the `cgroup` of the jbd2 process from the kernel's root `cgroup` to the corresponding container's `cgroup`.

4. Experiments

This section shows the recommended configuration can overcome the journaling-related problems in containers. Section 4.1 describes the experimental setup. Section 4.2 presents the result and the improvement of assigning the container with a virtual block device. Section 4.3 shows the total performance improvement when the journaling I/O is being accounted for each container.

4.1 Experimental Environment

The experiment setup 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 64bit 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 new version of control group "`Cgroupv2`" [34] with the proportional-weight policy. FIO benchmark is used to generate I/O workloads of three types, either 1) no-, 2) low-, or 3) high-fsync workloads that have been explained in Sect. 2. OpenVZ 7.0.10 with its customized kernel and resources control is used for OpenVZ container. KVM-qemu 2.11.1 is installed on the host. The guest environments are exactly the same as the host. Each VM is allocated one virtual CPU that pins to one CPU core and 1 GB RAM with a raw disk partition allocated as secondary storage. The same configuration is applied to the 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 [35] generates workloads, which runs in a separated machine connected via Cisco 1 Gbit Ethernet switch.

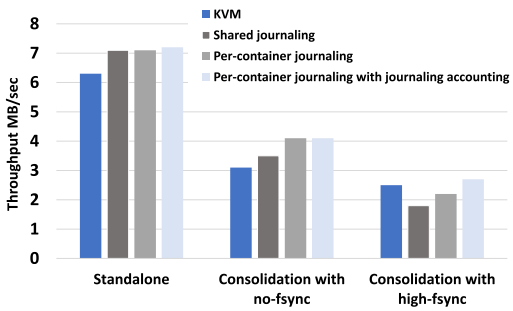


Fig. 6 Disk I/O throughput in KVM, shared journaling, and per-container journaling without/with accounting. A container/VM running the high-fsync workload is collocated with either 1) no-, 2) low-, or 3) high-fsync benchmarks.

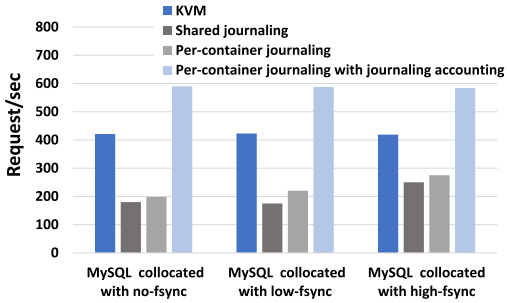


Fig. 7 MySQL throughput in KVM, shared journaling, and per-container journaling without/with accounting. MySQL container is collocated with either 1) no-, 2) low-, or 3) high-fsync workloads.

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.

4.2 Per-Container Journaling

Presenting each container with its own journaling module is achieved by using Ploop. Here, we evaluate the effectiveness of per-container journaling and whether this configuration has any tax on container's I/O performance. Figure 6 compares the I/O performance of the shared-journaling in OpenVZ container denoted by "shared journaling" and the per-container jounaling with Ploop denoted by "per-container journaling". For comparison, the figure shows the result of KVM. The result of "per-container journaling with journal I/O accounting" will be discussed later in Sect. 4.3. The experimental setting and the workloads are consistent with those of Fig. 2. In the standalone case, all containers outperform KVM regardless of the shared or per-container journaling. This verified that per-container journaling has no effects on I/O performance. In the consolidation case where both containers are performing no-fsync workload, the per-container journaling outperforms KVM and achieves better performance than the shared-journaling in the no-fsync consolidation case. The same results are obtained in the consolidation case with the high-fsync workload. This performance improvement is due to each container now has its own journaling module. The `fsync` latency is 58 ms and 80 ms in the per-container journaling and the shared journaling, respectively. However, the per-container journaling still performs lower than KVM in the high-fsync workload. This indicates the per-container journaling does not solve the journaling-related problems completely.

Figure 7 shows MySQL throughput when it is collocated with a container running either 1) no-, 2) low-, or 3) high-fsync workload. As it shown in the figure, MySQL throughput of the per-container journaling outperforms the shared-journaling in all cases. MySQL throughput is improved by up to 1.2x the per-container journaling. Table 1 shows the latency of `fsync` in the shared and per-container journaling. The latency of `fsync` in per-container journaling is smaller than that of shared journaling. The latency of `fsync` is reduced because no transaction is bundled in the per-container journaling. Compared with KVM, the per-container journaling still performs lower.

Table 1 Average fsync latency of MySQL when collocated with no-, low-, high-fsync workload. JA stands for journaling accounting.

Collocated workload	Shared journaling	Per-container journaling	Per-container journaling with JA
No-fsync	120.23 ms	103.80 ms	24.23 ms
Low-fsync	160.651 ms	92.65 ms	25.38 ms
High-fsync	96.20 ms	90.50 ms	24.75 ms

4.3 Combined Performance with Journaling Accounting

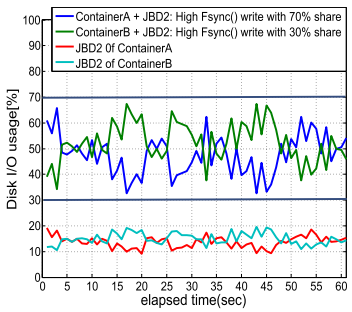
The results obtained with the per-container journaling indicates that using a virtual block disk with the container is not sufficient to overcome all the journaling-related problems. The per-container journaling performs lower than KVM for two reasons. First, journaling I/Os are still overlooked by `cgroup` because the journaling module runs outside of the controlled container. This affects the performance isolation between collocated containers because the overlooked I/Os affect the performance of the containers.

Second, the journaling process "JBD2", responsible for handling the journaling I/Os, is given equal share regardless of the share given to the corresponding container. Since JDB2 is a kernel process, it belongs to the root `cgroup` to which all the kernel processes belong by default. Even though we prepare a JDB2 for each container, all the per-container JDB2s belong to the same `cgroup` and are given equal share regardless of the I/O share of the container each JDB2 is responsible for.

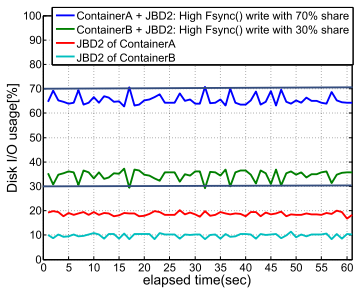
Figure 8 shows disk I/O usages of the per-container journaling without/with the accounting. In this experiment, two containers are lunched with disk I/O share set to 70% and 30%, respectively. A collocated container runs the high-fsync workload. As shown in Fig. 8(a), the per-container journaling without accounting cannot enforce performance isolation. Disk I/O usage of both containers fluctuates between 35%–65%. Figure 8(a) also indicates disk I/O usages

of the per-container JDB2s are almost the same around 15% for containers *A* and *B*. Although containers *A* and *B* are given different shares, the corresponding JDB2s are given an equal share because they belong to the same `cgroup` (the root `cgroup`).

On the other hand, the per-container journaling with accounting enforces the performance isolation as shown in Fig. 8(b). Container *A* and *B* consume 70% and 30% of disk I/O, respectively. JDB2 processes for container *A* and *B* consume 20% and 10%, respectively, because they belong to each `cgroup` to which the corresponding container belongs.



(a) Per-container journaling.



(b) Per-container journaling combined with journaling accounting.

Fig. 8 Disk I/O usage in per-container journaling and per-container journaling with journaling accounting". Both of containers run high-fsync workloads.

Table 2 Average fsync latency of the per-container journaling without/with journaling accounting.

Containers / Disk I/O shares	Per-container journaling	Per-container journaling with JA
container *A* / 70%	55 ms	45 ms
container *B* / 30%	53 ms	86 ms

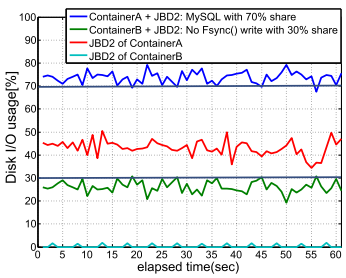
Table 2 shows the latency of `fsync` in the per-container journaling without and with journaling accounting. Without the accounting, the JDB processes for containers *A* and *B* are given the equal share. Hence, both containers *A* and *B* have almost the same `fsync` latency around 55 ms. With the accounting, `fsync` latency of container *A* is reduced to 45 ms while that of container *B* is increased to 86 ms. This indicates the container *A*' JDB process is given more share while the container *B*'s JDB process is less share.

In Fig. 6, the per-container journaling with the accounting outperforms KVM in all cases. This improved performance is obtained even in MySQL. Figure 7 shows the per-container journaling with the accounting beats KVM in throughput. MySQL throughput is improved by up to 1.5x compared to KVM and by up to 2.8x compared to the shared journaling. Table 1 shows `fsync` latency in the per-container journaling with the accounting becomes smaller than that of the per-container journaling.

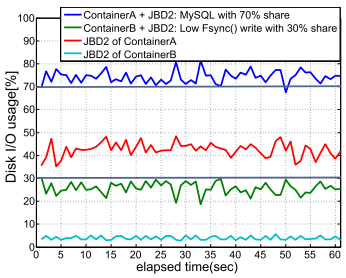
Aside from the performance, the per-container journaling with accounting has improved performance isolation. Figure 9 shows the disk I/O usage of the per-container journaling with accounting. A container running MySQL, which is given 70% share of disk I/O, is collocated with a container running either no-, low-, or high-fsync workload. In all cases, disk I/O share is around 70% in MySQL. On the other hand, the per-container journaling without accounting fails to enforce performance isolation. As shown in Fig. 10, MySQL container consumes around 25%, 30%, and 30%–50% share of disk I/O when collocated with no-, low- high-fsync workload, respectively.

5. Discussion

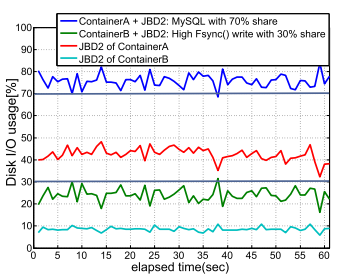
Disabling the file system journaling to overcome the journaling-related problems in containers is not acceptable especially in DBMS because the journaling is indispensable to guarantee the crash consistency. Developing a completely new file system or a journaling mechanism is not needed to solve the journaling issues. Our recommended configuration



(a) Collocated with no-fsync.

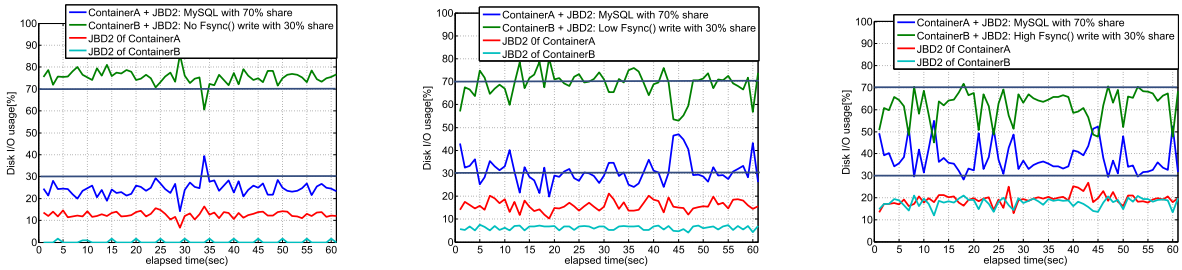


(b) Collocated with low-fsync.



(c) Collocated with high-fsync.

Fig. 9 Disk I/O usage in MySQL in per-container journaling with the accounting. Collocated with no-, low-, high-fsync workloads. MySQL is given 70% share and the other is 30% share. Performance isolation is improved more.



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

Fig. 10 Disk I/O usage in MySQL in per-container journaling. Collocated with no-, low-, high-fsync workloads. MySQL is given 70% share and the other is 30% share.

effectively alleviates the journaling-related problems in containers. We advocate the use of virtual block devices and the proper configuration for accounting journaling I/Os in containers. We denote that an ordinary loop device is not efficient and recommend the use of `Ploop` for virtual disks in containers. Although not all containers support the `Ploop`, they can adopt our configuration if they implement a similar virtual block device with a direct I/O support, preventing the double cache, and the required features for container's storage management like the snapshot and dynamic re-sizing.

Our performance analysis reveals that the using of a virtual block device is not sufficient to overcome the journaling problem in containers. The proper configuration of `cgroup` and journaling module is crucial to alleviate it. The quantitative analysis shows that the combination of the two methods can solve the journaling problem without re-design the file system or the journaling module.

Unfortunately, our recommended configurations are not widely adopted in major clouds. This results in poor I/O performance when consolidating DBMS and wastes the container's advantages over the VM, which results in violating the service-level agreement (SLA) and the financial loss for clouds provides. The use of VM to avoid the journaling problems comes with the cost of virtualization overheads. Our result proves that it is possible to use containers to get its performance gain without suffering from the journaling-related problems.

6. Related Work

Many works study the performance advantage of containers. Xavier et al. [36] compares Linux VServer [37], OpenVZ [38] and LXC with Xen 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. [39] study performance isolation among VMware, Xen, Solaris containers and OpenVZ. VMware imposes stronger isolation than the containers. However, the underlying issues that causes performance interference in containers is not investigated in these previous works.

Our previous works [13], [14] compare DBMS performance and isolation in container and VM. Our results show that KVM beats LXC in MySQL performance which is contrary to the general belive that container outperform VMs. Our analysis reveals that the file system journaling is the root cause of poor I/O performance and isolation in containers. Soltesz et al. [22] 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. [11] presents an evaluation of file operations of OverlayFS which is widely recognized method for improving I/Operformance in docker. According to their results, performance of file writing is severely low because of synchronization of data in the memory and storage. They suggest to disabling this synchronization to improve the I/O performance which is not acceptable for application like DBMS. Xavier et al. [40] 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 disk 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.

Kwon et al. [41] present a storage framework to enhance I/O performance and resource isolation of Docker containers in solid state device (SSD) storage. They achieve that by divided underlying hardware resource between containers to avoid resource conflict. Their investigation point out that the reasons for poor I/O performance and isolation in Docker are the sharing of same swap area among containers and the kernel. Hence when kernel perform I/Os due to page-in and page-out when running memory intensive containers, this swap area can cause storage resource conflict. In

our recommended configuration by using the per-container journaling, each container has its own viryual disk with the swap area and avoids the above problem.

Some works propose new file systems such as IceFS [9] and SpanFS [17] to provide logically separated units for independent journaling among containers. MultiLanes [18] 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. [19] 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. Our work proposed a configuration model for containers that overcomes journaling-related problems. Our proposed model is more straightforward and can be applied to mainstream file system without need for a complete new file systems or a novel journaling mechanism.

7. Conclusion

The file system journaling is the root cause of poor I/O performance and 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. Instead of designing a complete file system or developing a new journaling method, our work demonstrates careful configuration of containers can overcome the journaling-related problems. This configuration consists of presenting each container with its own virtual block device to achieve the per-container journaling, and properly configuring the control groups of the journaling processes. Our experimental results show that our configuration improves MySQL throughput up to 3.4x, and also achieves appropriate performance isolation comparable to that of KVM.

Acknowledgements

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

References

[1] “Well-Known Users of SQLite, A WEB page.” https://www.sqlite.org/famous.html, accessed 10-02-2020.
[2] “Salesforce, A WEB page.” https://developer.salesforce.com/page/Multi-Tenant-Architecture, 2018.
[3] “The facebook data center, a web page.” http://www.datacenterknowledge.com/the-facebook-data-center-faq-page-2/, accessed 10-02-2020.
[4] “Microsoft azure sql databaser, a web page.” https://azure.microsoft.com/en-us/services/sql-database, accessed 10-02-2020.
[5] “Google cloud sql, a web page.” https://cloud.google.com/sql/, accessed 10-02-2020.
[6] T. Lange, P. Cemim, M. Xavier, and C. DeRose, “Optimizing the management of a database in a virtual environment,” IEEE Symposium on Computers and Communications (ISCC), pp.181–192, 2013.
[7] Flashdba, “ Database Consolidation, A WEB page.” https://flashdba.com/2012/07/06/database-consolidation-part1/, accessed 08-02-2020.
[8] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci, “Cntr: Lightweight OS containers,” 2018 USENIX Annual Technical Conference (USENIX ATC 18), Boston, MA, pp.199–212, USENIX Association, 2018.
[9] L. Lu, Y. Zhang, T. Do, S. AI-Kiswany, A.C. Arpaci-Dusseau, and R. Arpaci-Dusseau, “Physical disentanglement in a container-based file system,” Proc. the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp.81–96, Oct. 2014.
[10] OpenVZ, “Ploop/Why, A WEB page.” https://wiki.openvz.org/Ploop/Why, accessed 08-02-2020.
[11] N. Mizusawa, J. Kon, Y. Seki, J. Tao, and S. Yamaguchi, “Performance improvement of file operations on overlayfs for containers,” Proc. IEEE Int. Conf. Smart Computing, pp.297–302, 2018.
[12] “OpenVZ open source container-based virtualization for Linux.” https://openvz.org/, 2019.
[13] A.A.A. Mardan and K. Kono, “Containers or hypervisors: Which is better for database consolidation?,” Proc. IEEE Int. Conf. Cloud Computing Technology and Science (CloudCom), p.564–571, 2016.
[14] A.A.A. Mardan and K. Kono, “When the virtual machine wins over the container: Dbms performance and isolation in virtualized environments,” J. Information Processing, vol.61, no.7, pp.1–9, 2020.
[15] A. Dusseau, H. Remzi, A. Dusseau, and C. Andrea, OPERATING SYSTEMS, Arpaci-Dusseau Books, 2014.
[16] “Linux cgroup resource management, a web page.”
[17] J. Kang, B. Zhang, T. Wo, W. Yu, L. Du, S. Ma, and J. Huai, “Spanfs: a scalable file system on fast storage devices,” Proc. USENIX Annual Technical Conference (ATC), pp.249–261, July 2015.
[18] J. Kang, B. Zhang, T. Wo, C. Hu, and J. Huai, “Multilanes: providing virtualized storage for os-level virtualization on manycores,” Proc. 12th USENIX Conf. File and Storage Technologies (FAST), pp.317–329, 2014.
[19] D. Park and D. Shin, “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, July 2017.
[20] “Ploop, Containers in a File.” https://wiki.openvz.org/Ploop, 2019.
[21] “Chroot.” https://linux.die.net/man/2/chroot, 2019.
[22] S. Soltesz, H. Pötzl, M. Fiuczynski, A. Bavier, and L. Peterson, “Container based operating system virtualization: a scalable, high-performance alternative to hypervisors,” SIGOPS Operating System Review, vol.41, no.3, pp.275–287, March 2007.
[23] “Linux container (LXC).” https://linuxcontainers.org/, 2018.
[24] Mircosoft, “Azure. Invent with purpose, A WEB page.” https://azure.microsoft.com/en-us/, accessed 08-02-2020.
[25] Google, “Google cloud platform, A WEB page.” https://cloud.google.com/, accessed 08-02-2020.
[26] Mircosoft, “Azure Container Instances documentation, A WEB page.” https://docs.microsoft.com/en-us/azure/container-instances/, accessed 08-02-2020.
[27] Google, “Google Kubernetes Engine documentation, A WEB page.” https://cloud.google.com/kubernetes-engine/docs/, accessed 08-02-2020.
[28] Documentation, “Linux namespace, a web page.” http://man7.org/linux/man-pages/man7/namespaces.7.html, accessed 01/02/2020.
[29] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, “An updated performance comparison of virtual machines and linux containers,” Tech. Rep. RC25482 (AUS1407-001), IBM Research Division, 2014.
[30] J. Corbet, “Notes from a container, A WEB page.”

https://lwn.net/Articles/256389/, accessed 10-11-2020.
[31] V. Prabhakaran, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Analysis and evolution of journaling file systems," Proc. 2005 USENIX Annual Technical Conference, pp.105–120, USENIX, 2005.
[32] "Anatomy of Linux journaling file systems." https://www.ibm.com/developerworks/library/l-journaling-filesystems/index.html, 2008.
[33] Cgclassify, " Linux Documentation, A WEB page." https://linux.die.net/man/1/cgclassify, accessed 10-02-2020.
[34] Cgroup2, "New version Cgroup2, A WEB page." https://www.kernel.org/doc/Documentation/-cgroup-v2.txt, accessed 10-02-2020.
[35] "Sysbench benchmark suite, A WEB page." https://github.com/akopytov/sysbench, accessed 10-02-2020.
[36] M. Xavier, M.V. Neves, F.D. Rossi, T.C. Ferreto, T. Lange, and C.F.D. Rose, "Performance evaluation of container-based virtualization for high performance computing environments," Proc. 21st Euromicro Int. Conf. Parallel, Distributed and Network-Based Processing (PDP), pp.233–240, IEEE, 2013.
[37] "Linux-VServer, A WEB page." http://linux-vserver.org/, 2018.
[38] "OpenVZ, a web page." https://openvz.org/, 2018.
[39] J. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, D.G. Hamilton, M. McCabe, and J. Owens, "Quantifying the performance isolation properties of virtualization systems," Proc. the 2007 Workshop on Experimental Computer Science, (ExpCS), p.6, ACM, June 2007.
[40] M. Xavier, M.V. Neves, F.D. Rossi, T.C. Ferreto, T. Lange, and C.F.D. Rose, "A performance isolation analysis of disk-intensive workloads on container-based clouds," Proc. 23rd Euromicro Int. Conf. Parallel, Distributed and Network-Based Processing (PDP), pp.253–260, IEEE, 2015.
[41] M. Kwon, D. Gouk, C. Lee, B. Kim, J. Hwang, and M. Jung, "Dc-store: Eliminating noisy neighbor containers using deterministic i/o performance and resource isolation," 18th USENIX Conf. File and Storage Technologies (FAST 20), pp.183–191, USENIX Association, Feb. 2020.

Appendix A: The Configuration of Containers

- Our configuration is based on OpenVZ container. The following steps show how to configure the container with a virtual block device.
  1. Editing the configuration file /etc/vz/vz.conf and setting VEFSTYPE to ext4.

     ```
     # grep VEFSTYPE /etc/vz/vz.conf
     VEFSTYPE="ext4"
     ```

  2. Rebuilding the OS templates cache for the new container with a virtual disk.

     ```
     # vzpkg --update cache
     ```

  3. Creating a container with a virtual block device "Ploop".

     ```
     # vzpkg create container-name --layout
     ploop --diskspace ??G --ostemplate ??
     ```

- Changing the `Cgroup` of the per-container JBD2 process to be included within the same `Cgroup` of its corresponding container.

  ```
  # cgclassify -g /sys/fs/cgroup/container-
  cgroup-directory JBD2-process-ID
  ```

Asraa Abdulrazak Ali Mardan received the BSc and MSc 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 Technology, and File systems.

Kenji Kono received his BSc degree in 1993, MSc 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.