A Thesis for the Degree of Ph.D. in Engineering

Study of Database Management System Performance and Isolation in Virtualization Environments

July 2021

Graduate School of Science and Technology
Keio University

Asraa Abdulrazak Ali Mardan

Acknowledgement

I would like to thank my advisor, Prof. Kenji Kono for his constant guidance during all the time of research. This dissertation would not have been possible without his advice and encouragement. I would like to thank my dissertation committee, Prof. Motomichi Toyama, Prof. Kenichi Kourai, and Dr. Takahiro Hirofuchi. This dissertation was improved by their insight comments and valuable feedback.

I am also thankful to my colleagues in the system software lab. who always gave me a hand and helped me when I need something or have a question.

Also, I would like to express my sincere gratitude to Ministry of Education, Culture, Sports, Science and Technology (MEXT) scholarship which gives me the opportunity to study in Japan and to get my PhD degree from Keio university.

Finally, I thank my family, my parents and sisters, for their support all these years. Without their support, encouragement, and financial help, many accomplishments in my life including this dissertation would not have been possible.

Abstract

Database management system (DBMS) is one of the foundational and largest applications in the cloud. Major cloud service providers like Amazon web services, Microsoft Azure, Google clouds offer DBMS as a service. The cloud employs virtualization to consolidate DBMSes for efficient resource utilization and to isolate collocated workloads. There are two major virtualization technologies: hypervisor-based (virtual machines) and operating-system-level virtualization (containers). The underlying virtualization technologies in the cloud have a critical impact on performance and isolation, especially in disk I/O. To guarantee the service-level agreement (SLA), the disk I/O performance and its isolation are important in DBMSs because they are inherently disk I/O intensive.

This dissertation investigates DBMS performance and isolation in containers and virtual machines. Containers are widely believed to outperform virtual machines because of their small virtualization overhead, but the sharing of the same kernel may violate the isolation. Virtual machines are expected to provide stronger isolation but suffer from performance overheads that come from devices' virtualization and running a complete OS. This trade-off between the performance and isolation in containers and virtual machines is not well understood. The need for a better understanding of the behavior of applications on these two virtualization technologies has become fairly desirable. This allows the cloud service providers to choose the appropriate virtualization technology for their application needs. In the case of DBMS, the need for high I/O performance and good isolation when disk I/O contentions occur. The failure to achieve these causes Quality-of-Service violations which results in a financial loss to clouds.

Containers have become widely used in clouds and preferred over virtual

machines to consolidate DBMS due to their near-native performance and lightweight deployment. The key finding in this dissertation is virtual machines outperform containers in DBMS performance. The analysis reveals that file-system journaling has negative effects on DBMS performance and isolation in containers. DBMS is an update-intensive application and causes a lot of file-system journaling. Journaling is very important to keep file-system consistency and for crash recovery. Hence, file-system journaling can not be disabled especially with DBMS applications.

The contribution of this dissertation is twofold. First, identifying the underlying causes behind file-system journaling problems in containers. Since containers share the same file-system, the sharing of journaling modules causes performance dependencies among containers. Also, file-system journaling interferes with disk I/O control of containers and violates isolation between them. Second, proposing a configuration method to overcome the journaling problems in containers. The method achieves per-container journaling without re-designing the file-systems or modifying the existing kernel. The results show that DBMS performance improves up to 3.4x in containers with the proposed configuration. Eventually, containers get their performance advantage and outperform virtual machines by 1.4x, and show an identical disk I/O isolation.

Contents

1 Introduction **1**
1.1 Motivation 3
1.2 Study Overview 4
1.3 Previous Studies 6
1.4 Organization 7

2 Related Work **8**
2.1 Investigating I/O Performance and Isolation 8
2.2 Exploring DBMS Performance and Isolation 10
2.3 Enhancing I/O Performance and Isolation 11
2.4 Designing New File-systems or Journaling Techniques 12
2.5 Proposing Storage Systems for containers 14
2.6 Summary 14

3 Background **16**
3.1 Hypervisor and OS-based Virtualization 16
3.1.1 KVM 17
3.1.2 LXC 18
3.1.3 OpenVZ 19
3.2 Disk I/O in Container and VM 19
3.3 Disk I/O Control by Cgroup 20
3.4 Disk I/O Performance and Isolation 23
3.4.1 Experimental Setup 23
3.4.2 Results 24
3.5 Summary 25

4 **DBMS Performance and Isolation** **27**
4.1 MySQL Performance and Isolation 27
4.1.1 Experimental Setup 28
4.1.2 Results 28
4.2 Analyzing DBMS Performance and Isolation 29
4.2.1 Investigating the effect of Fsync 31
4.2.2 File-system Journaling 32
4.2.3 Journaling Problems in Containers 34
4.2.4 Journaling Influence on MySQL Performance 38
4.2.5 Journaling Influence on MySQL Isolation 41
4.3 Summary 44

5 **Alleviating Journaling Problems in Containers** **48**
5.1 A Quest for Best Solution 49
5.2 Proposed Configuration Method 49
5.2.1 Per-container Journaling Module 50
5.2.2 Journaling I/O Accounting 52
5.3 Experiments 53
5.3.1 Per-container Journaling 53
5.3.2 Combined with Journaling I/O Accounting 57
5.3.3 Improvement of DBMS performance and isolation . . . 60
5.4 In-memory Database Performance 61
5.5 Discussion 68
5.6 Summary 70

6 **Conclusion** **72**
6.1 Contribution Summary 72
6.2 Future Work 73

Bibliography **75**

List of Papers **83**

List of Figures

3.1 Architectures of hypervisor and OS-based virtualization. . . . 18
3.2 Interior structure and I/O paths of KVM and LXC 20
3.3 Illustration of Cgroup with proportional-weight policy. 21
3.4 Proportional-weight policy in `cgroup` fails to control disk I/O bandwidth. 22
3.5 Disk I/O throughput in KVM, LXC, and OpenVZ. 24
3.6 Disk I/O throughput in KVM, LXC, and OpenVZ in consolidation case. 25
3.7 Performance isolation in KVM, LXC, and OpenVZ. 26

4.1 MySQL throughput in KVM, LXC, and OpenVZ. 29
4.2 MySQL performance isolation in KVM, LXC, and OpenVZ . . 30
4.3 Throughput of high-fsync workload in KVM, LXC, and OpenVZ. 32
4.4 A typical journaling file-system. 33
4.5 File-system journaling in container virtualization. 34
4.6 Disk I/O isolation when the low-fsync is collocated with no-fsync workload. 37
4.7 Disk I/O isolation when the low-fsync is collocated with high-fsync workload. 38
4.8 MySQL throughput with 70% share of disk I/O. 39
4.9 MySQL throughput with 30% share of disk I/O. 40
4.10 Disk I/O usage in MySQL in LXC with 70% share. 42
4.11 Disk I/O usage in MySQL in OpenVZ with 70% share. 43
4.12 Disk I/O usage in MySQL in KVM with 70% share. 44
4.13 Disk I/O usage in MySQL in LXC with 30% share. 45
4.14 Disk I/O usage in MySQL in OpenVZ with 30% share. 46

4.15 Disk I/O usage in MySQL in KVM with 30% share. 47

5.1 The architecture of "Normal approach" and "Ploop approach". 50
5.2 Disk I/O throughput with the proposed configuration. 54
5.3 MySQL throughput with the proposed configuration. 55
5.4 MySQL throughput with the proposed configuration, MySQL is given 30% share. 56
5.5 Disk I/O isolation in per-container journaling without/with accounting. 58
5.6 Throughput of high-fsync workload with the proposed configuration. 59
5.7 Disk I/O isolation of MySQL in per-container journaling with the accounting, MySQL is given 70% share. 62
5.8 Disk I/O isolation of MySQL in per-container journaling, MySQL is given 70% share. 63
5.9 Disk I/O isolation of MySQL in per-container journaling with the accounting, MySQL is given 30% share. 64
5.10 Disk I/O isolation of MySQL in per-container journaling, MySQL is given 30% share. 65
5.11 Disk I/O isolation when two MySQL containers/VMs are collocated together. 66
5.12 MySQL throughput when two MySQL containers/VMs are collocated together. 66
5.13 Typical In-memory database with data persistency. 67
5.14 Redis throughput in standalone case. 67
5.15 Redis throughput in consolidation case. 68
5.16 Redis throughput with the proposed configuration. 69

List of Tables

4.1 Average fsync latency of the high-fsync workload. 35
4.2 Average fsync latency of MySQL with 70% share of disk I/O. 39
4.3 Average fsync latency of MySQL with 30% share of disk I/O. 40

5.1 Average fsync latency of MySQL with the proposed configuration. 56
5.2 Average fsync latency of MySQL with the proposed configuration, MySQL is given 30% share. 57
5.3 Average fsync latency of the per-container journaling without/with accounting. 59
5.4 Average fsync latency of Redis in consolidation case. 68

Chapter 1

Introduction

In the era of cloud computing, many organizations as well as individuals are turning to the cloud services and applications. It is estimated that about 94% of enterprises already use a cloud service [17]. The cloud service simply means the delivery of computing services including servers, storage, databases, networking, and applications over the Internet [70]. The cloud provides a lot of benefits like, dynamic scalability of resources, remote management and maintenance, pay-as-you-go model, which means the pay only for resources that are being used. These benefits help in lower the operating cost, running the infrastructure more efficiently, and scale as the business needs change [70].

The cloud relies on virtualization technology, which is the building block of the cloud datacenters. Virtualization is needed to achieve server consolidation for efficient resource usage and to meet the growing demands of resources with resource sharing. There are two main virtualization technologies: and hypervisor-based and operating system (OS)-based virtualization. In hypervisor-based virtualization, a special program called hypervisor virtualizes hardware resource allowing multiple virtual machines (VMs) to run on a single physical machine. Each VM runs a complete guest OS. The OS-based virtualization shifts the layer of virtualization from hardware to the OS level by creating multiple virtual units at the user space known as containers. These containers share the same host kernel but are isolated from each other through private namespaces [33] and resource control mechanisms [34] at the OS level.

Containers are lightweight in size and provide a near-native performance, faster provisioning, better scalability, and fewer resources consumption compared to VMs [28]. Containers allow a rapid building, testing, deployment, and development of applications in the cloud [56]. These advantages make containers widely used and preferred over VMs to deploy applications in the cloud [76]. Cloud service providers like Amazon web services(AWS) [1], Google cloud platform [23], and Microsoft azure [52] adopt containers to deliver their services and applications.

One of the important applications and common services in the cloud is the database management system (DBMS). Many popular Web services and applications such as Facebook [55], Dropbox [75], and Saleforce [77] make use of DBMS. Major cloud services providers offer DBMS as a cloud service like Microsoft's SQL Azure [51], Google Cloud SQL [22], and AWS databases [2]. It is estimated that 45% of companies use a cloud-based DBMS and it will continue to increase with the next years [61]. It is expected that by 2022, 75% of all DBMS will be deployed and migrated to the cloud [20]. On the other hand, DBMS is one of the most deployed applications with containers in the cloud [11]. According to the real-world container's use report [10], popular DBMSes like MySQL, Redis, Postgres, and MongoDB are the most commonly deployed container's images in the cloud.

The deployment of DBMS in the cloud means the application runs on a shared virtualized environments. The performance and isolation are important in these virtualized environments, where multiple users' applications are consolidated on the same server. Users expect their applications run in isolated manner and to get the performance they pay for. Failing to achieve isolation results in performance degradation as users' applications negatively affected by each other. This leads to quality-of-service violations which result in financial loss to cloud services providers [78] [25]. Hence, DBMS consolidation in the cloud must be done while observing per user performance guarantees and good isolation when resource contentions occur.

The disk I/O contention is very common in the cloud when multiple I/O applications consolidated togather [24] [26] [73]. Since DBMS is an I/O intensive application and largely deployed in the cloud, the disk I/O performance and isolation are very important in such an application. In this

dissertation, I study the performance and isolation of DBMS consolidation in virtualization environments.

1.1 Motivation

The underlying virtualization technologies in the cloud have critical impact on the performance and isolation, especially in disk I/O, in DBMS. To guarantee the service-level agreement (SLA), the disk I/O performance and its isolation are important in DBMSs because they are inherently disk I/O intensive. In containers and VMs, the performance and isolation are intersected. Containers overcome VM performance overheads that come from devices virtualization by the hypervisor and running a complete OS inside each VM. However, containers share the same kernel components like file-system and buffer caches, the 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 isolation because no kernel components in the guest OSes are shared among VMs.

Although containers are preferred over VMs to avoid virtualization overhead, the trad-offs between performance and isolation in containers and VMs are not well understood. There is no sufficient studies regard the resource contention and the effect of kernel sharing on performance isolation among containers. This prevents the cloud services providers from selecting appropriate virtualization technology for application performance. For DBMS, storage I/O is mainly the key factor that determines the overall performance, despite that the other resource contention like memory and network I/O may also affect the performance. This dissertation focus on storage I/O as the first step toward investigation the performance interference among virtualized environments. Investigating I/O performance and isolation of containers and VMs answers the question of “What virtualization technology is better for DBMS consolidation in the cloud?”.

1.2 Study Overview

Throughout this dissertation, I study DBMS performance and isolation in virtualization environments. I investigate the disk I/O performance and isolation in containers and VMs. Disk I/O performance and isolation are critical in DBMS which orchestrates and performs a large number of disk I/Os. Surprisingly, our results show that DBMS performance is better in VMs than containers. VMs outperform containers by up to 2.4x in DBMS performance. This is contrary to the general belief that containers outperform VMs because of negligible virtualization overhead. Furthermore, disk I/O isolation is very terrible when consolidating DBMSes in containers. A container given 30% share of disk I/Os consumes 70% share although the resource control mechanism enforces disk I/O limits.

Our analysis reveals that the sharing of journaling module in containers degrade DBMS performance and violates disk I/O isolation. The journaling module is the kernel component that is responsible for handling file-system journaling. The journaling records updates not yet committed to the file-system and provides backup and recovery capabilities. The journaling is very important to guarantee consistency and for crash recovery in file systems [62] [13]. Hence, file-system journaling should not be turned off especially in update-intensive applications like DBMS. DBMS invokes a lot of fsync, a system call that ensures updates are written to disk and triggers file-system journaling.

In this dissertation, I identify the underlying causes behind the journaling problems in containers. The journaling degrades I/O performance in containers because of the following reasons. Since the journaling module is shared inherently among containers, it causes performance dependency among containers. To guarantee consistency, existing file-systems typically use journaling with transactions. A journaling module batches updates from multiple containers into a single 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 jour-

naling 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.

In addition, the journaling interferes with disk I/O control of the kernel resource control mechanism known as `cgroup` [34]. 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. This violates I/O isolation among the containers.

I propose a configuration method to alleviate these journaling problems in containers. I show that the careful configuration of containers can gracefully solve the journaling problems without re-designing the file-systems or modifying the existing kernel. Our configuration achieves per-container journaling by first, providing 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 bottleneck of the shared journaling module and its performance dependencies. Second, these per-container journaling modules are still running outside of containers and their I/Os are still overlooked. To account journaling I/Os, a proper configuration of kernel processes that handle the journaling operations is needed. These journaling processes should belong to the same `cgroup` of their corresponding containers.

This configuration is not widely adopted in the clouds because most container implementations do not support virtual disk device. 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 problems. The quantitative analysis shows the feasibility of our configuration in improving DBMS performance and isolation in containers. The 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 outperform VMs by 1.4x and show a comparable disk I/O isolation to that of VMs.

Finally, I explore the performance of an in-memory database system that gains popularity recently. The in-memory database is a DBMS that relies on main memory for data storage instead of disk storage. Our results show that

the sharing of the journaling module in containers degrades the in-memory database's performance as well. Our proposed configuration improves the in-memory database's performance and mitigates the journaling effects.

1.3 Previous Studies

There are no sufficient studies regarding DBMS performance and isolation under hypervisor and OS-based virtualization. The I/O performance and isolation are investigated in hypervisor-based virtualization [87, 5, 50, 7, 29, 65, 57, 86]. However, there is a scarcity in the studies on OS-based virtualization and the comparison between containers and VMs.

Some studies [14, 69, 9, 54, 64] compared disk I/O performance in containers and VMs but focus only on the performance; the I/O isolation is not addressed. Other studies [46] [84] [72] [19] compared disk I/O isolation in VMs and containers and show that VMs have a stronger isolation. However, the underlying causes that affect containers' I/O isolation is not investigated.

The studies in [74] and [85] explored the DBMS performance isolation in containers and showed that the database performance suffers from interference when it is collocated with I/O intensive workloads. However, no analysis are performed to understand how the interference occurs in containers. The study in [74] suggested that the sharing of buffer cache is the cause of database interference with I/O workloads. In this dissertation, I reveal that file-system journaling is the root cause of the interference. DBMS suffers from interference even when the buffer cache is not shared.

Other studies focused on enhancing the I/O performance in containers. The study in [37] divides underlying hardware resource between containers to avoid resource conflict. The study showed that the sharing of the swap area causes the resource conflict in solid state device storage (SSD). In our recommended configuration by using the per-container journaling, each container has its own virtual disk with the swap area and avoids the above problem. The study in [53] suggested to disable the data synchronization inside containers to improve I/O performance. However, such solution is risky in update intensive application like DBMS and equivalent to disabling file-system journaling.

Some OS researchers worked on designing a completely new file systems [43] [32] or developing novel journaling mechanisms [60] to improve file-system journaling. The study in [31] proposed a virtualized storage device for each container on top of which an isolated I/O stack is built. These solution can mitigate some of journaling problems in containers. However, all of these works involve non-negligible modifications to the kernel, and the existing file systems and I/O stack must be replaced to utilize it. Hence, these solutions are difficult to deploy on current cloud platforms. In this dissertation, I quest for possible solutions to alleviate the journaling problems in containers without any code modification. We show that these problems can be overcomed without re-designing the file systems or modifying the existing kernel.

1.4 Organization

This dissertation is organized as follows. Chapter 2 describes the related works in detail. Chapter 3 explains the background of container and VM architectures and the overview of disk I/O control in them. The chapter also compares the disk I/O performance and isolation in containers and VMs. Chapter 4 investigates DBMS performance and isolation in containers and VMs. The chapter highlights the motivation by showing that containers are not suitable for DBMS consolidation despite outperform VMs in I/O throughput. The chapter analyzes DBMS performance and isolation, and describes the file-system journaling problems in containers. Chapter 5 proposes a configuration method for containers to alleviate file-system journaling problems in containers. The quantitative analysis confirms the the feasibility of our recommended configuration in overcoming the journaling problems. Finaly, chapter 6 concludes this dissertation and discusses the future work.

Chapter 2

Related Work

As containers have grown in popularity during recent years, they gain attention from researchers to study their performance and compare it with VMs. Most of these research efforts focus on exploring the container's performance advantage over VM's performance overhead. In an application like DBMS which is an I/O intensive application, the I/O performance and isolation are very important. The I/O performance and isolation of VMs have been studied well in the literature [87, 5, 50, 7, 29, 65, 57, 86]. However, there are no sufficient studies regarding I/O isolation in containers and the effect of kernel sharing on performance isolation among containers.

This chapter overviews the existing studies and discusses the importance of this dissertation. Since file-system journaling affects DBMS performance and isolation in containers, I review the studies that improve file-system journaling as well. I divide the studies into categories based on whether they target the I/O performance and isolation in containers, or they target file-system journaling problems. I summarize the pros and cons of each study within each category.

2.1 Investigating I/O Performance and Isolation

Numerous researchers have studied the performance of containers and compared it with that of VMs. However, most of these studies focus only on

performance, the performance isolation is not addressed. Researchers from IBM compared the performance of a docker container [12] and a KVM [35], a hypervisor-based VM [14]. Their work focused on a single container or VM performance and compared them to native non-virtualized performance to isolate and understand the overhead introduced by VM. Their results showed docker outperforms KVM in disk I/O with a different type of I/O workloads. Same results are obtained in [54] which compared KVM, docker, and LXC container [44]. Both LXC and docker outperform KVM in disk I/O and archive near native-performance. Regola et al [69] evaluated the I/O performance of OpenVZ [16] container and KVM in high-performance computing (HCL) environment. OpenVZ outperforms KVM in I/O throughput and achieves near-native performance. Che et al [9] compared disk I/O performance in Xen [4], another hypervisor-based virtualization, with KVM and OpenVZ. The results show that OpenVZ has a comparative I/O performance to Xen but better than that of KVM.

Some studies explored disk I/O isolation in containers and compared it with VMs. Matthews et al studied performance isolation in OpenVZ, Solaris container [81], Xen, VMware [79], another hypervisor-based virtualization. The scenario consisted of running two VMs/containers and dividing system resources between them. The performance of applications is measured upon one of the VM/container while the other VM/container is performing a stress test with benchmarks. VMware imposes stronger isolation than Xen and the containers. Both OpenVZ and Solaris containers showed a performance degradation and suffered from poor isolation in disk I/O intensive workloads. However, the underlying mechanism that causes performance interference in containers is not investigated. Similarly, Xavier et al [84] evaluated performance isolation in LXC and OpenVZ with Linux VServer [80], one of the oldest implementation of a Linux container-based system. The results showed that disk I/O isolation is violated in all container systems but the underlying causes are not investigated. Surya et al [18] [19] studied the interference among multiple containers and how different types of workloads, when scheduled together affect the performance of each other. In the case of I/O intensive workloads, containers suffered from performance interference. They suggested that since the caching of the file-system is taken care of by

a shared kernel, this can result in one container having an impact on the performance of other concurrent containers. Sharma et al [72] compared the performance isolation in LXC and KVM and showed that despite LXC outperforms KVM in disk I/O, it lacked the I/O isolation between containers. They suggested that the sharing of host OS block layer components like the I/O scheduler increases the performance interference for disk workloads in containers.

2.2 Exploring DBMS Performance and Isolation

There is a scarcity in the studies that investigate DBMS performance and isolation in containers. Felter et al [14] compared the performance of docker and KVM against two database systems, MySQL and Redis. Their results showed that the container outperforms VM in both MySQL and Redis throughput. However, they didn't study performance isolation when DBMS is collocated with other containers/VMs. Soltesz et al. [74] compare performance isolation of database application in VServer container and Xen, using Open Source Database Benchmark (OSDB). The performance of the VServer container running a database application is impacted by other containers running I/O intensive workloads. They suggested that I/O intensive containers monopolize the buffer cache to degrade the database application. Our investigation reveals that the file-system journaling disturbs database performance 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. Xavier et al. [85] compared LXC and KVM in terms of the performance interference in DBMS. According to their study, LXC suffers more severely from interference than KVM when a database is collocated disk with I/O-intensive workloads. They suggested that `cgroup` is working properly when using it to restrict resources in KVM than LXC. However, no analysis is conducted to understand how performance interference occurs in LXC and why the resource control is failed. In this dissertation, I explain that file-system journaling interferes with disk I/O control of `cgroup` in containers. Also, I show that KVM beats LXC not

only in I/O isolation but also in DBMS performance as I consider the case of journaling-intensive workloads.

2.3 Enhancing I/O Performance and Isolation

A few studies worked on or suggested a solution to improve I/O performance and isolation in containers. Kwon et al. [37] presented a storage framework to enhance I/O performance and resource isolation of Docker containers in solid-state device (SSD) storage. They achieved that by divided underlying hardware resources between containers to avoid resource conflict. The divided SSD is achieved by implementing multiple NVM sets in real hardware to provide concurrent storage accesses. Their investigation points out that the reasons for poor I/O performance and isolation in docker are the sharing of the same swap space among containers and the kernel. The swap is a storage space that is used when the amount of physical memory is full. If the system needs more memory space, inactive pages in memory are moved to the swap space [67]. Hence when the kernel performs I/Os due to page-in and page-out when running memory-intensive containers, this swap space can cause storage resource conflict and degrade the I/O performance. However, this work only targets I/O performance in SSD storage and requires a hardware modification. It doesn't address the resource contention of the shared journaling module in containers. In this dissertation, our recommended configuration by using the per-container journaling, each container has its virtual disk device with its own swap space and avoids the above problem as well.

Mizusawa et al. [53] presented an evaluation of file operations of OverlayFS which is a widely recognized method for improving I/O performance in docker. OverlayFS is a union mount filesystem and one of the storage drivers of docker. According to their results, the performance of file writing is severely low because of the synchronization of data in memory and storage. The write operations required a remarkably long time due to data synchronization. They suggest disabling this synchronization to improve the I/O performance. However, such a solution is not acceptable for an application

like DBMS. Similarly, disabling file-system journaling to avoid journaling problems and to improve I/O performance is not acceptable. It has the risk of losing the user's data and file-system consistency.

Mavridis et al [47] suggested combining both containers and VMs to enhance the isolation among containers. They suggested running a group of containers on top of VMs instead of running all containers on the bare-metal host machine. This method can reduce the resource contention on the shared resource of host OS among containers. They evaluated the performance overhead on CPU, memory, network, and disk I/O that results from running containers inside VMs. They compared the performance overhead under KVM, Xen, and Hyper-V [82] VMs to determine which virtualization adds less overhead. Although this approach has been used by some cloud platforms to enhance the isolation and security among containers; but it adds a non-negligible overhead on I/O performance. Also, the group of containers that run inside VM, still share the same journaling module of VM. Hence, such a configuration approach is not suitable to avoid file-system journaling problems in containers.

2.4 Designing New File-systems or Journaling Techniques

Some OS researchers proposed new file-systems or journaling techniques to enhance file-system journaling. Such works can contribute to mitigating some journaling problems in containers. Lu et al [43] proposed IceFS, a novel file system that separates physical on-disk structures of the file-system. They create new file-system abstractions known as "cubes" that include logically related files and directories. These cubes allow concurrent file-system updates. IceFS includes novel transaction splitting machinery to enable per-cube journaling to file system, thus disentangling I/Os traffic in different cubes. Normally, file-system journaling uses bundled transactions that group updates from many files together. This causes performance dependencies between I/O applications that update files and directories. The transaction splitting allows file-system journaling to commit transactions from different

cubes in parallel which increases the I/O performance of applications. IceFS can solve the bundled transactions' problem in containers, however, they still share the same journaling module that interferes with disk I/O control of `cgroup`. Also, the existing file-system must be replaced to utilize IceFS benefit.

Similarly, Park et al [60], proposed iJournaling, a new journaling technique that adopts the transaction splitting approach. They enable journaling at the file level by creating per-file transactions instead. Hence, if the journaling is invoked by a fsync call, ijournaling commits only the transaction related to the fsynced file without flushing the other files' updates like in bundled transaction. The file-level transaction has only the related updates of the fsynced file and takes a shorter time to be committed. This enhances the fsync latency and improves I/O performance. Such journaling techniques can improve the performance of update-intensive containers but do not overcome all the journaling problems. For example, fsync calls serialization from multiple containers and uncounted journaling I/Os are not addressed.

Kang et al [32] introduced SpanFS, a novel file system that consists of a collection of micro file system services called domains. SpanFS distributes files and directories among the domains, provides a global file system view on top of the domains. The storage device blocks are partitioned among the domains. Each domain has its on-disk and in-memory data structures, and its own kernel services like journaling. This design enables independent and parallel journaling per-domain. Using such a file-system can overcome journaling problems if each container runs within one domain. However, since the number of domains is limited, running multiple containers on a domain makes containers share the same journaling module again. Besides the cost and difficulties in managing domains and storage partitions, the existing file-system is needed to be replaced.

2.5 Proposing Storage Systems for containers

Containers share the same storage system and kernel I/O stack of host OS. This can lead to I/Os contentions and poor scalability on many CPU cores among containers. To eliminate storage system contentions, some works proposed a dedicated storage system for containers. Kang et al [31] introduced MultiLanes, a virtualized storage device for each container on top of which an isolated I/O stack is built. They achieved that by partitioning kernel and virtual file-system data structures and creating a file-based virtualized block device for each container. The virtualized storage device allows each container to have its guest file-system and journaling module. However, this work requires a non-negligible file-system modification and the kernel I/O stacks must be replaced to utilize it. OpenVZ [16] developers show that file-system journaling can be a serious bottleneck in containers and try to address this problem. If one container fills up in-memory journal with lots of small operations leading to file metadata updates, all the other containers I/O will block waiting for the journal to be written to disk [59]. OpenVZ implements a special virtual block device for containers atop of it, a guest file system is running. Although each container will have its own file-system and journal, this solution alone does not solve the journaling problems in containers as it will be shown in this dissertation.

2.6 Summary

Several works discussed the performance of containers and compared it with VMs. They showed that the container achieves higher performance than VMs in disk I/O. However, the existing work does not focus on performance isolation when multiple VMs/containers are consolidated. The trad-off between disk I/O performance and isolation in containers and VMs are still not well understood. Some works compared DBMS performance in containers and VMs. However, they did not investigate disk I/O isolation in DBMS, for which the containers cannot provide comparable performance with VMs.

The performance isolation in disk I/O for containers is not easy to achieve. Some works suggested dividing the underlying storage among containers to avoid I/O contentions or running containers inside VMs to enhance the isolation. Other work suggested disabling data synchronization to enhance I/O performance in containers. However, non of these works solve file-system journaling problems that affect I/O performance and isolation in containers.

New file systems or journaling techniques have been proposed to overcome some of the file-system journaling limitations, like providing logically separated units for independent or parallel journaling within the file-system. These novel file systems do not overcome all of the journaling problems in containers, and the existing file systems or kernel I/O stack must be replaced to utilize them. Some works introduced a virtualized storage device to provide each container with its own file system. However, this solution alone does not solve all of the journaling problems and their effects on I/O isolation in containers.

Chapter 3

Background

The objective of this chapter is to describe the background of the container and virtual machine architectures, and the overview of disk I/O control in them. The chapter also shows the comparison of disk I/O performance and isolation in the container and virtual machine. KVM is chosen as representative of hypervisor-based virtualization, While LXC and OpenVZ a representative of OS-based virtualizations. The results confirm that the container is better than the virtual machine in disk I/O performance and shows a comparable disk I/O isolation.

3.1 Hypervisor and OS-based Virtualization

In traditional virtualization, a special program called hypervisor runs on top of the host OS and virtualizes the underlying physical hardware. This enables multiple virtual machines (VMs) to run on a single physical machine as shown in Figure 3.1. The result is that each VM contains a guest OS, a virtual copy of the hardware that the OS requires to run, and an application and its libraries and dependencies. Instead of virtualizing the underlying hardware, the OS-based virtualization shifts the layer of virtualization and starts abstractions at the OS level by creating multiple virtual units at the user-space known as containers. Containers share the same host OS and contain only the application and its associated libraries and dependencies. This gives the lightweight advantage of containers over VMs when it comes to

application deployments like faster provisioning, better scalability, and fewer resource consumption. Containers are built and isolated from each other by two underlying Linux kernel technologies:

Namespaces. Namespaces are a kernel mechanism for limiting the visibility that a group of processes has of the rest of a system. For example limiting the visibility to certain process trees, network interfaces, user IDs, or filesystem mounts [45]. It uses to create an isolated container that has no visibility or access to objects outside the container. A namespace provides an abstraction for a kernel resource that makes it appear to the container that it has its own private, isolated instance of the resource [72]. Processes running inside the container appear to be running on a normal Linux system although they are sharing the underlying kernel with processes located in other namespaces. Linux implements namespaces for isolating: process IDs, user IDs, file system mount points, networking interfaces, IPC, and host names [14].

Cgroups. `cgroups`, which stands for control groups, is a kernel mechanism for limiting and controlling the resources consumption by a group of processes running on a system. For example, you can apply CPU, memory, network, or IO quotas [45]. Linux implements `cgroup` for each major resource type: CPU, memory, network, and disk I/O. 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 [49]. The OS kernel provides access to multiple *controllers* (also called subsystems) that limit the resource usage; for example, the I/O controller limits disk usage, the CPU controller limits CPU usage, etc. A container's resource consumption can be controlled by simply changing the limits of its corresponding `cgroup`.

3.1.1 KVM

KVM (Kernel-based Virtual Machine) is an open-source hypervisor-based virtualization technology built into a Linux system. As a kernel module added into Linux, KVM turns Linux into a hypervisor that allows a host machine to run multiple isolated VMs [35]. The hypervisors need some OS-

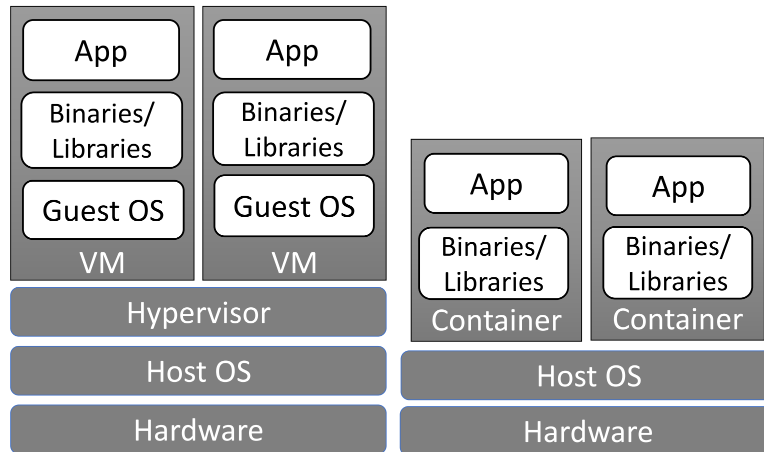


Figure 3.1: Architectures of hypervisor and OS-based virtualization.

level components such as a memory manager, process scheduler, I/O stack, device drivers, a network stack, and more to run VMs. KVM has all these components because it's part of the Linux kernel [68]. In KVM, each VM is implemented as a process by running an unmodified guest OS inside a Linux process. The VM is scheduled by the standard Linux scheduler, with dedicated virtual hardware like a network card, CPU, memory, and disks [68]. KVM uses hardware virtualization features in recent processors to reduce complexity and overhead such as Intel VT or AMD-V. KVM supports both emulated I/O devices through QEMU [6] and para-virtualized I/O devices using virtio [71]. QEMU is a device emulator and virtualizer that is used to simulate the VM's I/O and triggering the real I/O. Virtio is a para-virtualized driver that is used to accelerate I/Os in VMs. The combination of these two technologies is used to reduce I/O virtualization overhead in KVM [48].

3.1.2 LXC

LXC (Linux Container) is an open-source OS-based virtualization technology for or running multiple isolated containers in Linux systems. LXC share the same kernel of the host OS and are isolated from each other through private namespaces and resource control mechanisms of `cgroup`. During the container startup, by default, process IDs, user IDs, IPC, file system mount

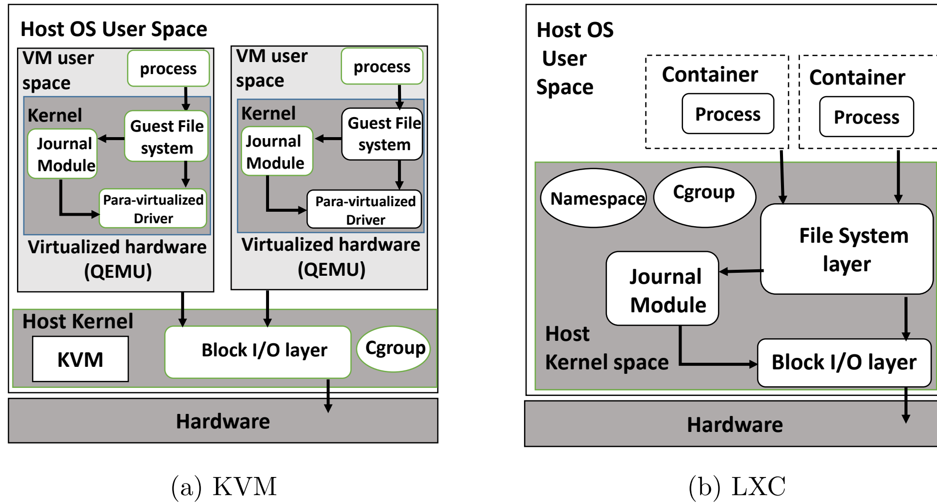
points, networking interfaces, are virtualized and isolated through the PID namespace, IPC namespace, and file system namespace, and network namespace respectively [84]. LXC offers an environment similar to VM but without the overhead that comes with running a separate kernel and simulating all the hardware [44]. The processes inside the container are run and handled like any regular process on the host OS. The process control and resource consumption are accomplished by `cgroup` of their corresponding containers.

3.1.3 OpenVZ

OpenVZ (Open Virtuozzo) is another open-source OS-based virtualization technology for Linux systems. Unlike LXC, OpenVZ uses a custom kernel instead of a mainstream Linux kernel. The custom kernel is a modified Linux kernel with the function of virtualization, isolation, checkpointing, live migration, and resource management [9]. OpenVZ introduces four resource management components named user beancounters (a set of limits and guarantees done through control parameters), fair CPU scheduling, disk quotas, and I/O scheduling [84]. In the same way as LXC, OpenVZ uses kernel namespaces to provide resource isolation among containers. OpenVZ supports the container with a virtual block device which allows each container to have its own file system. Normally, containers share the same file system of host OS where each container is just a directory of files that is isolated using `chroot` [40]. The `chroot` is a linux operation uses to provide a container with its root directory, which is a sub-tree of the host file system [74].

3.2 Disk I/O in Container and VM

Containers and VMs are different in how their disk I/Os are executed and handled on the host machine. Figure 3.2 (a) and (b) show the interior structure and I/O paths of KVM, and LXC respectively. OpenVZ has a similar structure to LXC. In KVM, a guest OS runs on top of a host OS. Hardware devices such as disk drives are virtualized with QEMU device emulator [87]. A disk I/O request is performed in the guest OS but the guest mode has no privilege to access the underlying I/O devices. The I/O requests of the guest



(a) KVM

(b) LXC

Figure 3.2: Interior structure and I/O paths of KVM virtual machine and LXC container.

OS are trapped into the host user-space and passed to the QEMU device emulator. QEMU handles the I/O requests and triggers the real disk I/Os which then are processed by the host OS [9]. These steps add performance overhead on VM's I/Os. In LXC, I/O processes run directly on a host OS without any virtualization overheads from guest mode/user-space context switching and QEMU device emulation. 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, the container provides weaker isolation of performance. Since containers share kernel components like buffer caches and other data structures at the OS-level, the activity inside one container is likely to affect the performance of other containers.

3.3 Disk I/O Control by Cgroup

`Cgroup` uses the block I/O controller to enforce disk I/O control. `Cgroup` supports two policies for controlling disk I/O: 1) I/O throttling and 2) proportional-weight [38]. *I/O throttling* policy is used to set an upper limit for the number of I/O operations performed by specific `group`, it simply

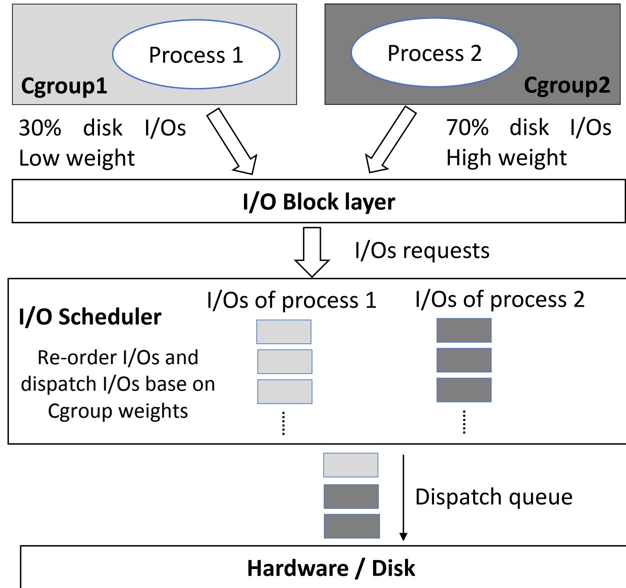


Figure 3.3: Illustration of Cgroup with proportional-weight policy.

caps the maximum usage of I/O request rates [66]. In I/O throttling, a container/VM cannot make use of an idle resource even if there is no contention over the resource; it waists the idle resource. *Proportional-weight* policy allows setting weights to specific `cgroup` . This means that each `cgroup` has a set percentage or share (depending on the weight of the cgroup) of I/O operations [66]. *Proportional-weight* enforces the resource limits only when there is contention over the resource. For example, a container given a 30% share of disk I/Os can consume as many I/O operations as possible if there is no contention over disk I/O. Needless to say, if there is contention over disk I/O, the container can use up to 30% share of disk I/Os. Figure 3.3 illustrates `cgroup` with *Proportional-weight* policy. The policy is implemented in the I/O scheduler layer where I/Os from different processes are queued and dispatched to the underlying disk base on their `cgroup` I/O weights. These weights specify the relative amount of disk time the `cgroup` can use in relation to other `cgroup` [27].

Similarly to old `Cgroup`, the new version of control group `Cgroupv2` uses disk time to apply the proportional-weights I/O control [27]. These time-based weight controls the I/O requests but does not consider the size of I/O

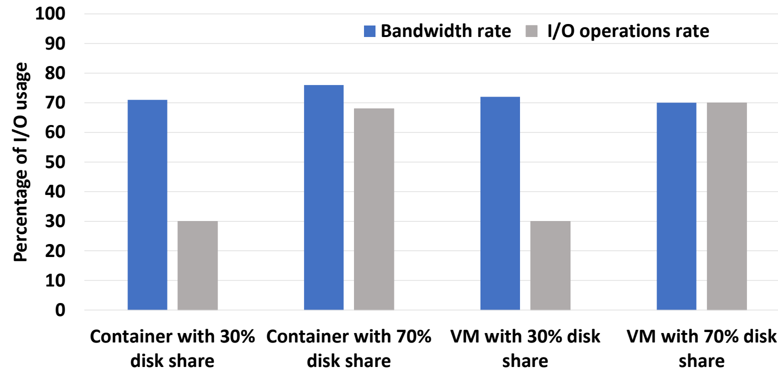


Figure 3.4: Proportional-weight policy in `cgroup` fails to control disk I/O bandwidth. High-weight VM/container (given 70% share of disk I/Os) gets less bandwidth than a low-weight VM/container (given 30% share of disk I/Os)

each request takes. In this 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 a huge amount of disk I/O, it can monopolize the disk bandwidth.

I conduct an experiment consisting of running two collocated containers/VMs and I used Cgroup proportional-weights policy for disk I/O control. The experiment setting is described in section 3.4.1. One container/VM is given a 30% share of disk I/O and the other container/VM is given a 70% share of disk I/O. Both of these two containers/VMs are performing sequential writes using FIO workload but with different I/O sizes. Here, the low-weight container/VM (given 30% share) issues 64KB-disk I/Os while the high-weight container/VM (given 70% share) issues 16KB-disk I/Os. Figure 3.4 shows a container/VM with a low disk I/O weight (given 30% share of I/O request rate) can consume more bandwidth than a container/VM with a high disk I/O weight (given 70% share of I/O request rate). Hence, I suggest that `Cgroup` developers should consider this issue to improve disk I/O control of `Cgroup`.

3.4 Disk I/O Performance and Isolation

To confirm the general belief that containers are better than VMs in performance, this section compares the disk I/O performance and isolation in containers and VMs. Past studies [87, 50, 7, 29, 65] have shown that traditional hypervisors such as Xen, VMware, and KVM have high-performance overheads. From the viewpoint of performance isolation, VMs promise to provide better isolation than containers because each VM runs a stand-alone OS without sharing any kernel components. The I/O performance and isolation are very relevant. The failure in achieving I/O isolation causes performance degradation between collocated containers/VMs. Performance isolation is a scheme of resource management that provides performance guarantees to containers/VMs [78]. If multiple containers/VMs are competing for a particular resource, each container/VM cannot use the resource beyond the share allocated to it. If there is no contention over a resource, a container/VM should be able to consume the resource as much as it demands, which leads to more efficient use of resources in clouds. For example, if a VM/container is given a 30% share of disk I/O, it can consume as much disk I/Os as it demands if there is no contention over disk I/O. But if other containers/VMs are competing for disk I/Os, it can consume the only 30% of disk I/Os.

3.4.1 Experimental Setup

The experimental environment 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 the 4.18.0-25-generic kernel is installed. SAS hard disk of 1TB is formatted with ext4 file system with the default journaling mode; i.e., only the metadata are journaled. Disk resource control is enforced through `Cgroup's` proportional-weight policy. The new version of control group "`Cgroupv2`" [27] is used with LXC and KVM. LXC 3.0.4 is used for Linux containers while OpenVZ 7.0.10 with its corresponding kernel and resources control is used for OpenVZ container. KVM-qemu 2.11.1 is installed on the host and the guest environments are the same as the host. Each VM is allocated one virtual CPU that pins to a one CPU core and 1GB RAM with a raw disk partition allocated as secondary storage. The same

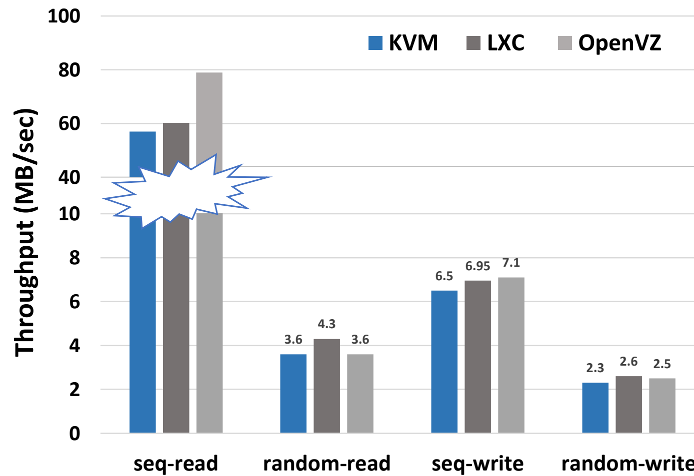


Figure 3.5: Disk I/O throughput in KVM, LXC, and OpenVZ. Containers beat VM in all I/O workloads.

CPU and memory configuration is applied to the containers.

3.4.2 Results

To examine the I/O performance, flexible I/O (FIO) benchmark [15] is used to generate four types of I/O workloads: 16KB random read/write and 64KB sequential read/write. Direct I/O mode is turned on to bypass the adverse effect of the buffer cache. Figure 3.5 shows the I/O throughput of KVM, LXC, and OpenVZ. In this experiment, one instance of a container or a VM is running. As shown from Figure. 3.5, both LXC and OpenVZ outperform KVM in all I/O workloads.

LXC and OpenVZ show better performance than KVM in consolidation cases when two VMs/containers are consolidated on a single physical machine. The one VM/container is given a 30% share of disk I/Os and the other is given a 70% share. Figure 3.6 shows the throughput of one container/VM (the one with 70% disk share) when the same I/O workloads in Figure 3.5 are run in two VMs/containers. As this figure shows LXC and OpenVZ beat KVM in all cases.

Figure 3.7 (a), (b), and (c) show the performance isolation of KVM, LXC, and OpenVZ respectively. In this experiment, two VMs/containers

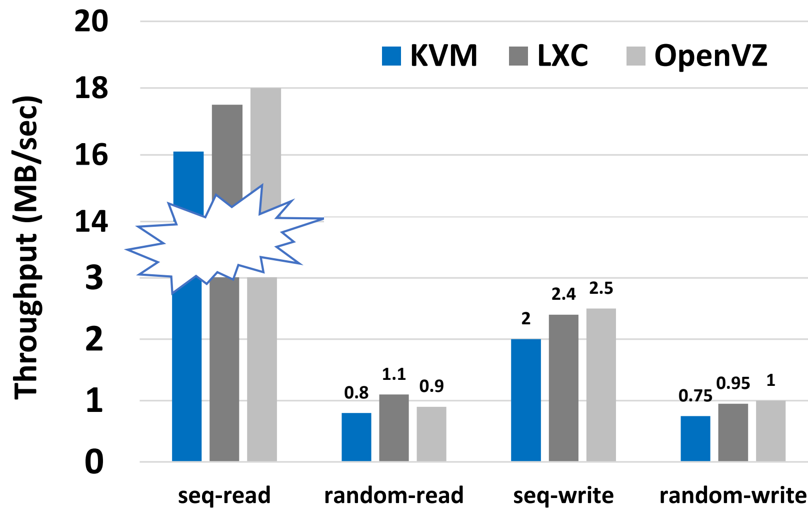
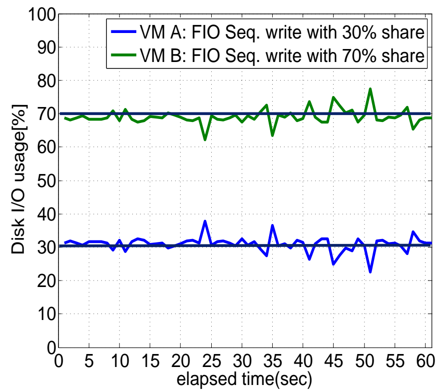


Figure 3.6: Disk I/O throughput in KVM, LXC, and OpenVZ in consolidation case when two I/O workloads are collocated together. Containers outperform VMs in all workloads

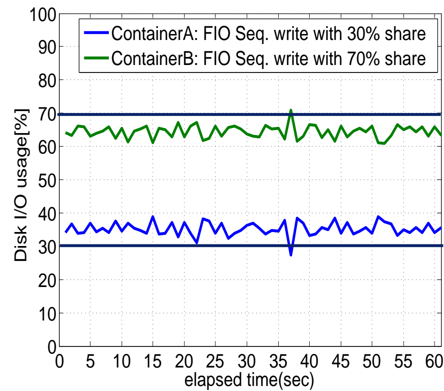
are launched to run the sequential write workload. The one VM/container is given a 30% share of disk I/Os and the other is given 70% share. Figure 3.7 shows that all of KVM, LXC, and OpenVZ respect the shares of disk I/O gracefully. The VM/container with 30% share consumes around 30% share, and the VM/container with 70% share consumes around 70% share. LXC and OpenVZ containers enforce the resource limit successfully and show comparable performance isolation to VM.

3.5 Summary

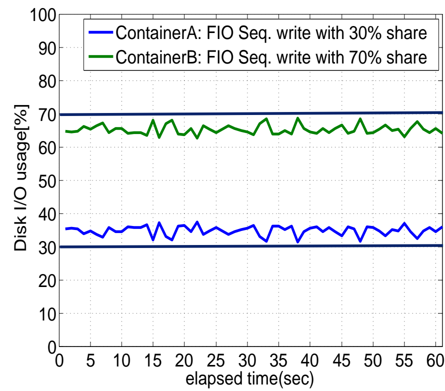
The background of the container and virtual machine architectures are presented in this chapter. The overview of KVM, LXC, and OpenVZ systems is presented as well. Disk I/O in these systems is explained and disk I/O control by `Cgroup` is examined. Disk I/O performance and isolation are compared in KVM, LXC, and OpenVZ. The results show that containers outperform virtual machines in disk I/O performance and provide identical I/O isolation to that of the virtual machine. Based on these results, it is expected that DBMS performance and isolation will be better in the container than in the



(a) Disk I/O usage in KVM



(b) Disk I/O usage in LXC



(c) Disk I/O usage in OpenVZ

Figure 3.7: Performance isolation in KVM, LXC, and OpenVZ where the one VM/container is given 30% share of disk I/O and the other is given 70%. All VMs/containers respect the given shares of disk I/O.

virtual machine as it will be compared in the next chapter.

Chapter 4

DBMS Performance and Isolation

The objective of this chapter is to investigate DBMS performance and isolation in the container and virtual machine. The chapter highlights the motivation that despite containers are preferred over virtual machines because of virtualization overhead, the trade-offs between performance and isolation in containers and virtual machines are still unclear. From the results obtained in Chapter 3, it is expected that containers are more appropriate than virtual machines for consolidating DBMSes. Surprisingly, the results show that DBMS performance is better in virtual machines than in containers. Furthermore, disk I/O isolation is very terrible when consolidating DBMSes in containers. The analysis reveals that file-system journaling in containers degrades DBMS performance and violates disk I/O isolation. Our investigation identifies the underlying causes behind file-system journaling problems in containers.

4.1 MySQL Performance and Isolation

Based on results in Chapter 3, it is expected that LXC and OpenVZ are more appropriate than KVM for consolidating DBMSes. LXC and OpenVZ outperform KVM in disk I/O throughput and show I/O isolation comparable with KVM. This section shows the experiments that have conducted to

confirm these results. Surprisingly, the experimental results are contrary to the expectation. Containers are not appropriate for DBMS consolidation. Containers are worse than VMs in DBMS performance and violate the I/O isolation even when the resource limit is imposed by `Cgroup`.

4.1.1 Experimental Setup

The same experimental environment in Chapter 3, Section 3.4.1 is used. 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 [36] generates workloads, which run in a separated machine connected via Cisco 1Gbit 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.

4.1.2 Results

The performance and isolation of MySQL are compared in LXC, OpenVZ, and KVM. In the experiment, two VMs/containers are launched, each executes MySQL workload. Figure 4.1 shows MySQL throughput in consolidation case (the throughput of one container/VM). For comparison, the figure shows MySQL throughput in a standalone case, where one VM/container is launched. Out of expectation VM outperforms containers in consolidation cases when two VMs/containers are collocated together on the same machine. LXC and OpenVZ performance is 22% and 25% worse than KVM in MySQL throughput respectively.

Figure. 4.2 compares the performance isolation of MySQL between VMs and containers in the previous experiment. The disk I/O control is imposed by `Cgroup`. One VM/container is given a 30% share of disk I/O and the other is given a 70% share. The X-axis shows the elapsed time and the Y-

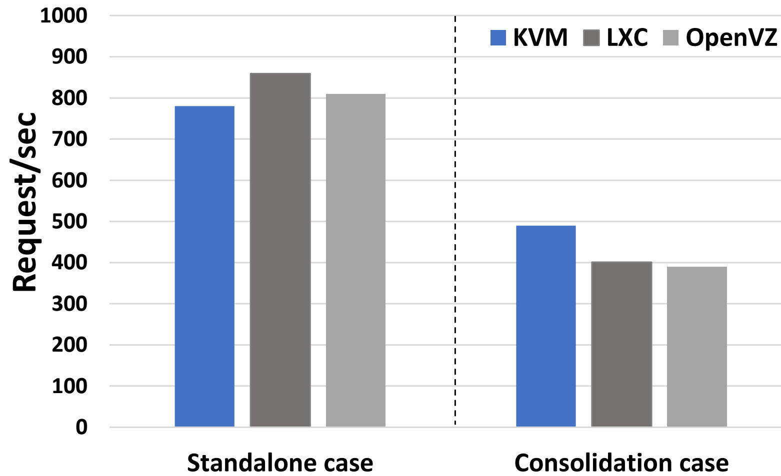
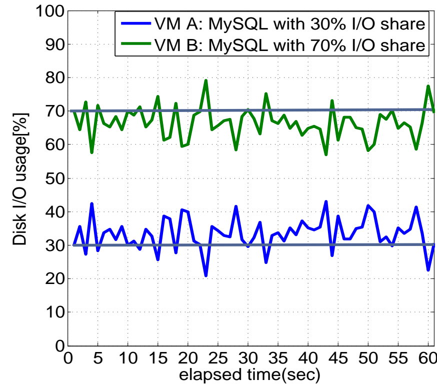


Figure 4.1: MySQL throughput in KVM, LXC, and OpenVZ. The graph shows 1) standalone, 2) collocated with other VM/container. Surprisingly KVM outperforms LXC and OpenVZ.

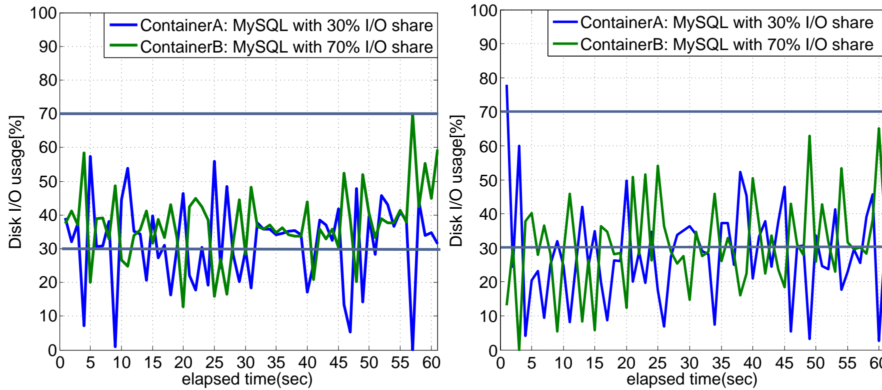
axis shows the percentage of the disk I/Os consumed by each VM/container. Figure (a) indicates that KVM respects the resource limit because the VM with 30% share consumes around 30% of disk I/Os and the other VM with 70% share does around 70%. On the other hand, Figure (b) and Figure (c) show disk I/O isolation in LXC and OpenVZ respectively. The figures show a clear disk I/O contention between collocated containers. The disk I/Os consumption of the container with 30% share and container with 70% share fluctuates terribly from 0% to 55%. This indicates that containers violate the resource control in MySQL workload although they previously show perfect isolation comparable to that of VM in FIO workloads.

4.2 Analyzing DBMS Performance and Isolation

This section provides a quantitative analysis of DBMS performance and isolation in containers and VMs. To understand the results of MySQL in containers that are contrary to the expectations, disk I/O performance and



(a) Disk I/O usage in KVM



(b) Disk I/O usage in LXC (c) Disk I/O usage in OpenVZ

Figure 4.2: Performance Isolation in KVM, LXC, and OpenVZ. Containers show a terrible disk I/O isolation with MySQL database.

isolation are investigated. Depending on the user case, DBMS can become an update-intensive application, it performs a lot of data write and change in a database. DBMS invokes `fsync` calls at a high rate to ensure updates are written to disk. I suggest `fsync` calls that are invoked at a high rate slow down disk I/O and affect DBMS performance in containers. The `fsync` is closely related to file-system journaling. Since containers share the journaling mechanism unlike VMs, journaling activities are serialized and bundled with each other, resulting in inferior I/O performance and isolation. I identify the underlying causes behind file-system journaling problems in containers. The results confirm the influence of journaling problems on MySQL performance

and isolation.

4.2.1 Investigating the effect of Fsync

MySQL performance degrades in LXC and OpenVZ even though the performance is better than KVM in the FIO workloads. A major difference in the MySQL and FIO workloads is that the MySQL invokes `fsync` frequently to ensure that all updates are written to the final destination on disk. The `fsync` is a system call transfers (flushes) modified data and metadata (information associated with data) in buffer cache to the disk device so that all data can be retrieved even if the system crashes or is rebooted [41]. The call blocks until the device reported that data transfer has completed [41]. In the MySQL workload, `fsync` is invoked at the rate of 27 times/sec in KVM, and 14 and 12 times/sec in LXC and OpenVZ respectively. On the other hand, the FIO workload in previous experiments does not invoke `fsync` explicitly. In FIO workload, updates are flushed to disk at the rate of 0.2 times/sec (every 5 seconds).

To verify that the high rate of `fsync` has a significant impact on I/O performance, I 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 4.3 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 and OpenVZ outperform 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 and OpenVZ. The throughput in LXC and OpenVZ degrade when the collocated workload invokes `fsync` more frequently in low-fsync workload. The throughput in LXC and OpenVZ degrade more when collocated with high-fsync workload.

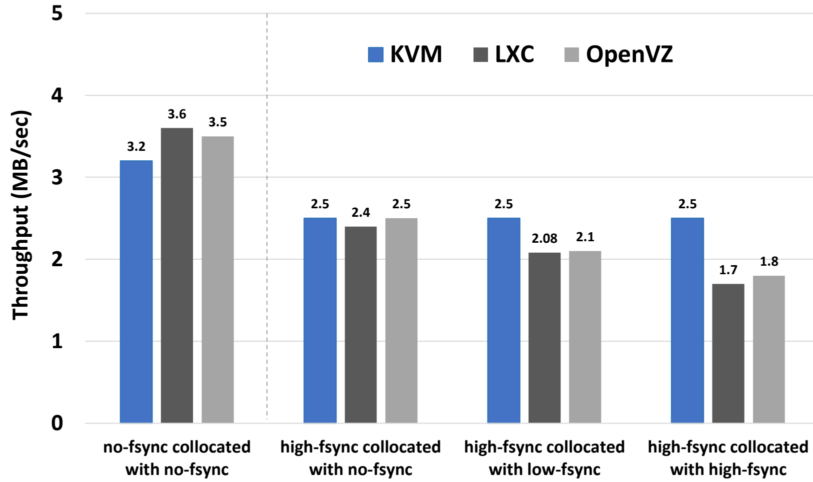


Figure 4.3: Throughput of disk I/O in KVM, LXC, and OpenVZ. A container/VM is running high-fsync workload and is collocated with either 1) no-, 2) low-, or 3) high-fsync workload. LXC and OpenVZ performance degrades as fsync intensity increases in the collocated workload.

While KVM shows a stable and constant throughput (around 2.5 MB/sec) regardless of collocated workload. These results point out that a high rate of `fsync` calls are relevant in disk I/O performance in the container but not in VM. The `fsync` is closely related to file-system journaling. It is the sharing of file-system journaling that degrade I/O performance as will be explained in the next sections.

4.2.2 File-system Journaling

Modern file systems use journaling [63, 60, 30, 13] to keep the file-system consistency and for data recovery 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

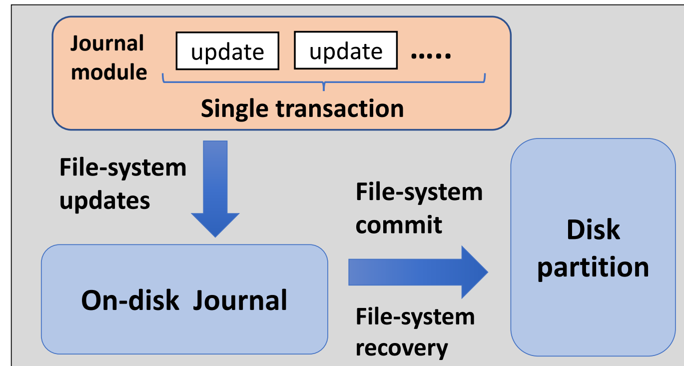


Figure 4.4: A typical journaling file-system.

partially updated, the file system becomes inconsistent. In the worst case, the user cannot access it anymore. Journaling is write-ahead logging. Before updating the file system, it logs the write operations to an on-disk region called *journal*.

Figure 4.4 shows a typical journaling file-system. The file-system uses *journal* to log file-system updates not yet committed to disk to avoid metadata corruption [30]. Metadata refers to the managing structures for data on a disk and it is important to keep file-system consistency. Metadata represents directory and file creation, removal, growing, truncating, and so on [30]. In case of a system crash or power failure, before updates are committed to disk, the file-system recovers from inconsistency by re-doing the logs in the journal.

A kernel component responsible for handling the journaling operation is called a *journal module* as shown in Figure 4.4. Only a single journaling module can run at a time [63]. If there are multiple journaling modules, they cause a race condition and file-system inconsistency because they may access sensitive on-disk data structures concurrently. 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), JBD2 (Journaling Block Device) is responsible for journaling. The JBD2 groups file-system updates from multiple processes in that single compound transaction. After logging the updates, the transaction is committed to the file system and then

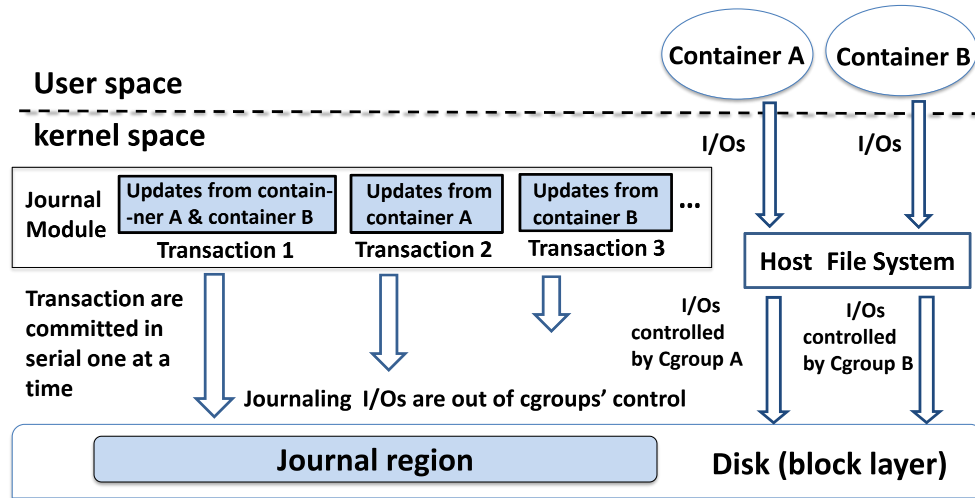


Figure 4.5: File-system journaling in container virtualization.

removed from the journal. The transactions are committed in serial one at a time, periodically (every 5 sec by default) or every time `fsync` is invoked [63].

4.2.3 Journaling Problems in Containers

Containers share the same kernel components of the host like the file system and journaling module. The sharing of a journaling module in containers causes a negative impact on disk I/O performance and isolation. The journaling module causes performance dependencies across collocated containers and interferes with disk I/O control of `cgroup`. These problems result in the violation of performance isolation and degrade I/O performance in containers.

Performance Dependencies Through Journaling:

Figure 4.5 illustrates file-system journaling in containers. In the figure, two containers share the single journaling module, and thus a single transaction bundle updates from the two containers. When one container invokes `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. Ideally, invoking `fsync` in a container should commit only

Table 4.1: 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	OpenVZ
No-fsync	18.58ms	33.81ms	31.78ms
Low-fsync	18.94ms	38.86ms	39.81ms
High-fsync	18.27ms	44.09ms	50.41ms

the updates belonging to that particular container. Because of the bundled transaction, calling `fsync` causes unrelated updates to be flushed as well. This dependency can violate the performance isolation because one container can degrade the performance of another by invoking many `fsync` calls. A performance dependency can be caused even if a single transaction solely contains updates from one container. The 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). Suppose transactions 2 and 3 contain updates solely from container A and B, respectively. Transactions 2 and 3 cannot be committed in parallel because transactions are serialized in the journaling module to keep the order of updates. Therefore, if two containers invoke `fsync` at the same time, container A, for example, has to wait until transaction 2 is committed.

To confirm the above observation, I measure the `fsync` latency of KVM, LXC, and OpenVZ in the previous experiment in Section 4.2.1. Table 4.1 shows the `fsync` latency of the high-fsync workload when it collocated with either 1) no-, 2) low-, or 3) high-fsync workload. The `fsync`latency increases in LXC and OpenVZ as `fsync` intensity increases in the collocated workload. This increase of `fsync` latency comes from `fsyncs` contention between containers due to transactions serialization. The `fsync` is blocked until the journaling transaction is committed. This increases `fsync` latency which degrades the I/O performance of the high-fsync workload. For KVM, the average latency is about 18.5ms regardless of the collocated workloads. KVM avoids the journaling problem because each VM has its own kernel and journaling module as shown in Figure 3.2 (a).

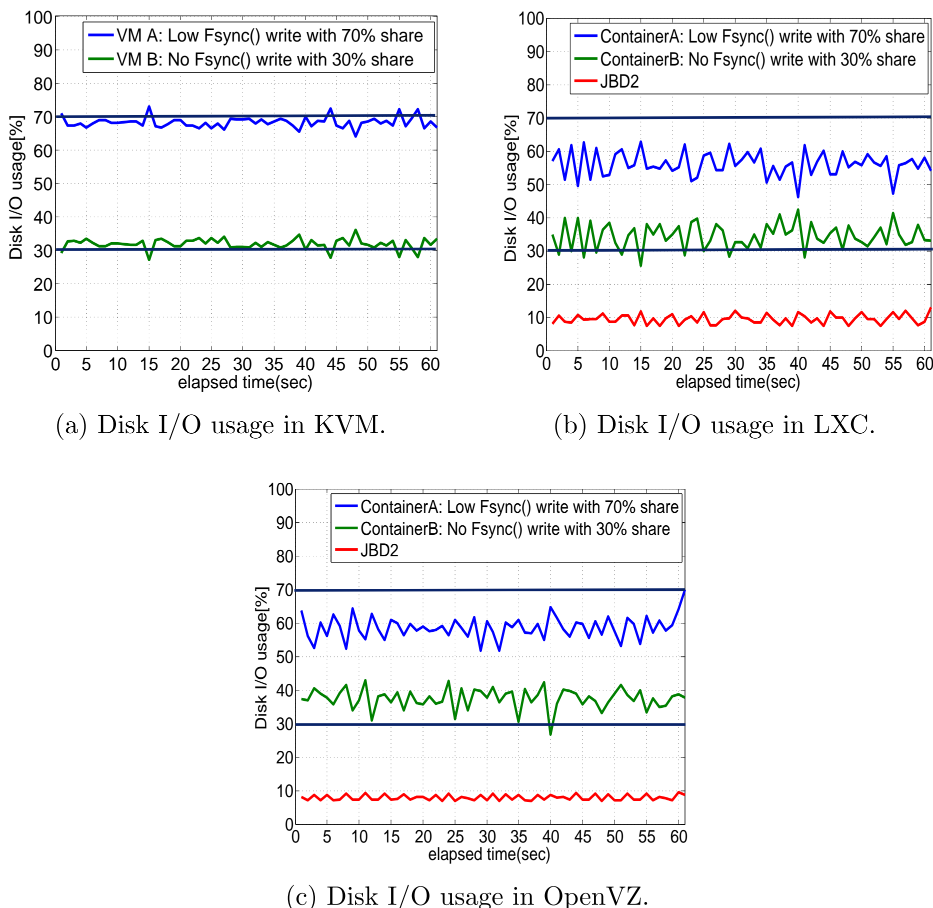
Figure 4.6 shows the usage of disk I/O in KVM, LXC, and OpenVZ when

the low-fsync workload is collocated with the no-fsync workload. The low-fsync container is given 70% disk share and the other is given 30% disk share. The figure also shows the usage of disk I/O consumed by *JDB2*, the journaling module for Linux Ext4. Performance isolation is gracefully respected in KVM whereas in LXC and OpenVZ are slightly violated. When the collocating workload is changed to the high-fsync, the journaling initiated by the high-fsync has a significant impact on the performance isolation of the low-fsync. Figure 4.7 shows that LXC and OpenVZ violate the performance isolation, whereas KVM respects it gracefully. The low-fsync container in LXC is used around 50–30% share while in OpenVZ, is used 50% share. This is instead of 60% share when the low-fsync container was collocated with no-fsync workload. Since the low-fsync container waits for slow `fsync` calls, it can not fully utilize the given share of disk I/O and `cgroup` judges the low-fsync container is not disk-intensive. As a result, the low-fsync container hands over its share to the high-fsync container. This indicates that the journaling activity of each container adversary impacts the performance isolation of the other.

Impact of Journaling on disk I/O control:

The sharing of journaling modules among containers interferes with the disk I/O control of `cgroup`. As shown in Figure 4.5, 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 invokes `fsync` calls frequently, its corresponding journaling I/Os are not accounted for the 30% disk I/O share. This results in the violation of performance isolation between containers A and B; container B gets a higher disk I/O share than 30%.

Figure 4.7 indicates that I/O operations from a journaling module are overlooked in LXC and OpenVZ. Disk I/O consumed by JBD2 increases from 10% to 28% in LXC and from 9% to 21% in OpenVZ when the collocated workload is changed from the no-fsync to the high-fsync. This increase is caused by the high rate of `fsync` calls in the high-fsync workload and thus



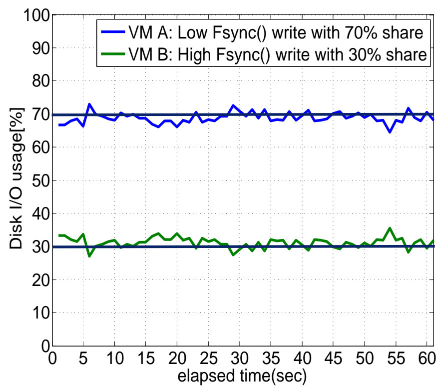
(a) Disk I/O usage in KVM.

(b) Disk I/O usage in LXC.

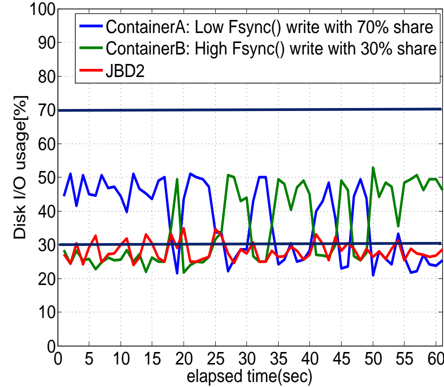
(c) Disk I/O usage in OpenVZ.

Figure 4.6: Disk I/O isolation in KVM, LXC, and OpenVZ when the low-fsync is collocated with no-fsync wrokload.

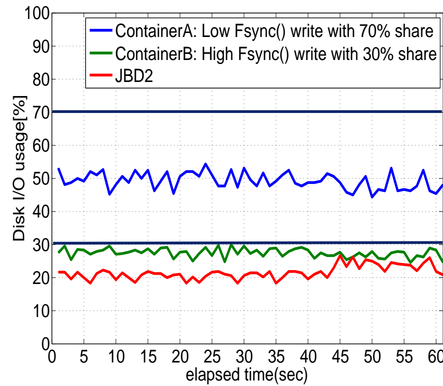
it should be accounted for the high-fsync container. As shown in Fig. 4.7, the high-fsync container, which is given 30% share, consumes 30% share of disk I/O without taking the journaling I/Os into consideration. Since the journaling I/O increases by 18% (= 28% − 10%) in LXC and by 12% (= 21% − 9%) in OpenVZ when the workload is changed to the high-fsync, this 18% and 12% consumption of disk I/O should be accounted for the containers. For example in the case of LXC, the high-fsync container should consume up to 12% (= 30% − 18%) share in total. While in OpenVZ, the high-fsync container should consume up to 18% (= 30%−12%) share in total. On the other hand, KVM divides the disk I/O to 70% and 30% between two



(a) Disk I/O usage in KVM.



(b) Disk I/O usage in LXC.



(c) Disk I/O usage in OpenVZ.

Figure 4.7: Disk I/O isolation in KVM, LXC, and OpenVZ when the low-fsync is collocated with high-fsync wrokload.

VMs perfectly. In the case of KVM, `cgroup` monitors all I/O operations from each VM which contains those from the journaling module that runs inside each VM.

4.2.4 Journaling Influence on MySQL Performance

To confirm that MySQL performance is affected by journaling problems, the throughput of MySQL is measured when it is collocated with a VM/container running 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%

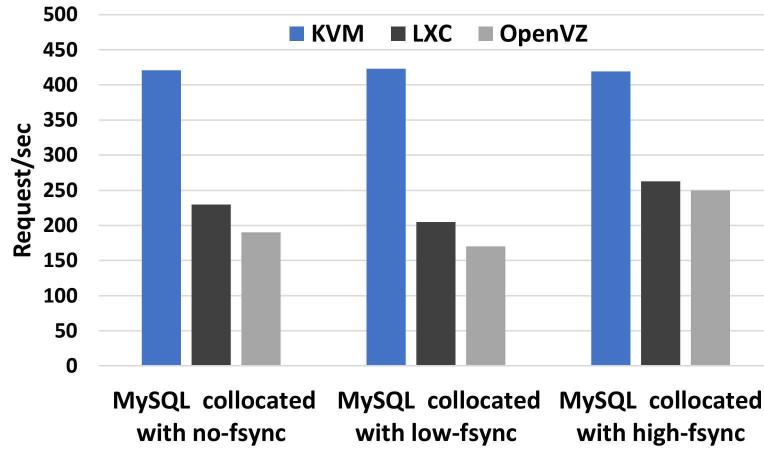


Figure 4.8: MySQL throughput in KVM, LXC, and openVZ with either 1) no-, 2) low-, or 3) high-fsync workloads. MySQL is given 70% share of disk I/O.

Table 4.2: Average fsync latency of MySQL when collocated with no-, low-, high-fsync workload. MySQL VM/container is given 70% share.

Collocated workload	KVM	LXC	OpenVZ
No-fsync	(26.87ms)	(72.51ms)	(120.23ms)
Low-fsync	(27.28ms)	(85.09ms)	(160.651ms)
High-fsync	(26.12ms)	(74.08ms)	(96.20ms)

share. Figure 4.8 shows MySQL throughput in KVM, LXC, and OpenVZ.

Since each VM has its own journaling module and can avoid all the journal-related problems, MySQL throughput in KVM is almost constant (around 425 requests/sec) in all the cases. By avoiding the interference from the sharing of a journaling module, KVM always shows better throughput than LXC and OpenVZ. on the other hand, the throughput in LXC and OpenVZ degrades when the collocated workload changed from no-fsync to low-fsync. Since the low-fsync workload invokes `fsync` more frequently than no-fsync, the `fsync` latency increases in MySQL and the throughput becomes worse. Table 4.2 shows `fsync` latency of MySQL in LXC and OpenVZ are increased when the collocated workload changed from no-fsync to low-fsync.

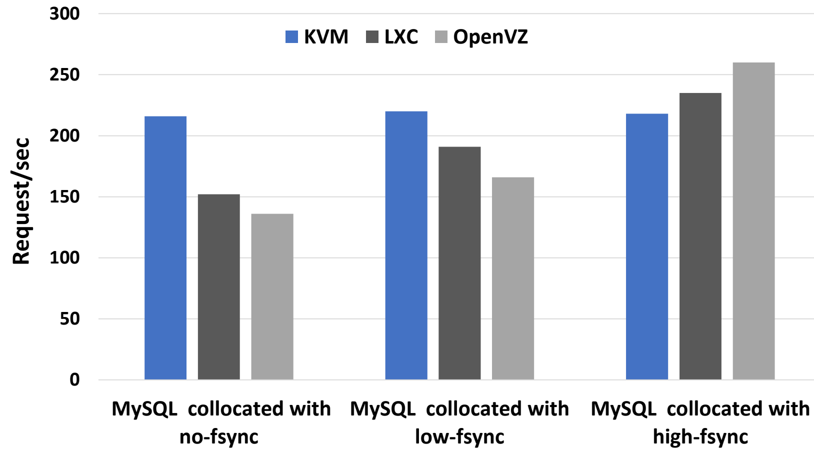


Figure 4.9: MySQL throughput in KVM, LXC, and openVZ with either 1) no-, 2) low-, or 3) high-fsync workloads. MySQL is given 30% share of disk I/O.

Table 4.3: Average fsync latency of MySQL when collocated with no-, low-, high-fsync workload. MySQL VM/container is given 30% share.

Collocated workload	KVM	LXC	OpenVZ
No-fsync	(24.52ms)	(116.4ms)	(148.3ms)
Low-fsync	(24.82ms)	(154.2ms)	(172.5ms)
High-fsync	(24.44ms)	(86.5ms)	(93.1ms)

When the collocated workload is changed from low-fsync to high-fsync, MySQL throughput improves in LXC and OpenVZ as shown in Figure 4.8. This looks strange because the high-fsync invokes more `fsync` calls than the low-fsync. Because high-fsync container workload invokes more `fsync`, it causes more journaling updates. Hence, MySQL updates are committed together with the updates from the high-fsync in the same transaction due to the sharing of the journaling module. From Table 4.2 the `fsync` latency of LXC and OpenVZ is reduced to 74.08ms and 96.2ms respectively with the collocation of high-fsync workload. The reduced latency of `fsync` results in improving the throughput of MySQL.

Figure 4.9 shows MySQL throughput when the shares of disk I/O are swap. MySQL VM/container is given a 30% share and the other is given

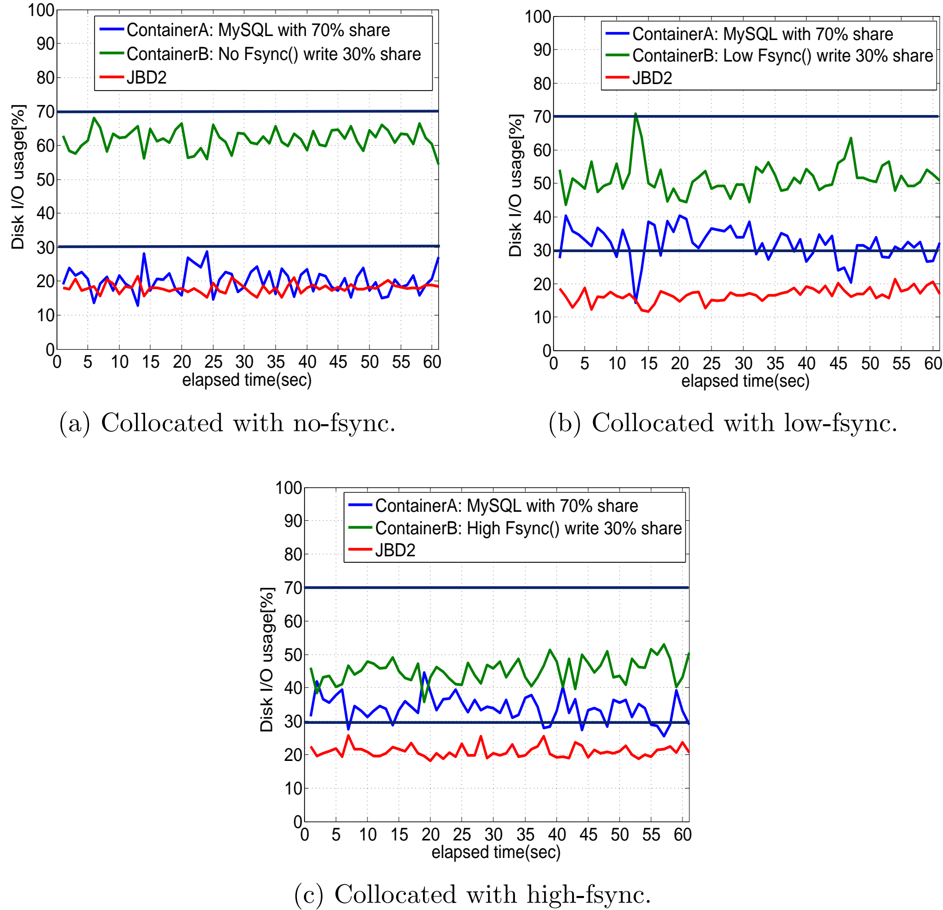
a 70% share. The throughput in LXC and OpenVZ degrades also when the collocated workload changed from no-fsync to low-fsync. Again, when the collocated workload is changed from low-fsync to high-fsync, MySQL throughput improves in LXC and OpenVZ. Table 4.3 shows the `fsync` latency in LXC and OpenVZ which confirm these similar results. On the other hand, MySQL throughput in KVM is almost constant again around 220 requests/sec. Since the disk I/O share that MySQL VM/container can use is reduced to 30% from 70%, the throughput becomes smaller than the previous experiment. In this setting, LXC and OpenVZ outperform KVM in the high-fsync workloads although they are beaten in the no-fsync and low-fsync workloads. This mystery is closely related to performance isolation and thus discussed in detail in Section 4.2.5.

4.2.5 Journaling Influence on MySQL Isolation

MySQL performance isolation in LXC and OpenVZ is terrible as shown in Section 4.1.2. To confirm that MySQL isolation in containers is violated due to the journaling problems, the disk I/O usage of MySQL is compared when it is collocated with no-, low- and high-fsync workloads. MySQL VM/container is given a 70% disk share while the other collocated VM/container is given a 30% share.

Figure 4.10 and Figure 4.11 shows the results in LXC and OpenVZ respectively. Since MySQL is update-intensive, the I/O usage of the journaling is around 20% in LXC and OpenVZ 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. MySQL container is given around 20% share in LXC and OpenVZ instead of 70% disk share.

When the collocated workload is changed to the low- or high-fsync workload, the MySQL container consumes more disk I/O share. This is because the low- or high-fsync container competes for journaling with the MySQL container. The `fsync` calls from the low- or high-fsync container are suspended in the journaling module to the contention. Hence, the I/Os from low- or high-fsync containers are reduced and `cgroup` allocates more disk



(a) Collocated with no-fsync.

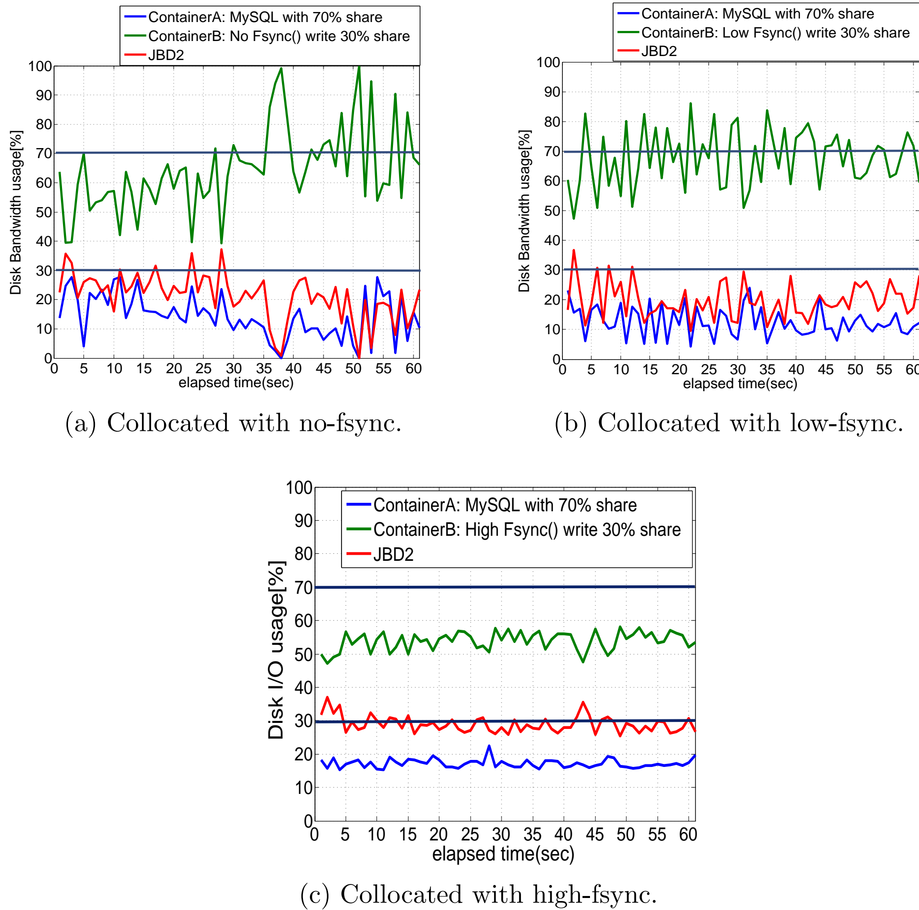
(b) Collocated with low-fsync.

(c) Collocated with high-fsync.

Figure 4.10: Disk I/O usage in MySQL in LXC . Collocated with no-, low-, high-fsync workloads. MySQL is given 70% share.

I/Os to the MySQL container. This contributes to improving MySQL performance when it collocates with high-fsync container as shown in Figure 4.8.

Figure 4.13 and Figure 4.14 shows the disk I/O usage in LXC and OpenVZ, where a MySQL container is given 30% share. As shown in the figures, the MySQL container is given less than 30% share when collocated with the no-fsync workload, but it is given more share when it collocates with the low- or high-fsync workloads. If a fsync-intensive workload is collocated with MySQL, it slows down due to severe contention over the journaling with MySQL and hands over disk I/O share to MySQL container. This explains the performance mystery in Figure 4.9. The performance of MySQL in LXC



(a) Collocated with no-fsync.

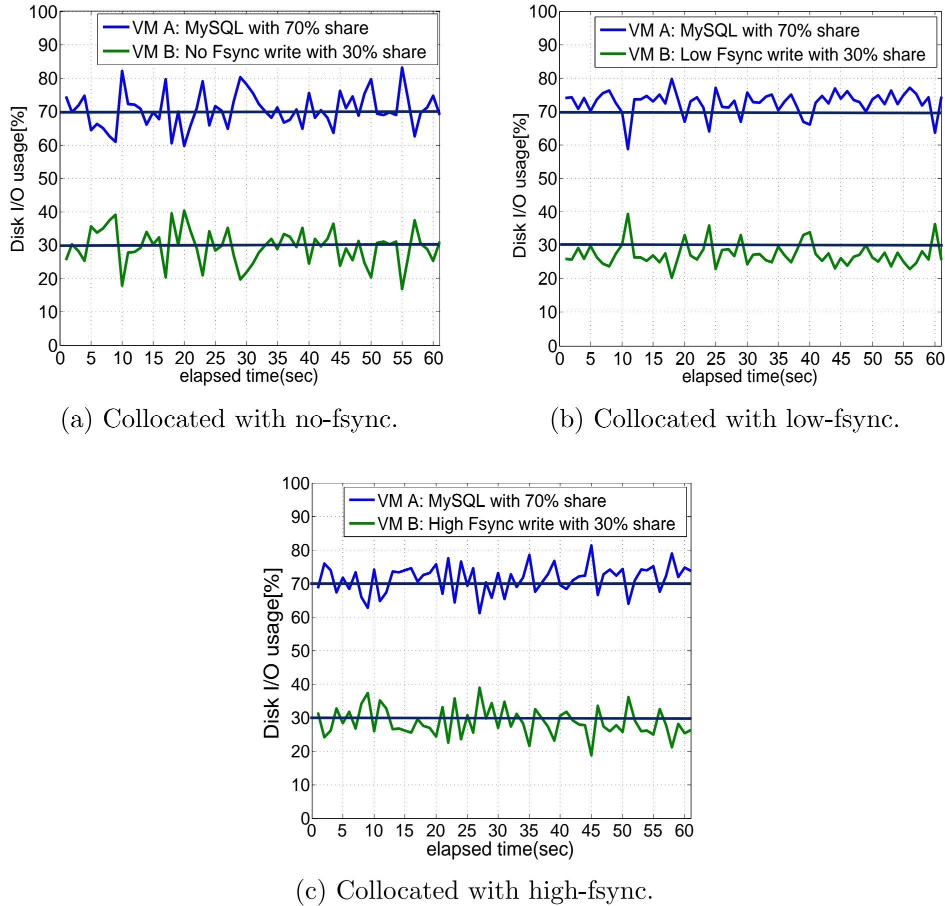
(b) Collocated with low-fsync.

(c) Collocated with high-fsync.

Figure 4.11: Disk I/O usage in MySQL in OpenVZ. Collocated with no-, low-, high-fsync workloads. MySQL is given 70% share.

and OpenVZ improves when MySQL is collocated with more fsync-intensive workloads. It is the violation of performance isolation that improves MySQL throughput in LXC and OpenVZ. This also explains why MySQL container with 30% share performs better than KVM when collocated with high-fsync workload as shown in Figure 4.9. In Figure 4.13 (c) and Figure 4.14 (c), MySQL container consumes 30% share without considering its journaling I/Os from JBD2 which are overlooked by `cgroup`. Hence, MySQL container consumes more than 30% of disk I/O share in total. While in KVM, MySQL VM consumes 30% share with its journaling I/Os in total.

KVM gracefully respects the I/O control of `cgroup` in the both setting.



(a) Collocated with no-fsync.

(b) Collocated with low-fsync.

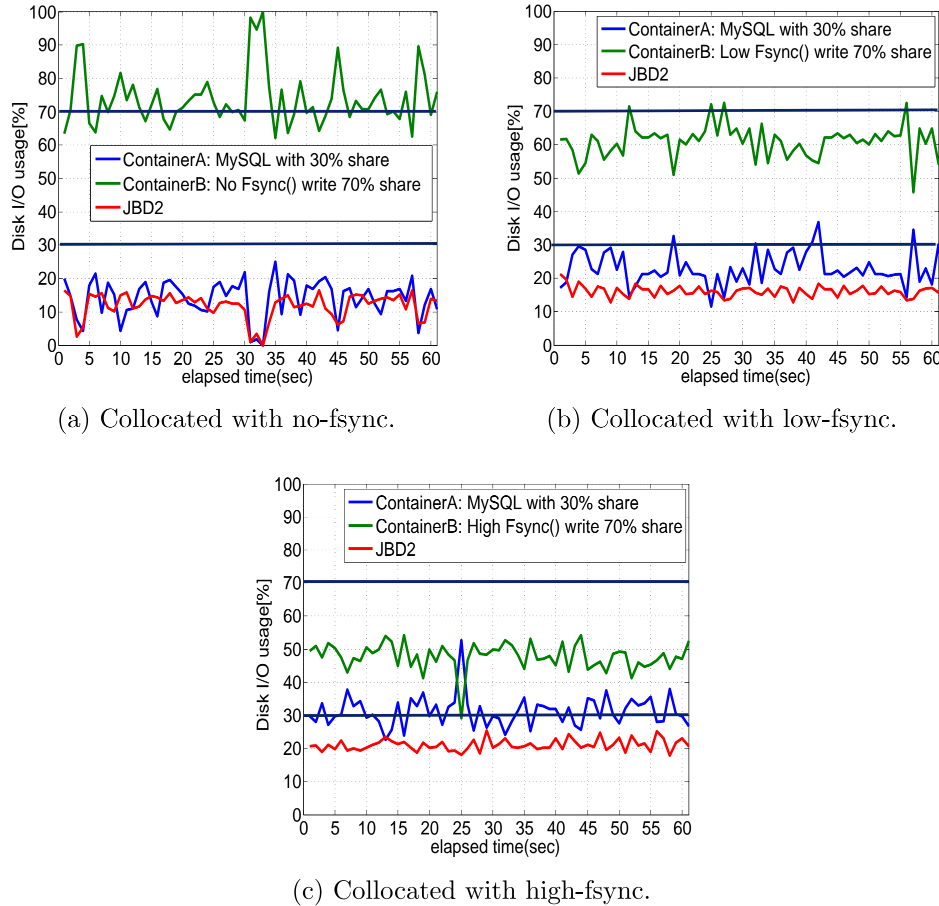
(c) Collocated with high-fsync.

Figure 4.12: Disk I/O usage in MySQL in KVM. Collocated with no-, low-, high-fsync workloads. MySQL is given 70% share.

KVM divides the disk I/O into 30% and 70% share between VMs as shown in Figure 4.12 and Figure 4.15.

4.3 Summary

This chapter presents a performance evaluation of DBMS consolidation in containers and VMs. The performance and performance isolation of MySQL is investigated in LXC and OpenVZ and compared with that of KVM. Our key finding is that KVM outperforms LXC and OpenVZ in both I/O performance and isolation in DBMS. This finding is contrary to the general



(a) Collocated with no-fsync.

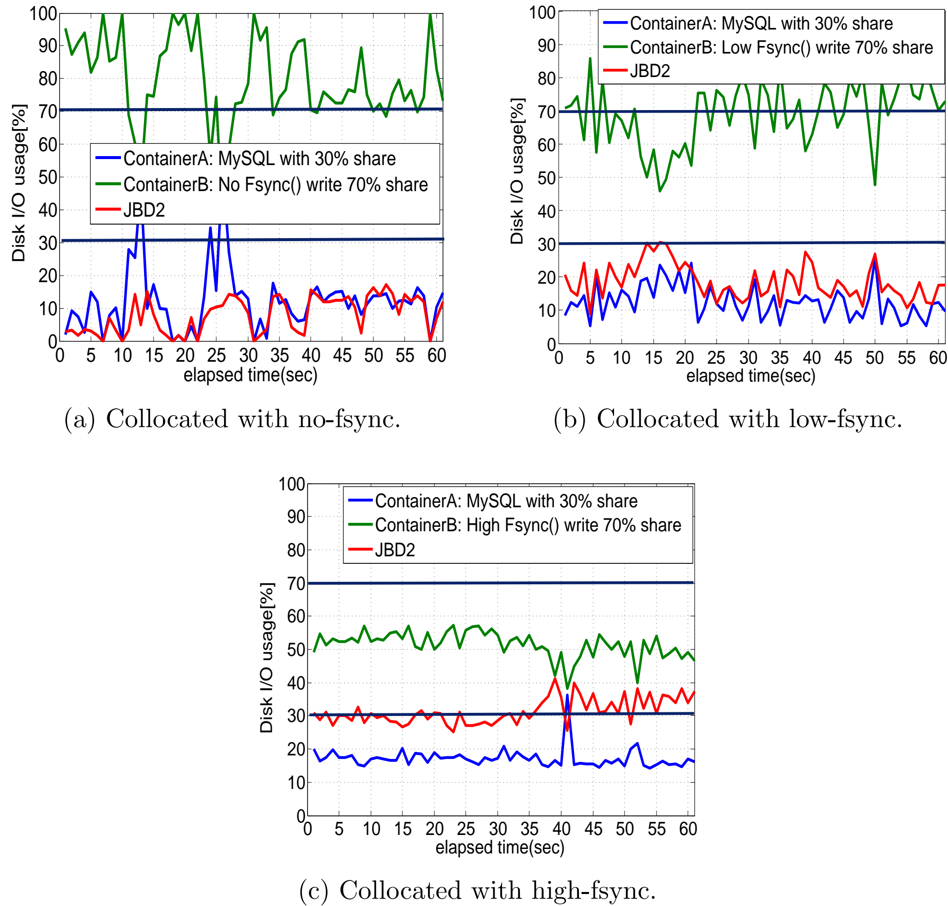
(b) Collocated with low-fsync.

(c) Collocated with high-fsync.

Figure 4.13: Disk I/O usage in MySQL in LXC . Collocated with no-, low-, high-fsync workloads. MySQL is given 30% share.

belief that container is always better than VM in performance because of no virtualization overheads. our results show KVM beats LXC and OpenVZ in MySQL performance by up to 2X and 2.4X respectively. Furthermore, LXC and OpenVZ fail to achieve performance isolation among containers although the resource control mechanism of `cgroup` enforces disk I/O control.

The analysis reveals that file-system journaling is the root cause of the poor I/O performance and isolation in the container. Since a journaling module is shared inherently among containers, it becomes a bottleneck in performance and causes a serious problem when running update-intensive applications such as DBMS. The journaling degrades I/O performance in



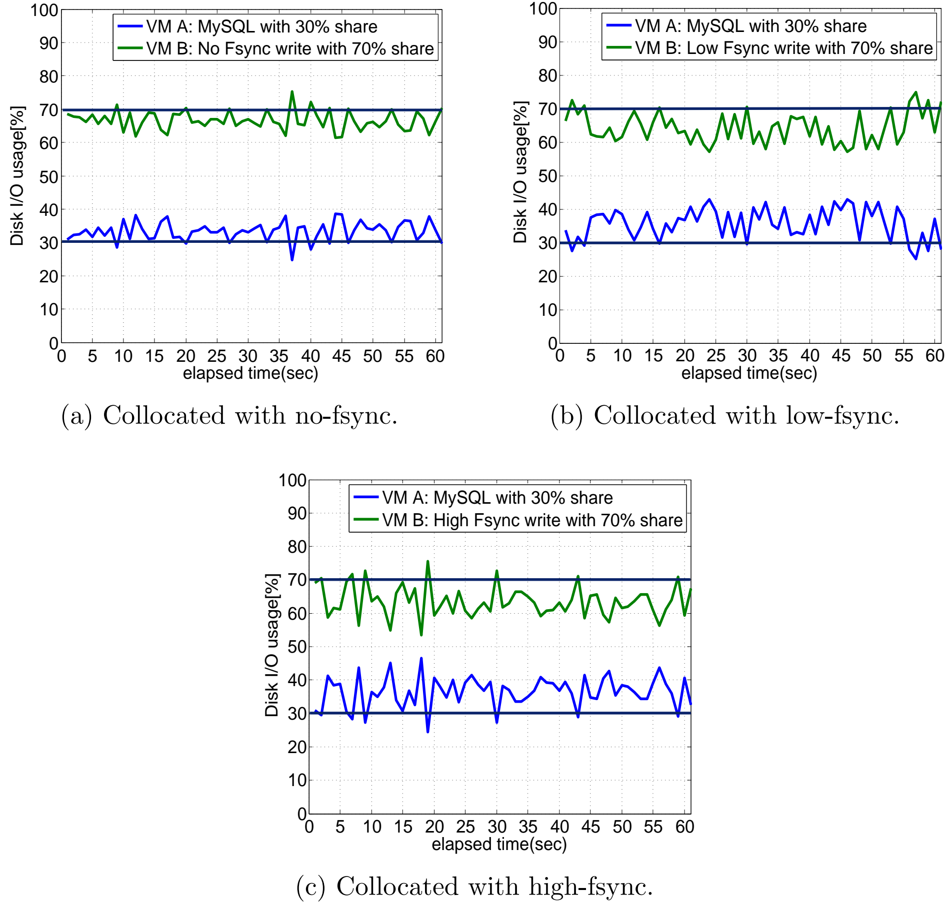
(a) Collocated with no-fsync.

(b) Collocated with low-fsync.

(c) Collocated with high-fsync.

Figure 4.14: Disk I/O usage in MySQL in OpenVZ. Collocated with no-, low-, high-fsync workloads. MySQL is given 30% share.

containers because of the following reasons. The shared journaling module causes performance dependency among containers; For optimization, the file-systems use journaling with transactions. A journaling module batches updates from multiple containers into a single 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 committed. 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 contain-



(a) Collocated with no-fsync.

(b) Collocated with low-fsync.

(c) Collocated with high-fsync.

Figure 4.15: Disk I/O usage in MySQL in KVM. Collocated with no-, low-, high-fsync workloads. MySQL is given 30% share.

ers are suspended because of the lack of parallelism.

Also, file-system journaling in containers interferes with disk I/O control of `cgroup`. Since a journaling module is running outside of controlled containers, I/O operations from the module are not accounted for containers that initiate them. This results in an inaccurate division of disk I/Os between containers. In contrast, KVM avoids journal-related problems because each VM has its own journaling module due to the complete separation of kernel components.

Chapter 5

Alleviating Journaling Problems in Containers

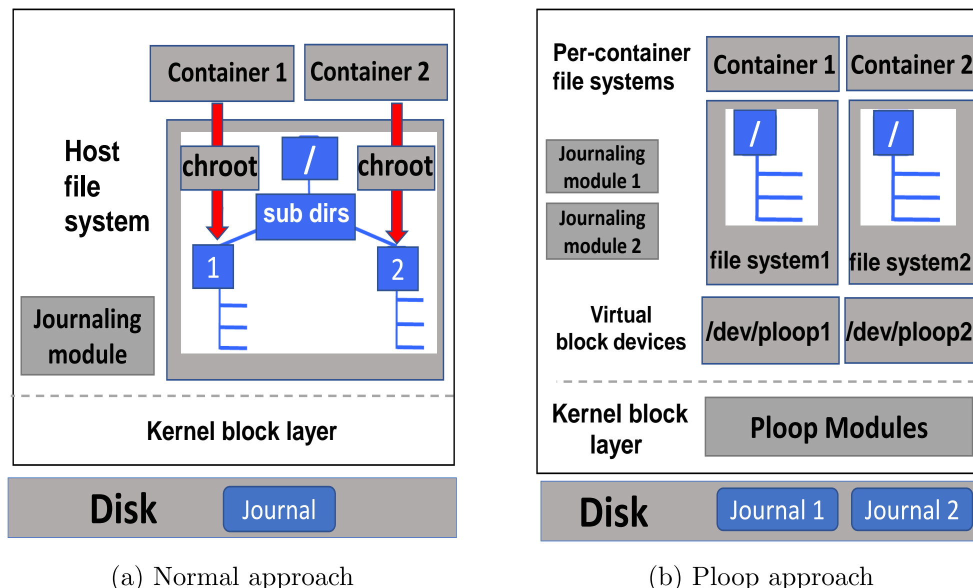
The objective of this chapter is to quest for possible solutions to alleviate file-system journaling problems in containers. The investigation in Chapter 4 reveals the underlying causes behind journaling problems in containers. This chapter proposes a method to overcome these causes without re-designing the file-system or modifying the journaling mechanism. The careful configuration of containers can gracefully solve the file-system journaling problems. The proposed method achieves per-container journaling and eliminates the bottleneck of the shared journaling module and its performance dependencies. Also, the configuration method overcomes the problem of overlooked journaling I/O by the `cgroup`. The quantitative analysis shows the feasibility of the proposed configuration in improving DBMS performance and isolation in containers. Finally, the chapter explores the performance of an in-memory database system that gains popularity recently. The results show that the sharing of the journaling module in containers degrades the in-memory database's performance as well. The proposed configuration successfully improves the in-memory database's performance and mitigates the journaling effects.

5.1 A Quest for Best Solution

The results of DBMS performance and isolation in Chapter 4 show that containers are not suitable for DBMS consolidation. The analysis reveals that the sharing of the journaling module degrades DBMS performance and violates its isolation in containers. Disabling the file system journaling to avoid the journaling problems in containers is not acceptable especially in DBMS because the journaling is indispensable to guarantee crash consistency. On the other hand, the use of VMs to avoid the journaling problems comes with the cost of virtualization overheads. Also, the use of VMs wastes the other advantages of containers like lightweight, scalability, and faster provisioning which are important for deploying applications in clouds. Some OS researchers work on designing a completely new file-system or developing novel journaling mechanisms to overcome the journaling issues, like the bundled transaction. However, all of these works require heavy implementation and involve non-negligible modifications to the kernel. Hence, these solutions are difficult to deploy on current cloud platforms. Also, these works don't solve all of the journaling problems in containers, like overlooked journaling I/Os. By identifying the underlying causes behind journaling problems in containers, it is possible to overcome these problems in a more straightforward and acceptable solution. A proper configuration of containers that bypass the underlying causes can solve the journaling problems as it will be shown in the next section.

5.2 Proposed Configuration Method

This section demonstrates that journaling problems can be gracefully alleviated 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. The proposed configuration can be applied to the mainstream Linux and existing file systems without any modification. Section 5.2.1 shows a configuration that avoids bundled transactions and `fsync` calls serialization on the journaling module. Section 5.2.2 shows the



(a) Normal approach

(b) Ploop approach

Figure 5.1: The architecture of "Normal approach" and "Ploop approach" of container implementation.

per-container accounting of journaling I/Os is possible. The combination of these two configurations solves journaling problems in containers and improves I/O performance and isolation.

5.2.1 Per-container Journaling Module

Containers share the same host file-system. Containers rely on `chroot` [40] to provide a per-container view of a file-system, which is a sub-tree of the host file system [74] [44]. 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. 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 file-system components. This results in the sharing of kernel resources for managing files. All the containers share the same file system type,

properties, total number of the inode, cache layer, and most importantly the on-disk journal and the journaling module. Figure 5.1 (a) illustrates the normal approach of container's configuration.

The sharing of the journaling module among containers 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. The journaling module batches updates from multiple containers into a single transaction and commits that transaction to disk periodically or when `fsync` is invoked. 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 alleviate this journaling problem is to provide a per-container journaling module. It disposes of bundled transactions and avoids the performance dependency among containers. By assigning a virtual block device to each container, each container will have its own file-system. Hence, 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. Also, `fsync` from one container will flush only data belonging to that particular container and it avoids the `fsync` calls contention on the shared journaling module.

The use of loopback device

An easy way to implement a virtual block device is to use a Linux loopback device. A loopback device is a pseudo-device that makes a regular file accessible as a block device. It maps data blocks not to a physical device such as a hard disk, but to the blocks of a regular file in a file-system [42]. 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 to bypass caching layer

in the kernel. Direct I/O is important in DBMS because DBMS manages its 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 backup and data protection.

The use of Ploop

OpenVZ supports “Ploop” [58], a special implementation of the loopback device which overcomes all the limitations of the Linux loopback device. In the Ploop approach, the Ploop modules in the kernel block layer are responsible for presenting a virtual disk for each container. Each container has its own file system of different types and properties. Figure 5.1 illustrates the Ploop virtual disk approach and compares it with the normal approach of providing file-system view in containers. The Ploop implementation has a modular and a layered design. It consists of the top layer, the main ploop module which provides a virtual block device to be used for the container’s file-system. The middle layer is the format module, which does the translation of block device numbers into image file block numbers. The bottom layer is the I/O module that responsible for dispatching I/Os to the underlying hardware. It provides support for direct I/O and avoiding the double cache problem. Also, Ploop provides support for features that are missing in the loopback device, like dynamic allocation, snapshot, and migration.

5.2.2 Journaling I/O Accounting

Assigning each container with a virtual block device is not enough to overcome all journaling problems in containers. Despite each container has its own journaling module and journaling transactions are separated, the journaling I/Os are not accounted for. The journaling I/Os from these per-

container journaling modules are still overlooked by the `cgroup`. The journaling modules are managed by the kernel and are running outside the controlled containers.

The journaling I/Os are performed by a kernel process known as "jbd2" which is responsible for file-system updates write to disk. In the case of per-container journaling, this kernel process is created for each container. These per-container jbd2 processes belong to the kernel, not the container. Since they are outside of `cgroup` control, their I/Os are not accounted for 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` [39] functionality. It changes the `cgroup` of the jbd2 process from the kernel's root `cgroup` to the corresponding container's `cgroup`.

5.3 Experiments

This section evaluates the proposed configuration method to confirm it can overcome journaling problems in containers. The experimental setup is described in Section 3.4.1 and Section 4.1.1 . Section 5.3.1 presents the results and the performance improvement of assigning the container with a virtual block device. Section 5.3.2 presents the combined performance improvement when the journaling I/O is being accounted for each container.

5.3.1 Per-container Journaling

To evaluate the effectiveness of the per-container journaling module on mitigating journaling problems, each container is assigned a virtual block device using Ploop. Since LXC doesn't support a container with a virtual block device, this configuration is only possible with OpenVZ. First, the I/O per-

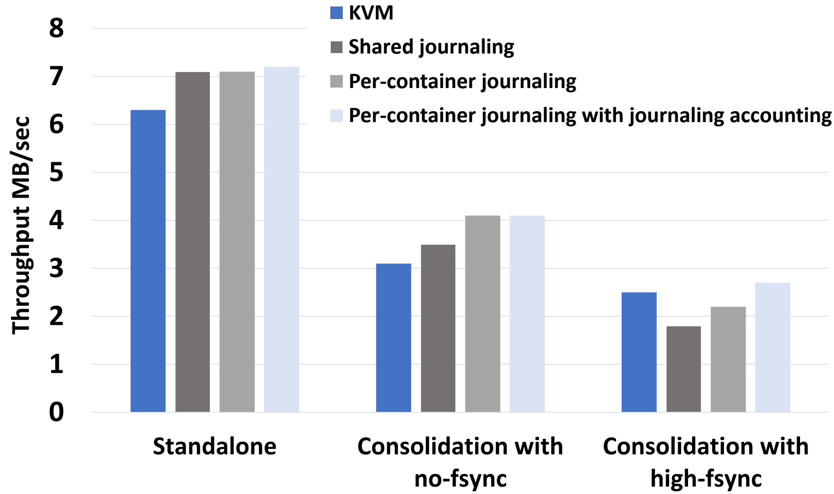


Figure 5.2: Disk I/O throughput in KVM, shared journaling, per-container journaling, and per-container journaling with journaling accounting.

formance is compared to check whether this configuration has any tax on the container's performance. The disk I/O throughput of the FIO sequential-write workload 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.

Figure 5.2 shows the I/O performance of the shared-journaling in OpenVZ container denoted by "shared journaling" and the per-container journaling 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 Section 5.3.2. 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 bundled transaction

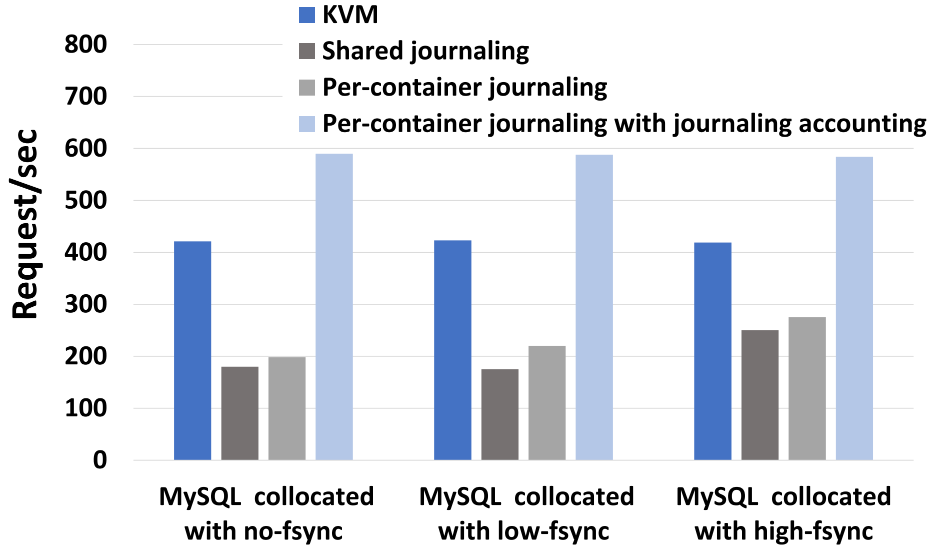


Figure 5.3: MySQL throughput in KVM, shared journaling, and per-container journaling without/with accounting. MySQL container is collocated with either 1) no-, 2) low-, or 3) high-fsync workloads. MySQL container/VM is given 70% share.

is not a problem anymore and each container has separate transactions that are committed in parallel. The reduction of the `fsync` latency confirms the improvement. The `fsync` latency is 80ms and 58ms in the shared journaling and the per-container journaling, respectively. However, the per-container journaling still performs lower than KVM in the high-fsync workload. This points out the per-container journaling does not overcome the journaling problems completely.

Figure 5.3 shows MySQL throughput when it is collocated with a container running either 1) no-, 2) low-, or 3) high-fsync workload. MySQL VM/container is given a 70% share and the other is given a 30% share. As shown in the figure, the MySQL throughput of the per-container journaling outperforms the shared-journaling in all cases. MySQL throughput is improved by up to 1.3x the per-container journaling. Table 5.1 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

Table 5.1: Average fsync latency of MySQL when collocated with no-, low-, high-fsync workload. JA stands for journaling accounting. MySQL container is given 70% share.

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

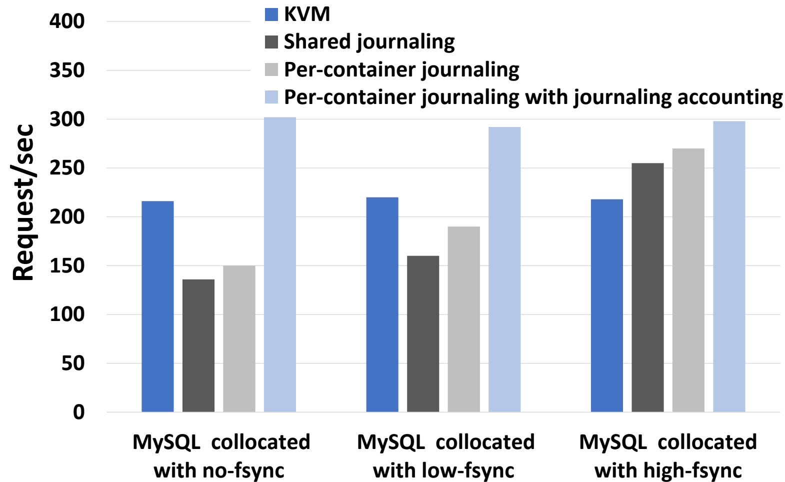


Figure 5.4: MySQL throughput in KVM, shared journaling, and per-container journaling without/with accounting. MySQL is collocated with either 1) no-, 2) low-, or 3) high-fsync workloads. MySQL container/VM is given 30% share.

per-container journaling. Compared with KVM, the per-container journaling still performs lower. Similar results are obtained in Figure 5.4 when MySQL VM/container is given a 30% share and the other is given a 70% share. The per-container journaling outperforms the shared-journaling in MySQL throughput. Table 5.2 shows the latency of `fsync` is reduced in per-container journaling. However, the per-container journaling still performs lower than

Table 5.2: Average fsync latency of MySQL when collocated with no-, low-, high-fsync workload. JA stands for journaling accounting. MySQL container is given 30% share.

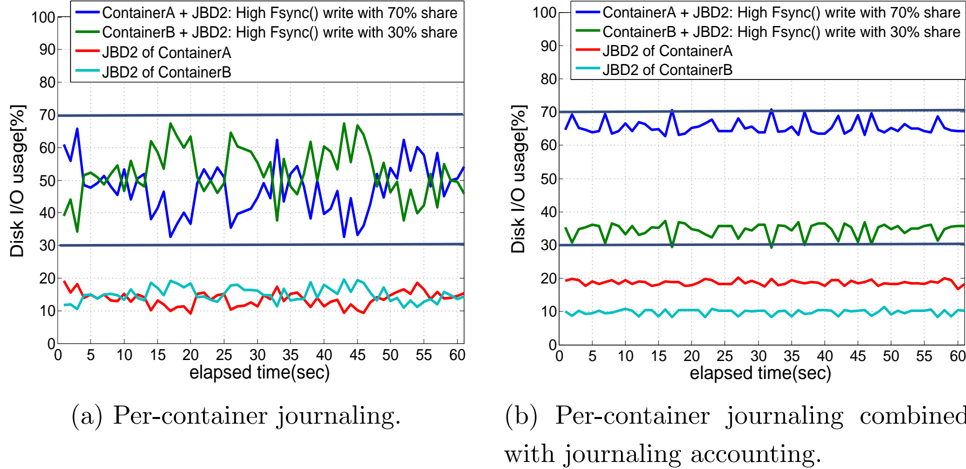
Collocated workload	Shared journaling	Per-container journaling	Per-container journaling with JA
No-fsync	148.3ms	99.4ms	47.6ms
Low-fsync	172.51ms	92.5ms	44.8ms
High-fsync	93.1ms	98.3ms	46.5ms

KVM in MySQL throughput. This indicates the per-container journaling module is not enough to improve DBMS performance in containers.

5.3.2 Combined with Journaling I/O Accounting

The use of a virtual block disk with the container is not sufficient to overcome all the journaling-related problems. Although the performance is improved with the per-container journaling module, it still performs lower than VM with its performance overhead. The per-container journaling performs lower than VM 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 there is a separate JDB2 for each container, all the per-container JDB2s belong to the same `cgroup` and are given an equal share regardless of the I/O share of the container each JDB2 is responsible for. This mismatch in disk I/O share between the container and its corresponding JDB2 affects the I/O performance.

Figure 5.5 shows disk I/O usages of the per-container journaling without



(a) Per-container journaling.

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

Figure 5.5: Disk I/O isolation in per-container journaling and per-container journaling with journaling accounting". Both of containers run high-fsync workloads.

and with the journaling I/Os accounting which is explained in section 5.2.2. In this experiment, two containers are launched with disk I/O share set to 70% and 30%, respectively. Both containers run the high-fsync workload to cause file-system journaling. As shown in Figure 5.5a, the per-container journaling without journaling I/O accounting cannot enforce performance isolation. Disk I/O usage of both containers fluctuates between 35%–65%. Figure 5.5a 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 journaling I/O accounting enforces the performance isolation perfectly as shown in Figure 5.5b. 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.

Table 5.3 shows the latency of `fsync` in the per-container journaling without and with journaling accounting. Without the journaling accounting, the JDB processes for containers A and B are given an equal share. Hence, both

Table 5.3: 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%	55ms	45ms
container B / 30%	53ms	86ms

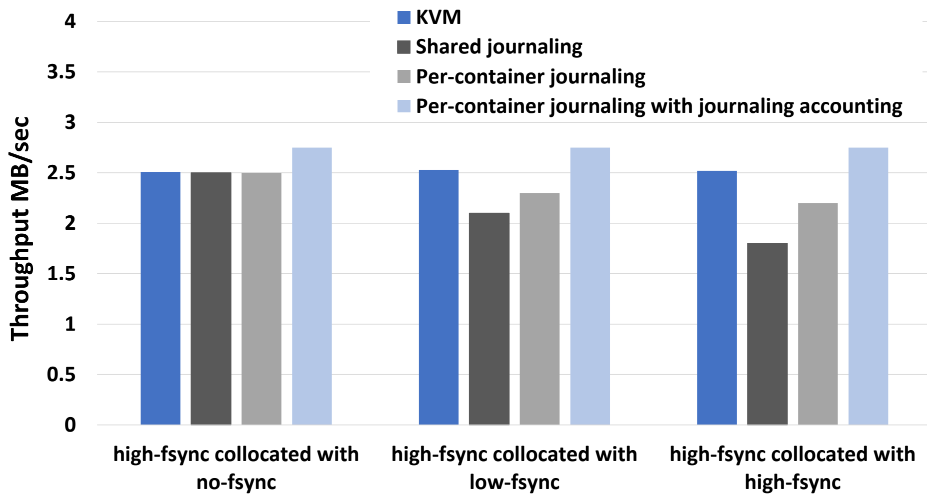


Figure 5.6: Throughput of disk I/O in KVM, shared journaling, and per-container journaling without/with accounting. A container/VM is running high-fsync workload and is collocated with either 1) no-, 2) low-, or 3) high-fsync workload.

containers A and B have almost the same `fsync` latency around 55ms. With the accounting, `fsync` latency of container A is reduced to 45ms while that of container B is increased to 86ms. This indicates the container A' JDB process is given more share while the container B's JDB process is less share in accordance with their respective container's `cgroup`.

Figure 5.6 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. The per-container journaling with the accounting improves the performance of per-container journaling as shown in the figure. The

per-container journaling with the accounting outperforms KVM in all cases and shows a stable throughput (around 2.7 MB/sec) regardless of collocated workload. This indicates that per-container journaling with the accounting gets back the performance gain of containers over VMs and mitigates the journaling problems successfully.

5.3.3 Improvement of DBMS performance and isolation

MySQL performance and isolation in containers are improved dramatically with the proposed configuration method. Figure 5.3 shows the per-container journaling with the accounting beats KVM in MySQL performance. MySQL throughput is improved by up to 1.4x compared to KVM and by up to 3.4x compared to the shared journaling. The per-container journaling with accounting enhances MySQL throughput by up to 2.6x compared to the per-container journaling. Table 5.1 shows `fsync` latency in the per-container journaling with the accounting becomes smaller than that of the per-container journaling. Similar results are obtained in Figure 5.4 when MySQL VM/container is given a 30% share instead of a 70% share. The per-container journaling with accounting tops the all in MySQL throughput. Table 5.2 confirms the latency of `fsync` is reduced with per-container journaling with accounting.

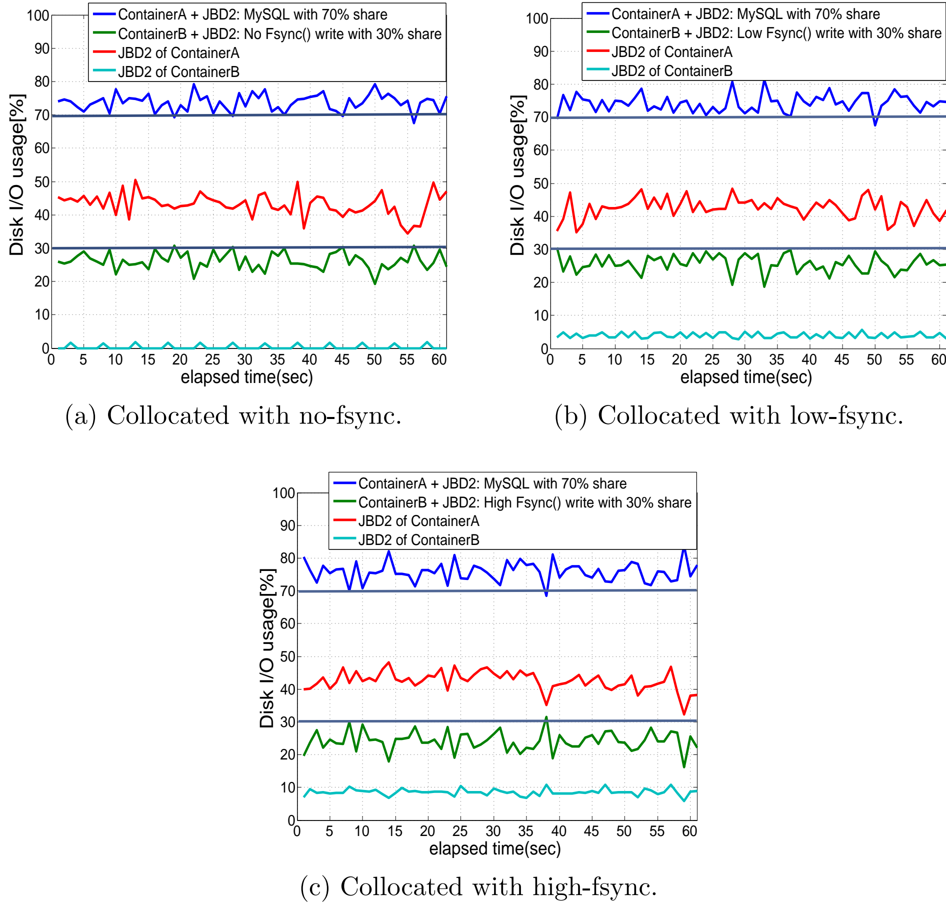
Aside from the performance, the per-container journaling with accounting has improved MySQL performance isolation. Figure 5.7 shows the disk I/O usage of the per-container journaling with accounting. A container running MySQL, which is given 70% share of disk I/O, is collocated with a container running either no-, low-, or high-fsync workload. Containers perfectly respect disk I/O control by `cgroup`. 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 Figure 5.8, MySQL container consumes around 25%, 30%, and 30%–50% share of disk I/O when collocated with no-, low- high-fsync workload, respectively. Figure 5.9 shows the disk I/O usage of the per-container journaling with accounting when MySQL is given 30% share. Containers show acceptable performance iso-

lation without any contentions. In all cases, disk I/O share is around 40% in MySQL. Figure 5.10 confirms that the per-container journaling without accounting is not sufficient to improve MySQL performance isolation since journaling I/O is not accounted.

MySQL performance and isolation are measured when both collocated containers/VMs run the MySQL database. Figure 5.11 compares the disk I/O usage in KVM and per-container journaling with accounting. The disk I/O is perfectly divided into 30% and 70% shares between containers. The per-container journaling with accounting achieves comparable performance isolation to KVM. Figure 5.12 shows MySQL throughput in this experiment (the throughput of 70% container/VM is reported). The per-container journaling with accounting outperforms KVM and tops the other containers' configuration in MySQL throughput.

5.4 In-memory Database Performance

Traditional DBMSes store datasets on disk or other underlying storage mediums. Traditional DBMS has high durability since data is stored in persistent storage. However, it requires disk access every time DBMS performs an operation like a query or an update which affects the performance. To enhance the performance, traditional DBMSes move frequently accessed data from disk to cache memory since the memory access is much faster than disk access. Recently, in-memory database management systems like Redis [8] and SQLite [75] have gained popularity. An in-memory database is a database that keeps the whole dataset in the main memory. An in-memory database offers higher performance and a minimal response time by eliminating the need to access the disk. Also, it avoids multiple data copies and irrelevant tasks, such as caching like an on-disk DBMS. Because all data is stored and managed in the main memory, it is at risk of being lost upon a server crash or a power failure. To ensure data durability, the in-memory database uses a transaction log (on-disk log) to persist each operation applied to memory as shown in Figure 5.13. In-memory database logs transactions by issuing `fsync` periodically or after a specific number of operations are performed in memory. Since in-memory database is also an journaling intensive and it



(a) Collocated with no-fsync.

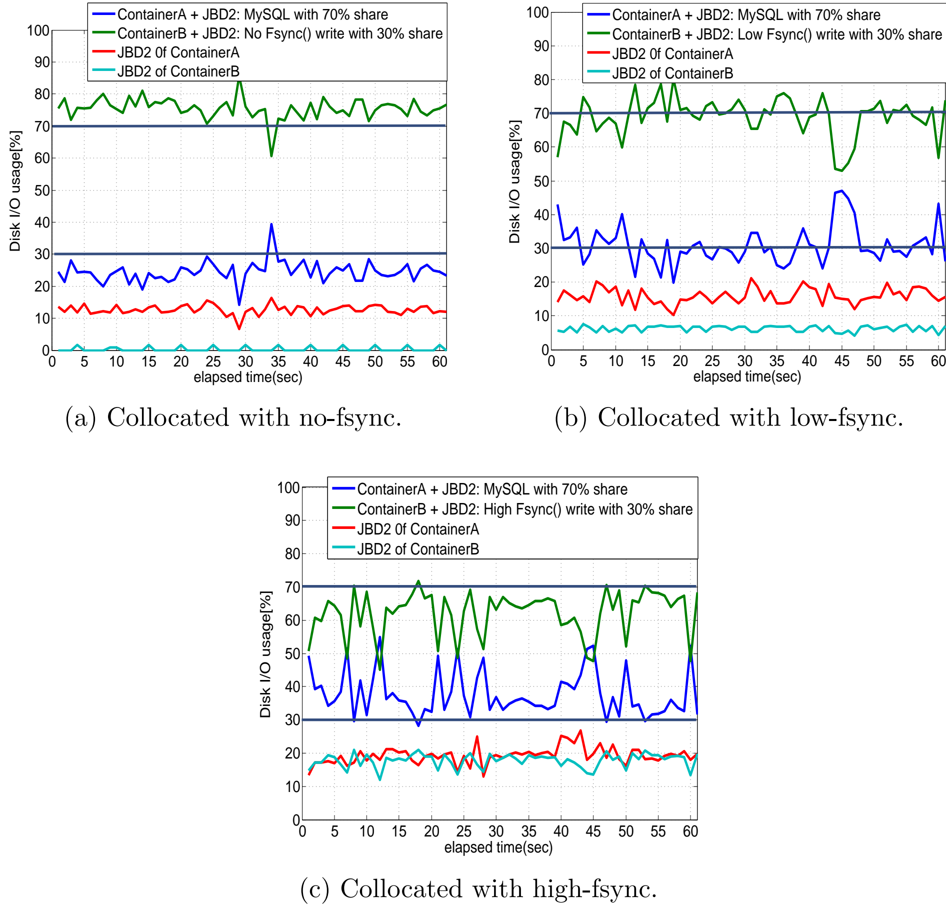
(b) Collocated with low-fsync.

(c) Collocated with high-fsync.

Figure 5.7: Disk I/O usage of 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.

performs disk I/O for transaction logging, it is highly likely that the sharing of journaling module affects the performance.

To verify whether the journaling problems in containers affect the in-memory database as well, the performance of the in-memory database is measured when it is consolidated with other workloads in containers. The same experimental environment in Chapter 3, Section 3.4.1 is used. Two collocated containers/VMs is launched, each with a 4GB of RAM which is enough to run the in-memory database. Redis ver. 2.8.4 is used as a represen-



(a) Collocated with no-fsync.

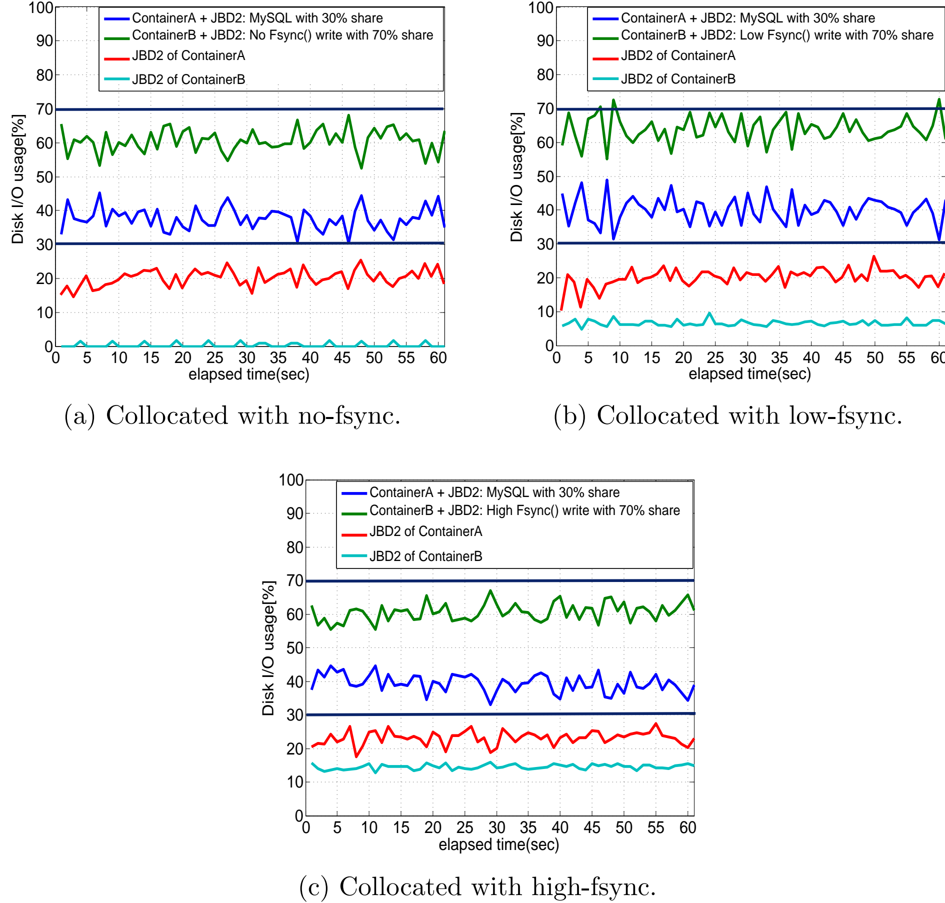
(b) Collocated with low-fsync.

(c) Collocated with high-fsync.

Figure 5.8: Disk I/O usage of MySQL in per-container journaling. Collocated with no-, low-, high-fsync workloads. MySQL is given 70% share and the other is 30% share.

tative of in-memory database and it is installed in each container/VM. Redis database is populated with 200,000 records. YCSB ver. 0.17.0 benchmark [21] generates workloads, which run in a separated machine connected via Cisco 1Gbit Ethernet switch. Redis is set to perform transactions logging every second which is the default setting.

First, the performance of Redis in a standalone case is measured. YCSB workload generates INSERT queries with 200,000 operation count. Figure 5.14 shows the throughput in KVM, LXC, and OpenVZ. As shown in the figure, LXC and OpenVZ outperform KVM which is expected because



(a) Collocated with no-fsync.

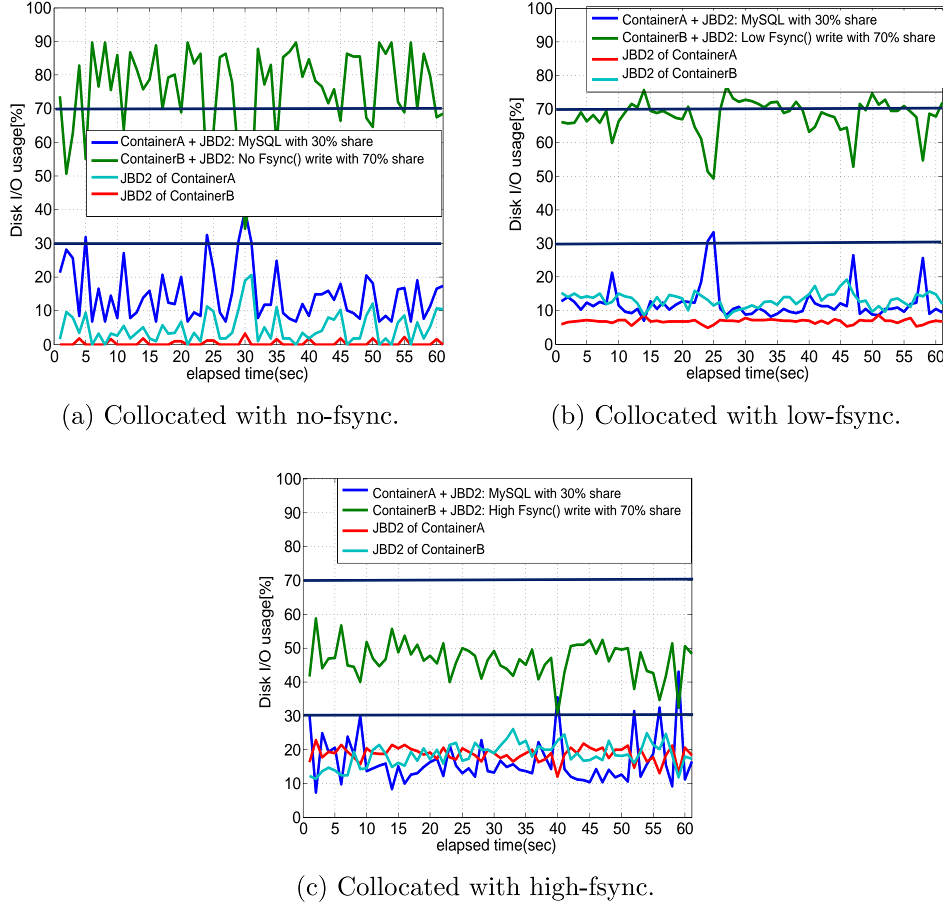
(b) Collocated with low-fsync.

(c) Collocated with high-fsync.

Figure 5.9: Disk I/O usage of MySQL in per-container journaling with the accounting. Collocated with no-, low-, high-fsync workloads. MySQL is given 30% share and the other is 70% share. Performance isolation is improved more.

of VM's virtualization overhead.

Next, the performance of Redis in the consolidation case is measured. Figure 5.15 shows Redis throughput when it is collocated with a VM/container running either 1) no-, 2) low-, or 3) high-fsync workload. In this case, the buffered I/O is used with FIO workloads since the in-memory database is being compared this time. As shown in the figure, Redis throughput in KVM is almost constant (around 1300 operation/sec) regardless of collocated workloads. On the other hand, the throughput in LXC and OpenVZ degrades



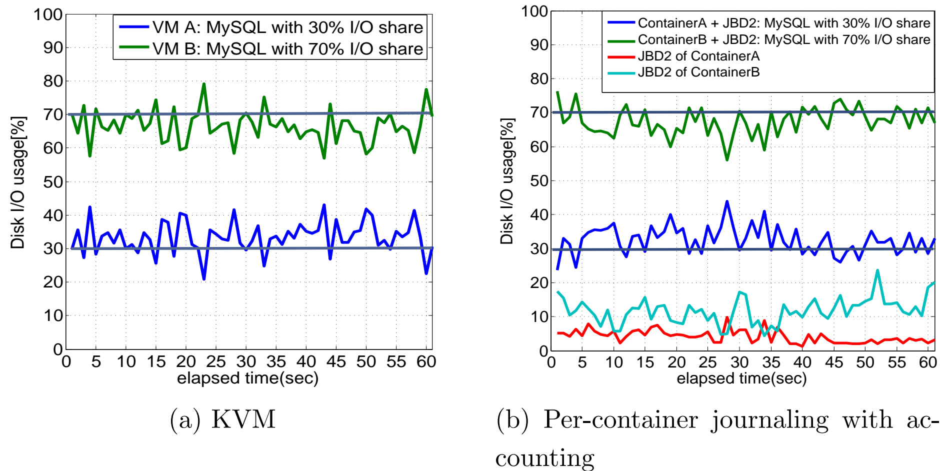
(a) Collocated with no-fsync.

(b) Collocated with low-fsync.

(c) Collocated with high-fsync.

Figure 5.10: Disk I/O usage of MySQL in per-container journaling. Collocated with no-, low-, high-fsync workloads. MySQL is given 30% share and the other is 70% share.

when the collocated workload changed from no-fsync to low-fsync. Redis throughput in LXC and OpenVZ is degraded more when the collocated workload changed from low-fsync to high-fsync. Table 5.4 shows `fsync` latency of Redis in LXC and OpenVZ are increased when the collocated workload issue more `fsync`. LXC and OpenVZ outperform KVM in Redis throughput only when it collocates with a no-fsync workload. This is because there is no much journaling contention between containers and because of virtualization overhead in VM. When Redis is collocated with a low- and high-fsync workload, KVM always shows better throughput than LXC and OpenVZ. In



(a) KVM

(b) Per-container journaling with accounting

Figure 5.11: Disk I/O usage in KVM and Per-container journaling with accounting when two MySQL containers/VMs are collocated together.

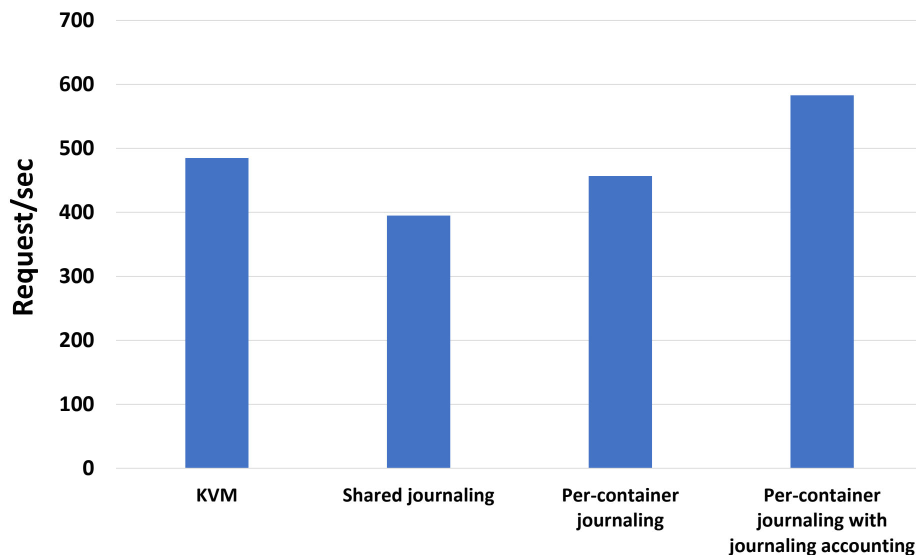


Figure 5.12: MySQL throughput in KVM and per-container journaling with accounting when two MySQL containers/VMs are collocated together.

this case, the journaling contention is higher between containers hence, the performance degradation from journaling problems in containers is higher than VM's performance overhead. These results suggest that the sharing

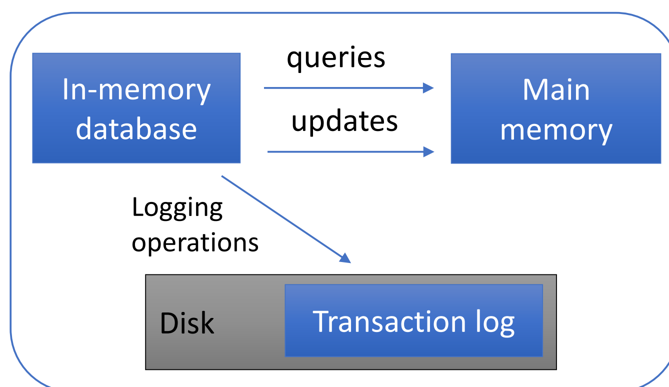


Figure 5.13: Typical In-memory database with data persistency.

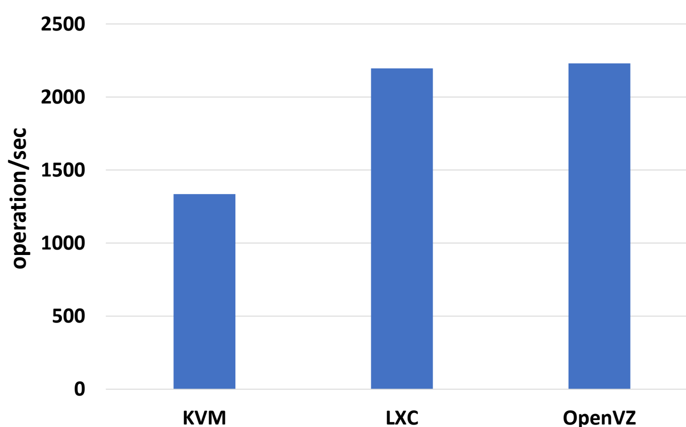


Figure 5.14: Redis throughput in KVM, LXC, and openVZ with INSERT queries workload.

of a journaling module among containers interferes with in-memory DBMS performance as well.

To confirm that the performance of the in-memory database is degraded because of journaling problems in containers, the performance of Redis is measured with the proposed configuration method. Figure 5.16 shows Redis throughput in shared journaling, and per-container journaling without/with accounting. The per-container journaling outperforms the shared journaling by up to 1.5x in Redis throughput. The per-container journaling shows a stable throughput (around 2200 operations/sec) regardless of collocated

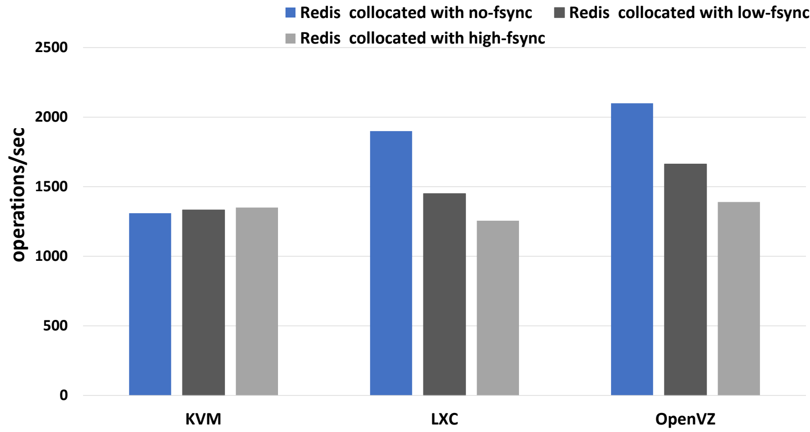


Figure 5.15: Redis throughput in KVM, LXC, and openVZ with either 1) no-, 2) low-, or 3) high-fsync workloads.

Table 5.4: Average fsync latency of Redis when collocated with no-, low-, high-fsync workload.

Collocated workload	KVM	LXC	OpenVZ
No-fsync	(32.7ms)	(47.6ms)	(43.8ms)
Low-fsync	(33.2ms)	(58.5ms)	(51.71ms)
High-fsync	(32.5ms)	(65.7ms)	(57.5ms)

workload. Since each container has its journaling module, the journaling contention is mitigated. The `fsync` latency of Redis in the per-container journaling is almost constant around 60ms in all cases. Compared to per-container journaling with the accounting, there is no much improvement in the throughput. Because in-memory database performance doesn't depend on disk I/Os hence, journaling I/Os accounting by `cgroup` have no effects on the performance.

5.5 Discussion

The proposed method effectively alleviates the journaling problems in containers. I advocate the use of virtual block devices and the proper configuration of kernel processes for accounting journaling I/Os in containers. I

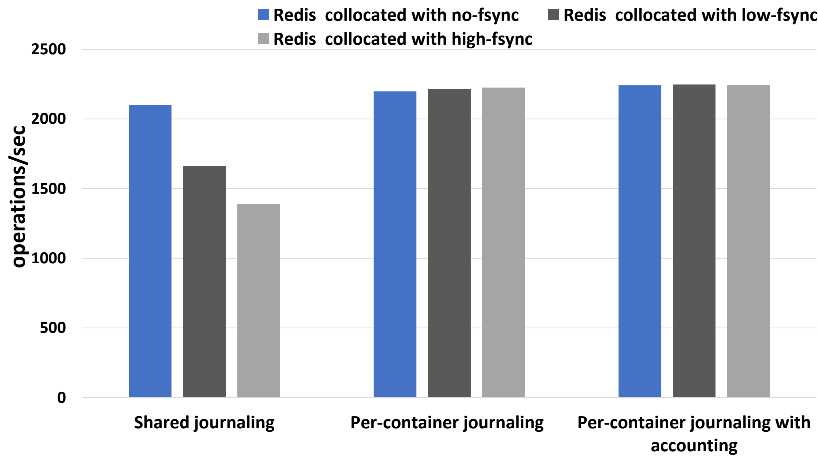


Figure 5.16: Redis throughput in shared journaling, per-container journaling, and per-container journaling with accounting.

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 this configuration if they implement a similar virtual block device with direct I/O support, preventing the double cache, and the required features for container storage management like the snapshot and dynamic re-sizing. Recently, Docker container adds the support of `direct-lvm` mode with `devicemapper` storage diver. Such configuration enables to use of a block device as a dedicated volume to each container hence, each container has its own file system similar to the `Ploop`. However, one problem with this configuration is `cgroup` weigh-based I/O controller doesn't work with lvm logical volumes because the group weight doesn't get propagated down through logical layers.

The performance analysis reveals that the use 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 problems without re-design the file-system or the journaling mechanism.

Unfortunately, this proposed configuration is not widely adopted in major clouds because most container implementations do not allow this configura-

tion. Major cloud services like Azure, Google cloud, and AWS neither use this configuration nor support the OpenVZ with Ploop. This causes poor I/O performance and isolation when consolidating DBMS and wastes the container's advantages over the VM. This results in violating the service-level agreement and causes financial loss for cloud provides. However, with adopting the proposed method, it is possible to use containers for DBMS consolidation in clouds with the performance gain of containers and without suffering from the journaling problems.

While the proposed method successfully solved the journaling problem in the current software system with disk storage, the work in this dissertation may add some contribution to the design of future software systems and the use of emerging hardware. The much faster flash memory storage like SSD and NVMe is replacing the disk storage in data centers. The problem of shared journaling module in containers may be amplified especially with a use case of update intensive applications like DBMS. Since these fast storage technologies are capable of performing a much higher number of storage I/O compared to disk I/O, the contention over shared journaling module will be higher, which will waste the performance of these fast storage devices. There will be poor scalability between the emerging storage devices and the traditional file-systems that are designed for general-purpose use. In the case of containers and production environments in the cloud, these file-systems may have contentions over shared components or data structures that cause performance interference among containers from one side and underutilizes the high performance of emerging storage devices from another side. Hence, the software developers should consider the performance gap between the emerging hardware and traditional design of software systems. This can be done by re-designing the OS kernel and file-systems with decentralizing the components and partitioning data structures that cause the bottleneck and contentions.

5.6 Summary

This chapter proposed a method to overcome file-system journaling problems in containers. Instead of designing a complete file system or developing

a new journaling method, the chapter demonstrates that careful configuration of containers in existing file systems can gracefully solve the journaling problems. The proposed configuration consists of first, per-container journaling by presenting each container with a virtual block device to have its own journaling module. Hence, eliminating the bottleneck of the shared journaling module among containers and its performance dependencies. Second, accounting journaling I/Os separately for each container with a proper configuration of journaling processes. The quantitative analysis shows the feasibility of the proposed configuration in improving DBMS performance and isolation in containers. The experimental results show MySQL performance in containers is improved by 3.4x with the proposed method. Eventually, containers outperform VMs by 1.4x and show the performance gain over VMs. Containers achieve reasonable performance isolation in MySQL consolidation, comparable to that of VMs. Finally, the chapter explores the performance of Redis, an in-memory database system. The results show that the sharing of the journaling module in containers degrades the in-memory database's performance as well. The proposed configuration successfully improves the in-memory database's performance and mitigates the journaling effects.

Chapter 6

Conclusion

6.1 Contribution Summary

This dissertation has conducted a study of DBMS performance and isolation in virtualization environments. The dissertation aimed at investigation the performance interference in virtualization environments by discussing the storage I/O in DBMS as a first step. I/O-intensive applications like DBMS demand high I/O performance and strict performance isolation when they are consolidated in the cloud. A comparative study of KVM, LXC, and OpenVZ is performed to understand the trad-off between I/O performance and isolation in containers and VMs. Most of today's cloud service providers adopt containers to achieve higher performance than traditional VMs. However, the experimental results show the opposite, VMs outperform containers in DBMS performance. Also, containers show a terrible disk I/O isolation in DBMS consolidation. The file-system journaling which is mandatory to guarantee file-system consistency especially in the database application is the root cause of poor DBMS performance and isolation in containers. When update intensive applications like DBMS are consolidated in containers, the shared journaling module easily becomes a bottleneck. Journaling activities of one container affect I/O performance and isolation of other containers since DBMS involves frequent journaling.

The dissertation exposes that the sharing of some kernel components violates the performance and isolation in containers. The investigation reveals

the underlying causes behind file-system journaling problems in containers. First, the shared journaling module causes performance dependency among containers and drgrades I/O performance. Second, file-system journaling in containers interferes with disk I/O control of `cgroup` which violate the isolation between containers.

Instead of designing a complete file-system or developing a new journaling method, the dissertation demonstrates that careful configuration of containers can resolve the journaling problems. The proposed configuration method 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. The experimental results show that DBMS performance in containers is improved up to 3.4x and outperform VM by 1.4x with the proposed configuration. Aside from the performance, containers achieve reasonable DBMS performance isolation comparable to that of VMs.

6.2 Future Work

This dissertation paves the way to more investigation on performance interference in containers. The study has uncovered that the sharing of some kernel components can become a bottleneck and violate performance isolation among containers. Thus, one of the future works is to investigate the sharing of other kernel components, for example, memory and networking components. Since containers have become widely used to deploy WEB applications, it is interesting to investigate the possible resource contention in network I/O with network-intensive applications. Although this study explores the effect of the shared journaling module on the in-memory database performance, a further investigation is needed to understand the effect of the sharing of memory cache on in-memory database performance or the other memory-intensive applications.

Another future direction is to investigate the journaling problem with the other journaling file-systems like XFS [3] and NTFS [83]. It is interesting to compare the effects of their journaling mechanism on the I/O performance and isolation of containers.

Since this study advocates the use of per-container virtual block device and the need for per-container journaling I/O accounting. These features should be supported in containers and `cgroup` implementations. Container systems other than OpenVZ, especially the most used ones like Docker[12] and LXC, should support a container with a virtual block device like Ploop. On the other hand, the kernel `cgroup` should provide a better disk I/O control. 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 the non-journaling I/O behaviors of each container. This estimate could be used to adjust the weight of disk I/Os for each container or a normal process by the `cgroup`.

Bibliography

[1] Amazon. Amazon Web Services, 2021. https://aws.amazon.com/.

[2] Amazon. Databases on AWS, 2021. https://aws.amazon.com/products/databases/.

[3] Archlinux. XFS, 2021. https://wiki.archlinux.org/index.php/XFS.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

[5] S. K. Barker and P. Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. In *Proc. of the first annual ACM SIGMM conference on Multimedia systems (MMSys)*, pages 35–46. ACM, 2010.

[6] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46. Califor-nia, USA, 2005.

[7] F. Caglar, S. Shekhar, and A. Gokhale. Towards a performance interferenceaware virtual machine placement strategy for supporting soft realtime applications in the cloud. In *Proc. of 3rd International Workshop on Real-time and Distributed Computing in Emerging Applications (REACTION)* , pages 15–20. Universidad Carlos III de Madrid, 2014.

[8] J. Carlson. *Redis in action.* Simon and Schuster, 2013.

[9] J. Che, Y. Y. ans C. Shi, and W. Lin. A synthetical performance evaluation of openvz, xen and kvm. In *Proc. of Asia-Pacific Services computing Conference*, pages 587–594. IEEE, 2010.

[10] Datadog. Real-world container use, 2020. `https://www.datadoghq.com/container-report/#11`.

[11] Diamanti. Container Adoption Benchmark Survey, 2019. `https://diamanti.com/wp-content/uploads/2019/06/Diamanti_2019_Container_Survey.pdf`.

[12] Docker. Get Started with Docker, 2021. `https://www.docker.com/`.

[13] A. Dusseau, H. Remzi, A. Dusseau, and C. Andrea. *OPERATING SYSTEMS*. Arpaci-Dusseau Books, 2014.

[14] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. Technical Report RC25482 (AUS1407-001), IBM Research Division, 2014.

[15] FIO. fio - Flexible I/O tester, 2021. `https://fio.readthedocs.io/en/latest/index.html`.

[16] M. Furman. *OpenVZ essentials*. Packt Publishing Ltd, 2014.

[17] N. Galov. Cloud Adoption Statistics for 2021, 2021. `https://hostingtribunal.com/blog/cloud-adoption-statistics/#gref`.

[18] S. K. Garg and J. Lakshmi. Workload performance and interference on containers. In *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, pages 1–6. IEEE, 2017.

[19] S. K. Garg, J. Lakshmi, and J. Johny. Migrating VM workloads to containers: Issues and challenges. In *11th IEEE International Conference*

on Cloud Computing, CLOUD 2018, San Francisco, CA, USA, July 2-7, 2018, pages 778–785, 2018.

[20] Gartner. The Future of the DBMS Market Is Cloud, data management usage projections by analyst house Gartner, Inc., 2019. `https://www.gartner.com/en/newsroom/press-releases/2019-07-01-gartner-says-the-future-of-the-database-market-is-the`.

[21] Github. Yahoo! Cloud Serving Benchmark, 2021. `https://github.com/brianfrankcooper/YCSB`.

[22] Google. Cloud SQL, 2021. `https://www.mysql.com/customers/view/?id=757`.

[23] Google. Google Cloud Platform, 2021. `https://cloud.google.com/`.

[24] A. Gulati, C. Kumar, and I. Ahmad. Storage workload characterization and consolidation in virtualized environments. In *Workshop on Virtualization Performance: Analysis, Characterization, and Tools (VPACT)*, page 4, 2009.

[25] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in xen. In *Middleware 2006*, pages 342–362. Springer, 2006.

[26] J. G. Hansen and E. Jul. Lithium: virtual machine storage for the cloud. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 15–26, 2010.

[27] T. Heo. Control Group v2, 2021. `https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html`.

[28] IBM. Containers vs. VMs: What's the Difference?, 2020. `https://www.ibm.com/cloud/blog/containers-vs-vms`.

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

[30] M. T. Jones. Anatomy of linux journaling file systems. *IBM Developer-Works*, 2008.

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

[32] J. Kang, B. Zhang, T. Wo, W. Yu, L. Du, S. Ma, and J. Huai. Spanfs: a scalable file system on fast storage devices. In *Proc. of USENIX Annual Technical Conference (ATC)*, pages 249–261, 2015.

[33] M. Kerrisk. Namespaces Linux manual page, 2020. `https://man7.org/linux/man-pages/man7/namespaces.7.html`.

[34] M. Kerrisk. Cgroup Linux manual page, 2021. `https://man7.org/linux/man-pages/man7/cgroups.7.html`.

[35] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the linux virtual machine monitor. In *Proc. of the Linux symposium*, volume 1, pages 225–230, 2007.

[36] A. Kopytov. Sysbench manual. *MySQL AB*, pages 2–3, 2012.

[37] 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. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 183–191. USENIX Association, Feb. 2020.

[38] Linux. Block IO Controller, 2021. `https://www.kernel.org/doc/Documentation/cgroup-v1/blkio-controller.txt`.

[39] Linux. cgclassify(1) - Linux man page, 2021. `https://linux.die.net/man/1/cgclassify`.

[40] Linux. Chroot- Linux man page, 2021. `https://linux.die.net/man/2/chroot`.

[41] Linux. fsync(2) — Linux manual page, 2021. https://man7.org/linux/man-pages/man2/fdatasync.2.html.

[42] Linux. loop(4) — Linux manual page, 2021. https://man7.org/linux/man-pages/man4/loop.4.html.

[43] L. Lu, Y. Zhang, T. Do, S. AI-Kiswany, and R. Arpaci-Dusseau. Physical disentanglement in a container-based file system. In *Proc. of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 81–96, 2014.

[44] LXC. Linux containers, Infrastructure for container projects, 2021. https://linuxcontainers.org/.

[45] D. Macrae. How Linux Kernel Cgroups And Namespaces Made Modern Containers Possible, 2016. https://www.silicon.co.uk/software/open-source/linux-kernel-cgroups-namespaces-containers-186240.

[46] J. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, and D. G. Hamilton. Quantifying the performance isolation properties of virtualization systems. In *Proc. of the 2007 Workshop on Experimental Computer Science, (ExpCS)*, page 6. ACM, 2007.

[47] I. Mavridis and H. Karatza. Combining containers and virtual machines to enhance isolation and extend functionality on cloud computing. *Future Generation Computer Systems*, 94:674–696, 2019.

[48] R. McDougall and J. Anderson. Virtualization performance: perspectives and challenges ahead. *ACM SIGOPS Operating Systems Review*, 44(4):40–56, 2010.

[49] P. Menage. Linux Cgroup resource management, 2021. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[50] A. Menon, J. R. Santos, Y. Turner, G. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the xen virtualmachine environment. In *Proc. of the 1st ACM/USENIX inter-*

national conference on Virtual execution environments (VEE), pages 13–23. ACM, 2005.

[51] Microsoft. Azure SQL Database, 2021. `https://azure.microsoft.com/en-us/services/sql-database/`.

[52] Microsoft. Microsoft Azure, 2021. `https://azure.microsoft.com/en-us/`.

[53] N. Mizusawa, J. Kon, Y. Seki, J. Tao, and S. Yamaguchi. Performance improvement of file operations on overlayfs for containers. In *Proc. of IEEE International Conference on Smart Computing*, pages 297–302. IEEE, 2018.

[54] R. Morabito, J. Kjallman, and M. Komu. Hypervisors vs. lightweight virtualization: a performance comparison. In *Proc. of the International Conference on Cloud Engineering (IC2E)*, pages 368–374. IEEE, 2015.

[55] MySQL. MySQL Customer: Facebook, 2021. `https://www.mysql.com/customers/view/?id=757`.

[56] NetApp. What are containers?, 2021. `https://www.netapp.com/devops-solutions/what-are-containers/`.

[57] L. Nussbaum, F. Anhalt, O. Mornard, , and J. P. Gelas. Linux-based virtualization for HPC clusters. In *In Proc. of Montreal Linux Symposium*, pages 221–234, 2009.

[58] OpenVZ. Ploop, 2021. `https://wiki.openvz.org/Ploop`.

[59] OpenVZ. Ploop/Why, 2021. `https://wiki.openvz.org/Ploop/Why`.

[60] D. Park and D. Shin. iJournaling: Fine-grained journaling for improving the latency of fsync system call. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 787–798. USENIX Association, 2017.

[61] Percona. Open Source Data Management Software Survey, 2020. `https://www.percona.com/open-source-data-management-software-survey`.

[62] V. Prabhakaran, A. Arpaci-Dusseau, , and R. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *Proc. of 2005 USENIX Annual Technical Conference*, pages 105–120. USENIX, 2005.

[63] V. Prabhakaran, A. Arpaci-Dusseau, , and R. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *Proc. of 2005 USENIX Annual Technical Conference*, pages 105–120. USENIX, 2005.

[64] M. Raho, A. Spyridakis, M. Paolino, and D. Raho. Kvm, xen and docker: a performance analysis for arm based nfv and cloud computing. In *Proc. of the 3rd Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*, pages 1–8. IEEE, 2015.

[65] L. Rasmusson and C. Diarmuid. Performance overhead of kvm on linux 3.9 on arm cortex-a15. *ACM SIGBED Review*, 11(2):32–38, 2014.

[66] Redhat. Chapter 3. Subsystems and Tunable Parameters, 2021. `https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/htmlresource_management_guide/ch-subsystems_and_tunable_parameters`.

[67] Redhat. CHAPTER 7. SWAP SPACE, 2021. `https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/5/html/deployment_guide/ch-swapspace`.

[68] Redhat. What is KVM?, 2021. `https://www.redhat.com/en/topics/virtualization/what-is-KVM`.

[69] N. Regola and J. Ducom. Recommendations for virtualization technologies in high performance computing. In *Proc. of International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 409–416. IEEE, 2010.

[70] D. Rountree and I. Castrillo. *The basics of cloud computing: Understanding the fundamentals of cloud computing in theory and practice.* Newnes, 2013.

[71] R. Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[72] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference*, pages 1:1–1:13. ACM, 2016.

[73] J. Y. Shin, M. Balakrishnan, T. Marian, and H. Weatherspoon. Gecko: Contention-oblivious disk arrays for cloud storage. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 285–297, San Jose, CA, Feb. 2013. USENIX Association.

[74] S. Soltesz, H. Potzl, M. Fiuczynski, A. Bavier, and L. Peterson. Container based operating system virtualization: a scalable, high-performance alternative to hypervisors. *SIGOPS Operating System Review*, 41(3):275–287, 2007.

[75] SQLite. Well-Known Users of SQLite, 2021. `https://www.sqlite.org/famous.html`.

[76] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci. Cntr: Lightweight OS containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 199–212, Boston, MA, July 2018. USENIX Association.

[77] D. Tobin. The Salesforce Database Explained, 2019. `https://www.xplenty.com/blog/the-salesforce-database-explained/`.

[78] B. Verghese, A. Gupta, and M. Rosenblum. Performance isolation: sharing and isolation in shared-memory multiprocessors. In *ACM SIGPLAN Notices*, volume 33, pages 181–192. ACM, 1998.

[79] VMware. Own Your Path to the Future, 2021. `https://www.vmware.com/`.

[80] Vserver. Welcome to Linux-VServer, 2018. `http://www.linux-vserver.org/Welcome_to_Linux-VServer.org`.

[81] Vserver. Oracle Solaris Containers, 2021. `https://www.oracle.com/solaris/technologies/solaris-containers.html`.

[82] Windows. Introduction to Hyper-V on Windows, 2021. `https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/`.

[83] Windows. NTFS — New Technology File System for Windows, 2021. `https://www.ntfs.com/index.html`.

[84] 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. In *Proc. of 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 233–240. IEEE, 2013.

[85] 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. In *Proc. of 23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 253–260. IEEE, 2015.

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

[87] B. Zhang, X. Wang, R. Lai, L. Yang, Z. Wang, Y. Luo, and X. Li. Evaluating and optimizing i/o virtualization in kernel-based virtual machine (kvm). In *IFIP International Conference on Network and Parallel Computing*, pages 220–231. Springer, 2010.

List of Papers

Articles on Periodicals

- Asraa Abdulrazak Ali Mardan and Kenji Kono. When the Virtual Machine Wins over the Container: DBMS Performance and Isolation in Virtualized Environments. *IPSJ Journal of Information Processing*, Vol.28, pp.369–377, July 2020.

- Asraa Abdulrazak Ali Mardan and Kenji Kono. Alleviating File System Journaling Problem in Containers for DBMS Consolidation. *IEICE Transactions on Information and Systems*, Vol.E104-D, No.7, pp.–, July 2021.

Articles on International Conference Proceedings

- Asraa Abdulrazak Ali Mardan and Kenji Kono. Containers or Hypervisors, Which is Better for Database Consolidation?. In *Proceedings of the IEEE 8th International Conference on Cloud Computing Technology and Science (CloudCom '16)*, pp.564–571, December 2016.