Container-Aware I/O Stack: Bridging the Gap between Container Storage Drivers and Solid State Devices

Song Wu
CGCL/SCTS/BDTS, HUST
Wuhan, China
wusong@hust.edu.cn

Zhuo Huang
CGCL/SCTS/BDTS, HUST
Wuhan, China
huangzhuo@hust.edu.cn

Pengfei Chen
CGCL/SCTS/BDTS, HUST
Wuhan, China
chenpf97@hust.edu.cn

Hao Fan
CGCL/SCTS/BDTS, HUST
Wuhan, China
haofan@hust.edu.cn

Shadi Ibrahim
Inria, Univ. Rennes, CNRS, IRISA
Rennes, France
shadi.ibrahim@inria.fr

Hai Jin
CGCL/SCTS/BDTS, HUST
Wuhan, China
hjin@hust.edu.cn

Abstract

Solid State Devices (SSDs) have been widely adopted in containerized cloud platforms as they provide parallel and high-speed data accesses for critical data-intensive applications. Unfortunately, the I/O stack of the physical host overlooks the layered and independent nature of containers, thus I/O operations require expensive file redirect (between the storage driver, Overlay2/EXT4, and the *virtual file system*, VFS) and are scheduled sequentially. Moreover, containers suffer from significant I/O contention as resources at the native file system are shared between them. This paper presents a ***C**ontainer-**a**ware I/O **st**ack* (CAST). CAST is made up of ***L**ayer-**a**ware **VFS*** (LaVFS) and ***C**ontainer-**a**ware Native **F**ile **S**ystem* (CaFS). LaVFS locates files based on layer information and enables simultaneous *Copy-on-Write* (CoW) operations and thus avoids the overhead of searching and modifying files. CaFS, on the other hand, provides contention-free access by designing fine-grain resource allocation at the native file system. Experimental results using a NVMe SSD with micro-benchmarks and real-world applications show that CAST achieves 216%-219% (38%-98%, respectively) improvement over the original I/O stack.

CCS Concepts: • **Software and its engineering → File systems management**.

Keywords: SSD, container, file system, Overlay2

VEE '22, March 1, 2022, Virtual, Switzerland
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Song Wu, Zhuo Huang, Pengfei Chen, Hao Fan, Shadi Ibrahim, and Hai Jin. 2023. Container-Aware I/O Stack: Bridging the Gap between Container Storage Drivers and Solid State Devices. In *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '22), March 1, 2022, Virtual, Switzerland*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Containers, as driven by the popularity of Docker [11] and Kubernetes [5], are widely adopted in cloud platforms. Containers can easily build isolated runtime environments for applications by *cgroup* [1] and *namespace* [30] on a shared kernel. The data of a container (i.e. , dependencies, configurations, and results) are stored in a layer-structured image. Through the storage driver, which is built on the I/O stack of the physical host, a unified file system view is provided to each container, and I/O operations of different containers are directed to those layers. Efficient data access to layers is critical for performance (i.e. , starting up containers, updating container image, and storing ephemeral data) [17].

Due to their high access performance and decreasing prices [15], *Solid State Devices* (SSDs) are gradually replacing disks in cloud platforms. Thus, applications execution can be accelerated by adopting them as local server storage [16]. Unfortunately, the performance benefit of SSDs is still not fully exploited when applications run in containerized environments. On the one hand, the additional tier introduced by image management (i.e. , OverlayFS, AUFS, Device Mapper, and BtrFS) results in an extra latency due to the file redirect [10, 27, 32]. On the other hand, operations from different containers show intense competition for shared resources because the resource allocation of the native file system is centralized and serialized.

Providing isolated I/O stacks for each container [12, 13, 18] can improve the performance of containerized applications. However, this requires more storage space and causes longer image download time (layers are not shared among containers). In addition, this may not be practical as the storage spaces of containers cannot be dynamically adjusted at runtime. In this paper, we propose a ***C**ontainer-**a**ware I/O **st**ack* (CAST). The basic idea of CAST is to make file location – in the *Virtual File System* (VFS) – layer-aware, and to isolate container resources at the native file system, i.e. , *Block Groups* (BGs) and the journal service. To do so, CAST is made up of *Layer-aware VFS* (LaVFS) and *Container-aware native File System* (CaFS). LaVFS opens files by switching view file path to real file path based on the layer information. Thus, OverlayFS does not need to search all the layers when opening files. LaVFS also enables simultaneous *Copy-on-Write* (CoW) operations by removing the VFS locks that are unnecessary for layered container files. CaFS, on the other hand, provides fine-grain resource allocation by taking advantage of the layer-structure of container files and considering that container modifications that involve resource allocation happen only in its read-write layer (*upper layer*). CaFS replaces the original journal with multiple micro journals. Each micro journal has its own transactions and is mapped to a specific *upper layer* to make sure different versions of the same data are issued to the same transaction. *Block Groups* (BGs) are grouped as *Container Regions* (CRs), and thus can be dynamically adjusted. An *upper layer* is bound to a CR and block allocation of an *upper layer* happens only in its own CR to prevent contention caused by BG sharing.

In summary, we make the following contributions:

- We comprehensively analyze the performance behaviors of the container storage driver on SSDs. We find that container operations suffer from expensive file redirect in the additional tier (OverlayFS) and intense resource competition in the native file systems. First, the existing I/O stack overlooks the layered and independent nature of containers. Thus, OverlayFS searches in all layers before opening a file and competes for a global lock during CoW. Second, the native file system cannot isolate resources based on containers.
- We propose ***C**ontainer-**a**ware I/O **st**ack* (CAST) composed of LaVFS and CaFS. LaVFS reduces file redirect overhead caused by opening files and modifying files by employing layer-aware path switch and simultaneous CoWs, respectively. CaFS provides fine-grain resource allocation at the native file system by grouping BGs into isolated container regions and replacing the original journal with isolated micro journals.
- We implement CAST in Linux kernel 4.16.1. Experimental results with micro-benchmarks show that CAST improves the throughput by up to 219% compared to the original I/O stack. In addition, CAST achieves 38%-98% improvement over the original I/O stack for real-world applications.

The rest of this paper is organized as follows. Section §2 introduces the background and the motivation of this work. Section §3 describes the design and implementation details of CaFS. Experimental methodology and results are discussed in Section §4. Section §5 discusses related work and Section §6 concludes this study.

2 Background and Motivation

2.1 Storage Driver of Container

Accessing data efficiently within layers is critical for the performance of concurrently running containers (i.e. , starting up containers, updating container images, and storing ephemeral data). Taking popular Docker and Kubernetes for example, as shown in Figure 1, each image consists of multiple read-only lower layers. When creating a new container based on an image, Docker simply creates a new writable upper layer on top of lower layers (we use *upper* and *lower* to represent the upper layer and lower layer, respectively, in the rest of this paper). All of these layers are stored in the native file system. The storage driver mounts them onto a directory, giving a unified file view (i.e. , merged layer) for each container. Each container has its own *upper*, so when reading or writing files in *upper*, *upper* is directly updated. *Lowers* are shared by containers for storage-saving and high-speed startup. Therefore, when updating files in *lowers*, storage driver leverages CoW to redirect modifications in *lowers* to the *upper*. Specifically, (1) write operations will first copy the file or block to the *upper* of the container, and then modify the copied file or block in the *upper*. (2) Delete operations will create a whiteout file or block in the *upper* to hide the corresponding file or block in the *lower*.

The storage driver is composed of two parts: the additional tier to manage layers and the native file system to store layers. The additional tier can be realized in three ways: Union file system (i.e. , AUFS and OverlayFS), virtual block device (i.e. , device mapper), and native file system which supports snapshot (i.e. , BtrFS). OverlayFS and AUFS do not store data directly on disk, they only redirect files to provide isolated views for each container. Device mapper leverages the thin provision and snapshot capabilities of the kernel-based Device Mapper framework to manage image data in EXT4. BtrFS is a CoW-based file system. Layers are stored and managed by the snapshot function. File data and file system metadata are stored in CoW optimized B-trees. As for CoW granularity, UnionFS uses file-level CoW, while BtrFS

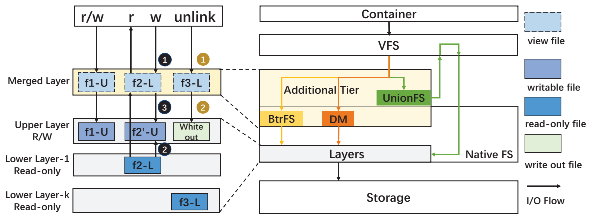


Figure 1. Operations of containers. 'f1-U' is a file in the *upper*, 'f2-L' is a file in the *lower*.

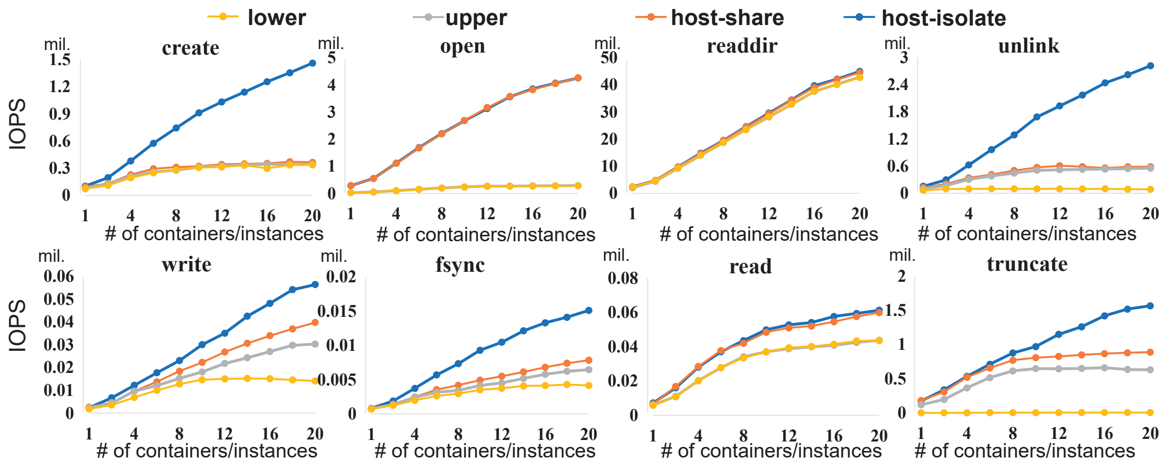


Figure 2. Results of the evaluated operations. The x-axis represents the number of containers or instances. The y-axis represents the throughput of host storage. The details of experimental setup are to be presented in Section IV.

Table 1. Evaluated operations. These operations are performed on files in different layers.

Type	Operation	Mode	Type	Operation	Mode
META	Open	READ	DATA	Read	READ
	Readdir	READ		Write	WRITE
	Create	WRITE		Fsync	WRITE
	Unlink	DELETE		Truncate	DELETE

and device mapper use block-level CoW. Overlay2/EXT4 is recommended officially [25] and deployed widely because of its high performance and simple implementation [6]. Note that a container can also attach a data volume to store data. Data volume is a directory in a native file system. Compared to accessing data in layers, a container can access data in data volume at a higher speed because a data volume can be accessed without the additional tier. However, the speed is still affected by resource competition in native file systems.

2.2 Performance Bottlenecks Caused by Storage Driver

High-speed SSDs can effectively mitigate the performance gap between back-end storage and upper system, which makes overhead of storage driver more obvious than before. In what follows, through a set of experiments, we first show the performance bottlenecks of storage drivers with the officially recommended plan, Overlay2/EXT4. We identify that the major bottlenecks are due to file redirect in OverlayFS and shared resources in the native file system.

2.2.1 Experimental Setup. We evaluate the overall throughput when performing basic file system operations on different layers (i.e. , *upper* and *lower*) with the number of concurrently running containers/instances varies from 1 to 20. Note that it is common to run 20 containers on a single server[26]. The physical node has 20 cores which in turn prevents any possible CPU contention in our experiments. The basic file system operations are generated by a modified FXMARK benchmark suite [3] which can issue operations to different layers of a container with a single thread. As shown in Table 1, operations are performed on both data and metadata, because the operations on data (i.e. , read) need to obtain the file descriptor in advance, the overhead of open operation will be considered as well. For the purpose of comparison, we carry out two more groups of experiments in which multiple single-threaded original FXMARK instances are running concurrently on a physical host. The number of instances ranges from 1 to 20. In the first group of experiments, operations of different instances are issued to different directories on a shared I/O stack (i.e. , *host-share*). In the second group of experiments, operations of different instances are issued to different directories on isolated I/O stack (i.e. , *host-isolate*). We realize isolated I/O stack by building block devices that have their own VFSs and native file systems. The operations are performed on data with a size of 4KB. Note that the I/O model is *directio*, because we mainly focus on optimizing the I/O performance of the storage driver. We plan to explore page cache management in future work.

2.2.2 Performance Bottleneck Analysis. Based on Figure 2, we have two main observations. First, operations (open, unlink, write, fsync, read, and truncate) show better overall throughput as the number of containers increases when issued in *host-share* compared to when issued in layers of containers (*upper* and *lower*). This means file redirect of OverlayFS introduces extra overhead. Second, by comparing the results of *host-share* with those of *host-isolate*, we find that the throughput of operations (create, unlink, write, fsync, and truncate) when issued in layers is also limited by the shared native file system.

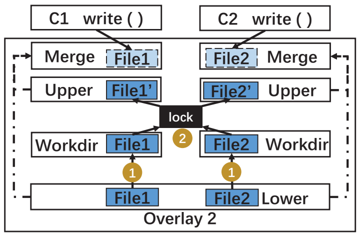


Figure 3. CoW from two containers

Time-Consuming File Redirect of OverlayFS: *Lower* shows lower throughput for unlink, write, fsync, and truncate operations compared to that of *upper*. The performance degradation is due to CoW. Modifications cannot be made in *lower*s, because *lower*s are read-only. OverlayFS uses CoW to redirect the modifications of *lower*s. As shown in Figure 3, ① when a container modifies a file in *lower*, OverlayFS copies the file to a *workdir*. Then, ② OverlayFS renames the file to *upper* to hide the corresponding file in *lower*. As a result, the modification is redirected to the *upper*. Rename operation of VFS in step ② needs a global lock to prevent deadlock as we will discuss in Section 3.2.2, so file redirect which involves CoW is executed serially. Note that *workdir* is used in OverlayFS for consistency purposes. Files in *workdir* will be removed when files are completely updated to the *upper*.

By comparing the throughput of *upper* and *host-share*, we find that OverlayFS introduces extra overhead to open, write, fsync, read, and truncate. When doing the mentioned operations in a container, VFS needs to search in all layers to open the file due to building view dentries that store the mapping relationship between file paths and exact files on disk. As shown in Figure 4, searching dentries occupy most of the time (90%) of open operations in *upper* when 20 containers are running concurrently. We take opening file B in *upper/a* as an example. As shown in Figure 5(a), to open file B in traditional environments, VFS needs to search three dentries, ordered: dentries of *upper*, *upper/a*, *upper/a/B*. While in a container (Figure 5(b)), these three dentries (dentries of *merged*, *merged/a*, *merged/a/B*) have no corresponding file or directory on disk and VFS cannot find these dentries. Accordingly,

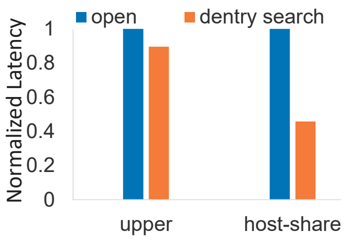
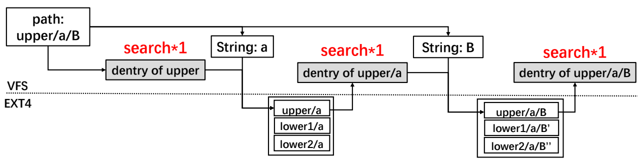
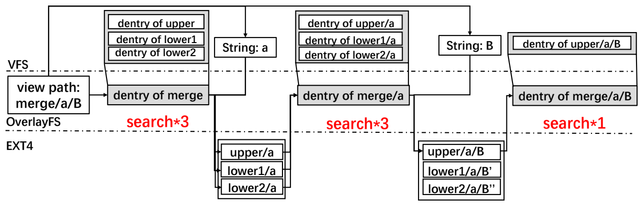


Figure 4. Normalized latency of open operations and dentry search under "upper" and "host-share". Normalized latency refers to the ratio of the latency to that of open operations.



(a) Dentry search of VFS



(b) Dentry search of OverlayFS

Figure 5. File redirect

OverlayFS cooperates with VFS to build these dentries. In order to build a dentry of *merged*, OverlayFS first traverses all layers and finds all directories (*upper*, *lower1*, and *lower2*) that form *merged*. Next OverlayFS turns to VFS to search dentries of these three directories and combine them as the dentry of *merged*. And then OverlayFS builds dentries of *merged/a* (*upper/a*, *lower1/a*, and *lower2/a*) and *merged/a/B* (*upper/a/B*). In summary, VFS needs to search seven dentries before opening *merged/a/B*. However, only three dentries (*upper*, *upper/a*, and *upper/a/B*) are really needed to open file B. Note that opening files in *lower* also experience this overhead. Create, readdir, and unlink in *upper* show no performance degradation compared to those in *host-share*, because those metadata operations do not open files.

Observation 1: VFS is unaware of the layers and cannot cooperate efficiently with OverlayFS, which makes file redirect of OverlayFS time-consuming. First, VFS needs to find extra real dentries to help OverlayFS to build view dentries. Second, CoW caused by modifying files in *lower* leads to severe global lock contention in VFS.

Competition Caused by Shared Resources in Native File System: By comparing the results of *host-share* and *host-isolate*, we find that create, unlink, write, fsync, and truncate, which are involved with modifying data, suffer from throughput degradation due to resource competition. Resource competition in *host-share* is mainly due to shared block allocation and journal service[4]. Traditional native file systems, i.e. , EXT4, XFS, split the storage space into many BGs. Each BG has its own inode bitmap, inode table, and data blocks. Operations from different *upper*s may be performed in the same BG because the native file system is unaware of containers, and block allocation is serialized to reduce the seek time of hard disk driver. Consequently, operations from different containers may compete for locks to get the data structures of a BG.

Centralized journal service causes contention. Native file system runs a JBD2 (*journal block device*) to record the updates before writing data to the destination. First, JBD2 groups updates into one *running transaction*, a double linked list. Second, when updates in the *running transaction* reach the threshold, the *running transaction* is transformed into a *committing transaction* and begins to write the updates to storage. Then, a new *running transaction* is created to receive new updates. In order to ensure the consistency of updates, there is only one *running transaction* and one *committing transaction* at a time for JBD2 [24], and list locks and state locks are required when multiple containers request to access the running transaction as shown in Figure 6. Furthermore, the model of system-wide journal service cannot be customized in the granularity of container. In *host-isolate*, with the number of containers increasing, the overall throughput increases almost linearly for all operations because each file system manages its own resources. However, assigning an exclusive file system to each container is hard and this complicates layer sharing between containers.

Observation 2: The native file system is unaware of the existence of containers. When containers need to use shared file system resources (i.e. , BGs, journal service) simultaneously, they suffer from lock contention.

3 CAST: Container-Aware I/O Stack

In a containerized environment, VFS and the native file system overlook layers causing time-consuming file redirect (i.e. , extra dentry search during file open and lock contention due to CoW) and intense resource competition (i.e. , BGs and journal service). In an effort to improve overall throughput, we design CAST which is composed of LaVFS and CaFS. LaVFS carries out layer-aware path switch and simultaneous CoWs to reduce the overhead caused by file redirect in OverlayFS. The container-aware native file system, CaFS, provides fine-grain resource (i.e. , BGs and journal service) allocation to reduce resource competition. Hereafter, we first summarize the system overview and then focus on the design details.

3.1 Overview of CAST

CAST is designed with the following goals in mind:

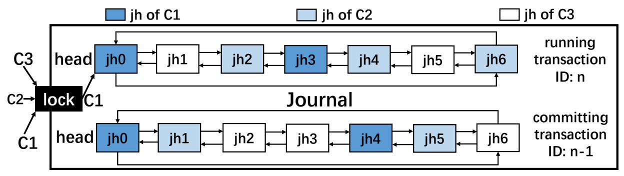


Figure 6. Running and committing transactions. C: container, ID: the number of transaction, jh: journal_head to store an update

- **Reduce search overhead:** While opening a file in a container, OverlayFS must redirect the view file to its real file which exists in *upper* or *lower*, thus consuming extra dentry search compared to the native file system. CAST aims to reduce search overhead by directly locating the real file based on layer information.
- **Avoid Copy-on-Write wait:** Modifications in *lowers* are redirected to *upper* by CoW operations which are serial due to the global lock. CAST tries to eliminate the impact of the global lock by enabling simultaneous CoWs.
- **Prevent block group sharing:** Centralized resource allocation of the native file system makes data from different containers share BGs. To reduce resource competition which are caused by BG sharing, CAST groups BGs based on containers.
- **Reduce journal contention:** Shared journal service makes updates from different containers need to compete for one running transaction. In order to reduce journal competition among containers, CAST provides journal service to each container individually, based on its needs.
- **Identify layers and containers:** Original VFS and the native file system cannot distinguish layers and containers. CAST maintains this information to enable container-aware scheduling.

Figure 7 shows the overview of CAST. To achieve the above goals, first, we design a *layer-aware VFS*, named LaVFS, which reduces redirect overhead by employing layer-aware path switch and simultaneous CoWs. The path switch can switch view path to real path based on layer information and find real dentry based on the real path like if the file is opened in a traditional environment. In order to enable simultaneous CoWs, LaVFS realizes a lock-free rename based on the fact that rename operations from *lower*s to *upper*s do not cause deadlocks like rename operations in a traditional environment. Second, we realize a container-aware native file system, named CaFS, to reduce resource competition among containers. Specifically, CaFS groups BGs into *Container Regions* (CRs) and carries out block allocation of each container individually (within a specific CR) to prevent BG sharing. A two-level mapping table is designed to maintain

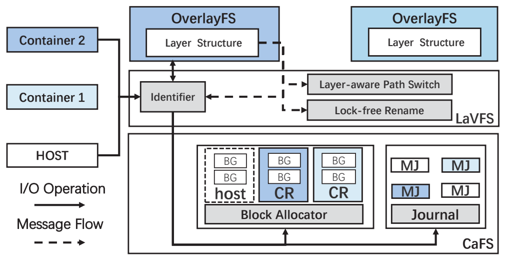


Figure 7. Overview of CAST

the relationship among containers, CRs, and BGs at low cost. CaFS also divides the file system-wide journal service into micro journals, so updates can be collected and submitted from containers independently.

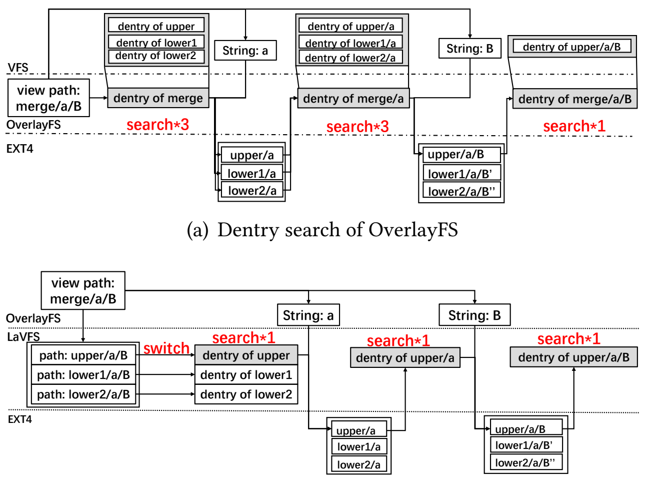
LaVFS gets the information of layer structure from OverlayFS. And CaFS allocates containers based on *upper*s, because actual write operations which involve block allocation and journaling of each container happen only in its *upper*.

CAST works as follows: When an operation arrives, LaVFS identifies the source of the operation. For an operation from a physical host, LaVFS directly sends it to CaFS. If the operation needs a new block, CaFS carries out block allocation in its own region. If the operation involves modifications, updates are submitted to its corresponding micro journal. For an operation from a container, LaVFS cooperates with OverlayFS to redirect the operation to the target file. Specifically, LaVFS carries out layer-aware path switch for an open operation and lock-free rename for a file modification in *lower* based on layer structure provided by OverlayFS to reduce the overhead of file redirect. After file redirect, the operation is sent to its corresponding CR on CaFS.

3.2 Layer-Aware VFS

OverlayFS causes performance degradation due to file redirect. In what follows, we introduce *Layer-aware VFS* (LaVFS) which can cooperate with OverlayFS efficiently to reduce the overhead of file redirect.

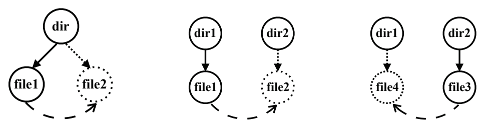
3.2.1 Layer-Aware Path Switch. Analyses in Section 2.2.2 reveal that VFS cannot distinguish a view file and needs to search extra real dentries to build view dentries before opening the view file. We decouple file open and view dentry build. First, LaVFS switches the view path of the file to be opened to the real path so that the file can be opened like it is



(a) Dentry search of OverlayFS

(b) Dentry search of LaVFS

Figure 8. Optimization of file redirect



(a) Case 1 (b) Case 2 (c) Case 3

Figure 9. Rename in different cases

in a traditional environment. Second, OverlayFS builds and fills view dentries asynchronously to hide the overheads.

LaVFS realizes layer-aware path switch. From the perspective of the host, we can switch the view path to the real path by replacing the path of *merged* with the path of the exact layer where the file is located. We explain how LaVFS works with OverlayFS using an example shown in Figure 8(b). When we want to open file B in *upper*, LaVFS first gets the layer structure of the container (i.e. , dentries of layers and sequence of layers) from OverlayFS as we will discuss in Secion 3.4. Second, LaVFS switches the view path to the real path by replacing the path of merged layer with the path of a layer from the top to the bottom. As the latest version of file B is in the higher layer, LaVFS can open file B with the fewest tries. Suppose that an image has K layers, a file is in k^{th} layer, and the file path has L levels from the perspective of a container. Original VFS needs to search $K{*}L{+}k$ dentries before opening the file, while LaVFS needs to search $k{*}L{+}k$ dentries, which has (K-k)*L fewer search times than the original VFS. This means that LaVFS shows good performance when a container has many layers and the file is located in high layers. Moreover, in the worst case, the number of dentries that need to be searched under LaVFS and VFS are the same when file open operations happen in the deepest layer.

Before path switch, all empty view dentries are built by OverlayFS based on the view path. And view dentries are filled asynchronously to accelerate file open in the adjacent path that may come soon. OverlayFS starts a search thread and creates a search list to hold real dentry search requests. When OverlayFS needs to fill a view dentry, first, OverlayFS generates real dentry search requests belonging to the view dentry as before and puts these requests to the search list. Second, the search thread fetches dentry search requests from the search list and informs LaVFS to find these real dentries. The results are put in the corresponding view dentry structure. When all real dentries belonging to the same view dentry are found, the view dentry can be used for accelerating file open.

3.2.2 Simultaneous Copy-on-Write Operations. CoW, which is realized based on rename operations, causes performance degradation due to the contention of file system-wide lock. A rename operation needs to modify the source and destination directories of the file, and the inode mutex (i.e. ,

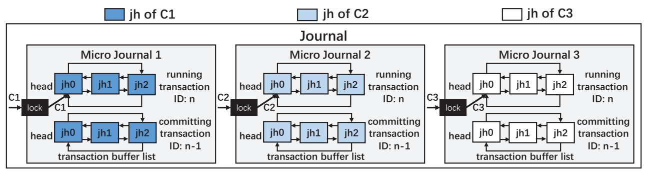


Figure 10. Micro journal architecture

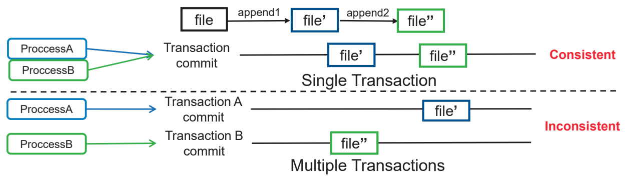
i_mutex) of the two directories must be available. As shown in Figure 9(a), if the source and destination are the same, only one *i_mutex* needs to be obtained. However, if the source and destination are different like in Figure 9(b), *i_mutex*s of two different directories must be obtained in order (i.e. , *dir1* to *dir2*). When *case 2* and *case 3*, shown in Figure 9(b) and Figure 9(c) happen at the same time, the locking order of the two operations is opposite, which may cause deadlock. Therefore, *Linux* provides a file system-wide lock to a rename operation to prevent deadlock. As a result, when containers are carrying out CoWs concurrently, lock contention happens due to rename operations.

However, the global lock of rename is unnecessary for CoW operations, because renames of CoW operations do not cause deadlock. We use *unlink* in *lower* as an example. When deleting a file in *lower*, a whiteout file will be created in a *workdir* (i.e. , source) and then the whiteout file is renamed to *upper* (i.e. , destination) to hide the corresponding file in *lower*. There is no deadlock, because each container has its own *workdir* and *upper*, and the locking order is always from *workdir* to *upper*. Accordingly, we provide a lock-free rename to enable simultaneous CoWs. The lock-free rename is realized in LaVFS. It is the same as the original rename except that it will not try to acquire the file system-wide lock before obtaining the *i_mutex* of two directories. This design does not modify the original locking mechanism, so it has no effect on other operations that need rename operation.

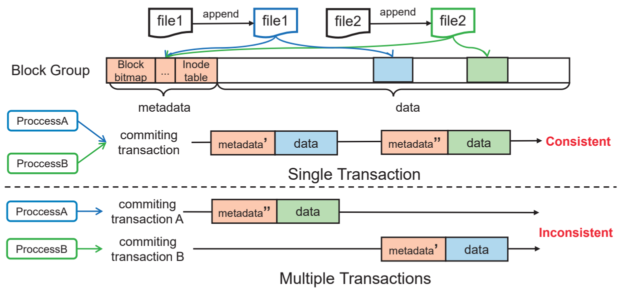
3.3 Container-Aware Native File System

Shared resources (i.e. , BG and journal service) among containers in the native file system cause performance degradation. Accordingly, we isolate shared resources based on containers. First, we divide journal service into several micro journals that can be dynamically bound to containers and independently handle transactions shown in Figure 10. Second, we allocate blocks based on containers. Note that we realize a container-aware file system based on EXT4. Other native file systems, i.e. , EXT3, XFS, that organize blocks based on BG can also leverage container-aware block allocation.

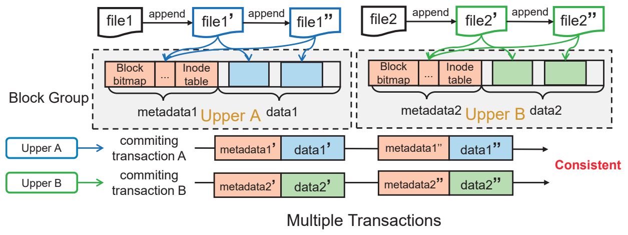
3.3.1 Container-Aware Micro Journals. We need to solve two issues to realize concurrent micro journals. First, running multiple micro journals may cause consistency problems as concurrent transactions may hold different versions of the same data. As shown in Figure 11(a), when two containers are modifying the same file, the modifications may be



(a) Data



(b) Metadata



(c) New design

Figure 11. Consistency problems when multiple transactions are committing concurrently

submitted to different transactions. As the transactions are submitted in parallel, the sequence of the submission cannot be guaranteed which may cause inconsistency if the system crashes while handling the transactions. Second, updates of file system metadata must be submitted together with their corresponding updates of data as shown in Figure 11(b). As block allocation of the file system is serial, files of different containers may be located in the same BG and share file system metadata. Consequently, inconsistency happens, because concurrent transactions may hold different versions of the same metadata.

The unique feature of the layered storage driver (i.e. , updates of a container happen only in its own *upper*) makes concurrently submitting multiple transactions possible as shown in Figure 11(c). First, if we group updates of containers based on *upper*s and bound *upper*s to micro journals, we can guarantee that different versions of the same container data are collected by the same micro journal. Second, if we can allocate BGs based on *upper*s as we will describe in Section 3.3.2, we can guarantee that updates of shared metadata are submitted to the same micro journal. Note that updates from the physical host are all issued to a micro journal.

In order to enable micro journals, data structure in memory and region in disk of JBD2 are reconstructed. *Journal_t* structure is the key data structure. First, it manages updates using shared transaction lists (i.e. , running transaction list, committing transaction list, and checkpoint transaction list), locks, and on-disk regions. Second, functions operate data structures through *journal_t*. We create *m_journal_t* which is alike *journal_t*. It has its own transaction lists, locks, and on-disk regions. We also modify the functions so that they can operate micro journals like operating the original journal. *Mjournal_t*s are organized with a hash table (*mjournal_table*) in *journal_t* so that functions can access *mjournal_t* through *journal_t*. The journal region on disk is divided into *N* (equals to the number of micro journals) sub-regions to store updates of different micro journals. The start address and size of sub-region are recorded in its corresponding *mjournal_t*. Each micro journal can log updates like the original journal service. When CaFS needs to recover data after a crash, the journal thread gets the address of micro journals on disk and recovers the micro journals one by one. Furthermore, we also realize *sysfs_api* for each micro journal like the original journal so that the configurations of each micro journal can be adjusted independently through writing *sys*.

Micro journals are flexible. On the one hand, the number of micro journals can be adjusted. We start *N* (i.e. , the number of CPUs) micro journals by default. The first micro journal logs updates of physical host and other *N-1* logs updates from containers. The number of micro journals can be adjusted through */sys* when the system is free. CaFS flushes data in journal region, redivides region on disk, and updates *mjournal_table* to add micro journals. On the other hand, a container can be dynamically bound to a micro journal. We use a table to record the mapping relationships between containers and micro journals. In order to prevent consistency problems, bounding a container to multiple micro journals is forbidden. Containers with few I/O requests can be bound to the same micro journal. Containers that do not have data recovery requirements can choose not to bound a micro journal. The table is updated when a container is built or destroyed.

3.3.2 Container-Aware Block Allocation. The key idea behind container-aware block allocation is that operations of different containers should be issued to different BGs. Accordingly, first, CaFS maintains the mapping relationships between BGs and containers at a low cost. Second, CaFS allocates blocks of BGs to containers in parallel.

Recording the relationship between BGs and containers with a hash table may introduce extra cost, because the table needs a global lock and all containers need to compete for it when they need to add or delete BGs. Accordingly, we introduce a new structure *Container Region* (CR) which consists of multiple BGs. Thus, BG allocation of each container can be handled independently. As shown in Figure 12, we

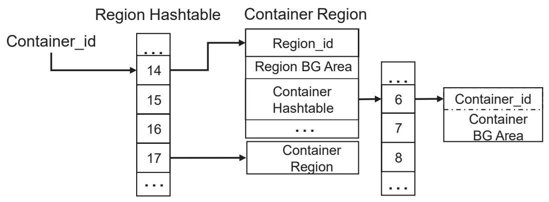


Figure 12. Two-level mapping table

design a two-level mapping table to maintain the mapping relationships among containers, CRs, and BGs. The first level is a region hash table which records mapping relationships between containers and CRs. When a container is created, the container is bound to a region. The second level is a container hash table which records the mapping relationships between BGs and containers. BG Area maintains basic information of the region, such as the number of free BGs. All the BGs in the container hash table are free at the beginning. When a container needs blocks, CaFS selects a proper BG in the CR and allocates blocks to the container on this BG. The BG selection strategy and block allocation algorithm in CR are the same as the original EXT4. After the allocation, CaFS updates the BG entries in the hash table. Region hash table is protected by a system-wide lock. CaFS needs to acquire this lock when a container is created or destroyed. As these operations are not frequent (79% of containers are alive for more than 10 seconds [26]), the lock will cause negligible lock contention. The container hash table is protected by a local lock. When a container needs to get a new BG, it needs to get the container hash table lock of the CR. Note that in the original EXT4, a global orphan list is used to maintain inodes that have no owner (orphan inodes). For CaFS, each CR maintains its own orphan list to reduce lock contention.

The container-aware block allocation works as follows: As shown in Algorithm 1, when a container, $container_i$, is created, CaFS searches region hash table, CR_{set}, for region with enough BG which is marked as CR_i. Otherwise, CaFS will create a new region with n BGs, $CR_{new}(n)$ for the container. When a block allocation $request$ comes, CaFS judges the source of the $request$. If the $request$ is from a physical host, CaFS carries out block allocation like the original EXT4 in the BGs that are not grouped into CRs. If the $request$ is from $container_i$, block allocation is within CR_i. CaFS searches proper BG, PBG, in container hash table. If there is a BG belonging to $container_i$ with enough blocks for the request, or it is just a new BG, BG_{new}, CaFS allocates the blocks of this BG to the $request$, and updates hash table entries to map BG_{new} to $container_i$. When there is no enough free BG for CR_i, i.e. , the number of used BGs recorded by Region BG Area is 80% of the number of BGs for CR_i, CaFS expands CR_i. CaFS allocates n new BGs to CR_i in every expansion. When $container_i$ is destroyed, all BGs of $container_i$ will be

Algorithm 1: Container-aware Block Allocation

```
Data: $CR$; $BG$; $CR(free)$: free BGs in CR; $p$: BGs container
      requests; $BG(free)$: free blocks of BG; $m$: blocks container
      requests; $BG_i$: BG set of $container_i$; $rate(CR_i)$: utilization
      rate of $CR_i$; $CR_i(size)$: Number of BGs in $CR_i$
Result: $PBG$: The proper BG
if $Container_i$ $created$ then                    /* container creation */
    for $CR \in CR_{set}$ do
        if $CR(free) > p$ then
            $CR_i = CR$;
    if $no$ $proper$ $CR$ $in$ $CR_{set}$ then
        $CR_i = CR_{new}(n)$;
if $request \in host$ then                          /* block allocation */
    for $BG \notin CR_{set}$ do
        if $BG(free) > m$ then
            $PBG = BG$;
else
    for $BG \in CR_i$ do
        if $BG(free) > m$ $and$ $BG \in BG_i \bigcup BG_{new}$ then
            $PBG = BG$;
if $rate(CR_i) > 80\%$ then                         /* region expansion */
    $CR_i(size) = CR_i(size + n)$;
if $Container_i$ $destroyed$ then               /* container destruction */
    for $BG \in BG_i$ do
        $free(BG)$;
```

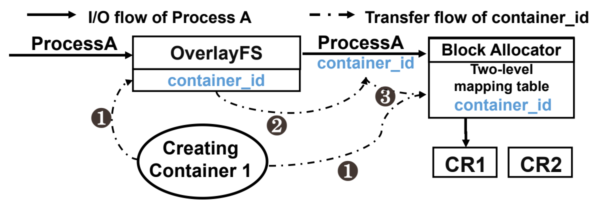


Figure 13. Identification of containers

released. Furthermore, if there is no container in CR_i, the CR_i is released.

3.4 Identification of Containers and Layers

CAST needs to identify layers to enable container-aware management. First, CaFS needs to identify the source (i.e. , container or host) of an operation when it comes. Second, LaVFS needs to get the layer structure information of containers to do layer-aware path switch and lock-free rename.

CAST identifies containers by introducing a new label, i.e. , *container_id*, to OverlayFS, CaFS, and process. We take block allocation as an example, shown in Figure 13. ① We add a *container_id* in the *superblock_info* of the OverlayFS to label which container the OverlayFS belongs to. When an OverlayFS is mounted, it is bound to a container and the *container_id* is set. The *Container_id* is unique as it is generated based on *SHA256* of layers. The *container_id* is set to 0 by default if the operation is from the physical host. CAST also records the relationships between CR and *container_id* in the two-level mapping table of *Block Allocator*. ② In CaFS, operations are managed by processes. So we add *container_id* to *task_struct* to identify which container a process belongs to. When a process issues an operation to LaVFS, LaVFS copies the *container_id* of *superblock_info* to the *container_id* of *task_sturct* to bind the process and container. ③ When an operation is issued to CaFS, *task_struct* finds its corresponding CR based on *container_id*.

CAST identifies layers with the help of the OverlayFS which maintains the layer structure information of its container. We implement functions that can provide dentries of layers and sequence of layers to LaVFS. When an operation arrives at LaVFS, LaVFS checks the *task_struct* of *container_id*. If the operation is from a container, LaVFS further checks which layer it belongs to based on the sequence of layers. If the operations involve file open and are directed to the *merged*, LaVFS gets the sequence of layers and carries out layer-aware path switch to redirect the file. If the operations involve file modifications and are directing to *lowers*, LaVFS provides lock-free rename to OverlayFS to enable simultaneous CoWs.

4 Evaluation

In this section, we evaluate the performance of CAST against the original I/O stack in shared environments. First, we evaluate the performance improvement of each design component with the micro-benchmarks. Second, we evaluate the overall throughput of CAST with real-world workloads.

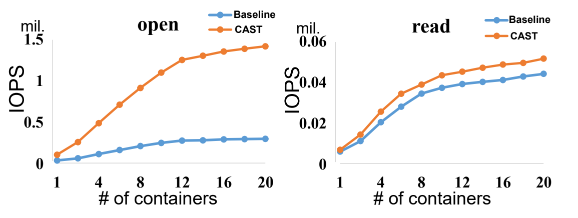
4.1 Evaluation Environment

We implement CAST on Linux 4.16.1 kernel in 3600+ lines of code, in which 2400+ lines of code for CaFS (modifications based on EXT4) and 1200+ lines of code for LaVFS (modifications based on VFS). We carry out our experiment on a machine with Intel Xeon E5-2698 V4 CPU which has 20 cores running at 3.1 GHZ and 64 GB memory. We use a SamSung PM991 NVMe SSD 512GB as a fast back-end storage. The operating system is Ubuntu Server 16.04.1 LTS. Journal service is mounted in ordered journal models. In the experiment, each container has two layers (*upper* and *lower*).

4.2 Analysis of Each Design Component

We evaluate the performance gain of CAST via basic file system operations. Each design component is evaluated separately. As a component cannot work on all types of operations, we choose two operations to evaluate each component.

4.2.1 Layer-Aware Path Switch. Layer-aware path switch can accelerate operations that need to open files. We evaluate it by running open and read in *upper*s separately. As shown in Figure 14, for open operations, CAST achieves 3.1 and 4.8 times higher throughput in comparison to the original I/O



(a) open (b) read

Figure 14. Throughput of open and read in *upper*s while varying number of containers. "Baseline" represents the results of the original I/O stack. CAST only conducts layer-aware path switch.

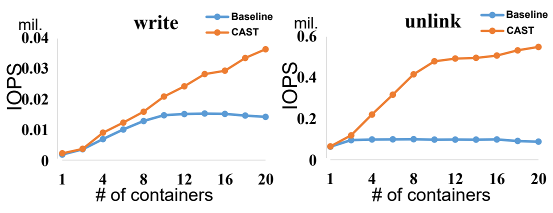
stack for 1 container and 20 containers, respectively. The performance improvement is because that layer-aware path switch can hide the overhead caused by extra dentry search when building view dentries. For read operations, optimizing file open shows 10% and 20% throughput gains for 1 container and 20 containers, respectively, because containers need to open a file before reading data from it. However, the performance gain is not so obvious like open, because reading data from SSD takes more time compared to opening files. We observe that the throughput of open and read increases linearly as the number of containers increases. However, when the number of containers is above 12, the throughput increase gets smaller. The reason is that before doing path switch, empty view dentries must be built in OverlayFS.

4.2.2 Simultaneous Copy-on-Write Operations.

Simultaneous CoWs can accelerate concurrent modifications in *lower*s. We evaluate it by running write and unlink in *lower*s separately. Figure 15 shows the results. For write operations, CAST shows 156% performance improvement for 20 containers. For unlink operations, CAST shows 5.5 times performance improvement for 20 containers. The performance improvement is because that CAST uses a lock-free rename based on the observation that rename operation in CoW will not cause deadlock. Unlink operations show higher performance gain compared to write operations. The reason is that unlink operations do not copy the original file from *lower*s to *upper*s like write operations. They only create whiteout files in *upper*s and the overhead caused by rename lock is more obvious for unlink operations. As a result, unlink operations benefit more from lock-free rename compared to write operations. Moreover, unlike write operation where the throughput increases linearly as the number of containers increases, we find that the throughput increase of unlink operations gets smaller when the number of containers is above 10 under CAST. This is due to writing whiteout files and journal.

4.2.3 Container-Aware Block Allocation.

Container-aware block allocation can reduce BG competition among



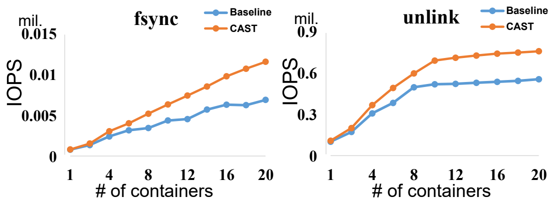
(a) write (b) unlink

Figure 15. Throughput of write and unlink in *lower*s. CAST only conducts simultaneous CoWs.

containers. We evaluate it by running unlink and fsync in *upper*s separately. Figure 16 shows the results. Unlink operations need to modify the metadata of BG, and fsync operations need to modify both data and metadata. For fsync operations, CAST shows 65% improvement compared to the original I/O stack for 20 containers. For unlink operations, CAST brings 37% of performance improvement for 20 containers. Furthermore, fsync operations show an obvious higher performance gain compared to unlink operations. The reason is that fsync operations need to modify both metadata and data, which causes more intense BG competition compared to unlink operations. Furthermore, the throughput of unlink operations scales linearly with the number of containers. But, the throughput increase gets smaller when the number of containers is above 10 under CAST. This is because of journal service and accessing *inode list* in VFS. We mainly focus on reducing the file redirect overhead caused by VFS, and we will isolate *inode list* based on containers in our future work.

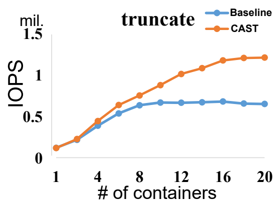
4.2.4 Container-Aware Micro Journals.

Micro journals can reduce journal resource competition among containers by collecting updates based on containers. We use truncate and create to evaluate it shown in Figure 17. For create operations, CAST shows 99% improvement compared to the original I/O stack. For truncate, CAST shows 86% performance improvement for 20 containers compared to the original I/O stack. The throughput of truncate and create operations scales linearly with the number of containers. However, the

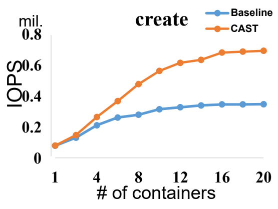


(a) fsync (b) unlink

Figure 16. Throughput of fsync and unlink in *upper*s. CAST only conducts container-aware block allocation.



(a) truncate



(b) create

Figure 17. Throughput of truncate and create in *uppers*. CAST only conducts container-aware micro journals.

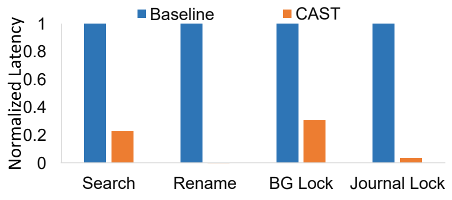
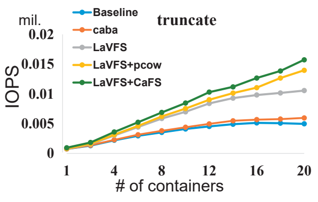
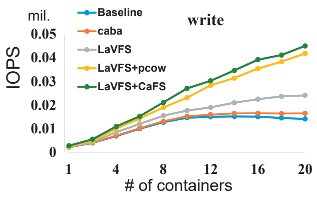


Figure 18. Normalized latency of dentry search, rename, BG lock, and journal lock. Normalized latency refers to the ratio of the latency to that of the original I/O stack. "Search" is the latency caused by dentry search before opening a real file. "Rename" refers to latency of a rename operation. "BG Lock" refers to latency of *s_blockgroup_lock*. "Journal Lock" refers to latency of *j_list_lock*.



(a) truncate



(b) write

Figure 19. Throughput of truncate and write in *lowers*. "Caba" represents container-aware block allocation. "Pcow" represents simultaneous CoWs.

throughput increase of truncate operations gets smaller after 16 containers, because of the performance bottleneck of SSD; and the throughput increase for create operations gets smaller after 12 containers due to accessing *inode list* in VFS.

4.2.5 Stack-up Validation. We evaluate the latency of dentry search, rename operation, BG lock, and journal lock with truncate operation for 20 containers shown in Figure 18. The latency of dentry search, rename operation, BG lock, and journal lock under CAST is reduced by 77%, 99%, 69%, and 96%, respectively, compared to the original I/O stack. This means each component of CAST functions well.

We also choose write and truncate as their performance can be improved by all the four components and carry out stack-up validation by integrating our design components into the original I/O stack one by one shown in Figure 19. When all design components work together, throughput of truncate and write is improved by 216% and 219%, respectively. The overall throughput increases linearly as the number of containers increases. For truncate operations, the original I/O stack achieves 20%, 92%, 68%, and 36% throughput improvement when container-aware block allocation, micro journals, simultaneous CoWs, and layer-aware path switch are integrated into the original I/O stack one by one for 20 containers. For write operations, the throughput improvements of the four components are 17%, 54%, 25%, and 23% for 20 containers. We can observe that container-aware journal and simultaneous CoWs bring the major part of the performance gains.

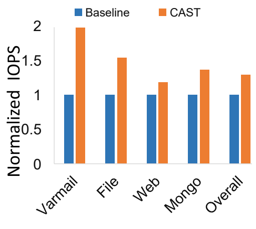
Table 2. Workload specification

Workload	File Size	Read/Write Size	Read/Write ratio
Fileserver	128KB	1MB/16KB	1:2
Varmail	16kb	1MB/16KB	1:1
Webproxy	16KB	1MB/16KB	5:1
MongoDB	16kb	1MB/16KB	1:1

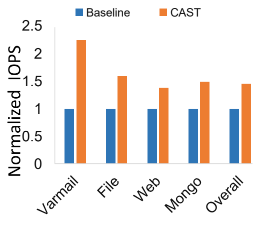
4.3 Overall Storage Driver Performance

To evaluate the performance of CAST in more realistic scenarios, where reads, writes, and metadata operations run simultaneously in multiple containers, we run Fileserver, Varmail, Webproxy, and MongoDB using Filebench [28]. The parameters are shown in Table 2. Referring to the default settings of filebench, we run each instance for 60s. For each instance, 1000 files are created in each layer of a container in advance.

We run different workloads in containers. Figure 20(a) shows the results when 4 containers are running 4 different workloads. The throughput of Varmail, Fileserver, Webproxy, and MongoDB is improved by 79%, 54%, 18%, and 36%, respectively. Varmail operates on small data compared to other workloads, which means Varmail involves more metadata



(a) 4 concurrent containers



(b) 20 concurrent containers

Figure 20. Throughput when multiple Filebench containers are running concurrently. The normalized IOPS refers to the ratio of the IOPS to that of in the original I/O stack.

modifications. As a result, Varmail benefits more from isolating BGs and journal service. For Webproxy which is dominated by read and MongoDB where read operation is very large, resource competition is not high. As a result, the performance gain of these workloads is lower than Varmail and Filesever. Moreover, the overall throughput is improved by 29% under CAST.

Figure 20(b) shows the results when 20 containers are running 4 different workloads (5 containers for each workload). The throughput of Fileserver, Varmail, Webproxy, and MongoDB is improved by 98%, 58%, 38%, and 49%, respectively. And the overall throughput is improved by 45% for CAST. CAST shows higher overall throughput improvement for 20 containers compared to that of 4 containers. The reason is that resource competition is higher for 20 containers and CAST can effectively mitigate resource competition.

5 Related Works

Scalable native file systems: Many researchers [3, 4, 12, 13] have studied and improved the scalability of the native file system by isolating its services. For example, MultiLanes [12] provides independent VFS and native file system for each core and transmits operations to their own VFS and native file system. SpanFS [13] replaces the centralized file system service with a collection of independent micro file system services, called domains. Each domain performs its file system service independently. However, isolating the shared I/O stack in a containerized environment causes longer image download time, because layers of different containers cannot be shared. Min et al. [4] use their open source benchmark suite FXMARK [3] to analyze the scalability of five widely-used file systems. They find that there are many hidden scalability bottlenecks such as locks, ref count, cache-line conflict in file systems. Vijayan et al. [21] conduct an in-depth evaluation on journaling file systems and find that journal service is one of the major reasons for the scalability problem. In order to realize paralleled journal service, Son et al. [24] adopt lock-free data structures and operations using atomic instructions to implement concurrent updates on data structures. The new structure is implemented in *j_checkpoint_mutex* and *j_list_lock* of JBD2. Park et al. [20] propose *ijournal* which can handle updates of *fsync* operations independently. SCALEFS [2] logs operations in a per-core log so that it can delay propagating updates to the disk representation until a fsync is triggered. Kim et al. [14] propose to log updates based on cores to solve scalability problems. However, native file systems cannot assign services based on containers due to the unawareness of layers. Furthermore, new overheads are introduced by the additional tier when managing layers.

Optimizations on Docker storage: CoW cause redundant data and throughput degradation due to extra data write. Accordingly, TotalCOW [31] adds a new cache layer to cache mapping relationships between the block device and the file system to save memory usage due to CoW operations in BtrFS [22]. Guo et al. [9] discover that there are a lot of redundant data in storage drivers due to the unshareable nature of cached data and coarse-grained CoW. They propose HP-Mapper which provides a two-level mapping strategy to support fine-grained CoW. Apart from optimizing CoW, Xu et al. [32] analyze the performance of containerized applications with different storage parameter configurations on the local and remote storage. Vasily et al. [27] do a similar work which focuses on comparing different kinds of storage drivers. These works [9, 27, 31, 32] mainly focus on analyzing the overhead of storage driver for single container. We in contrast study the performance bottleneck when multiple containers are running concurrently. There are also some works [7, 10, 17] that enable remote images. These images read data on-demand from the registry, which means the performance bottleneck is mainly due to network transfer.

I/O schedulers for reducing resource competition: Many researchers have tried to improve overall performance of back-end storage by reducing I/O competition in I/O schedulers. For example, McDaniel et al. [19] address I/O contention challenge in containers and provide a two-tiered approach with schedulers at both cluster level and node level to prevent overall performance degradation. As for SSD, Shen et al. [23] propose to reduce I/O competition by leveraging anticipation. Hao et al. [8] improve I/O throughput by proposing NCQ-aware I/O scheduling method for SSD. Tavakkol et al. [29] propose to fully explore the parallelism of SSD to improve the throughput of SSD. These works are orthogonal to our research as they cannot solve resource competition in native file systems.

6 Conclusions and Future Work

In this paper, we analyze the performance bottlenecks caused by storage driver, and find that the I/O stack of the physical host overlooks the layers of containers causing time-consuming file redirect in OverlayFS and intense resource competition in the native file system. Accordingly, we propose CAST, which consists of LaVFS and CaFS. LaVFS reduces file redirect overhead by leveraging layer information to switch view path to real path and by enabling simultaneous CoWs. Meanwhile, CaFS alleviates resource competition by providing fine-grain resource allocation at the native file system. As future work, we plan to explore the impact of page cache in memory and inode list in VFS on the performance of container storage driver in SSDs.

Acknowledgements

This work was supported by National Science Foundation of China under grants No.62032008 and No.61872155. We also thank Alibaba Group for their support through AIR Project. Hao Fan is the corresponding author.

References

[1] Sungyong Ahn, Kwanghyun La, and Jihong Kim. 2016. Improving I/O Resource Sharing of Linux Cgroup for NVMe SSDs on Multi-core Systems. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'16)*. 1–5.

[2] Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2017. Scaling a File System to Many Cores Using an Operation Log. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'17)*. 69–86. https://doi.org/10.1145/3132747.3132779

[3] Min Changwoo. 2021. FxMark. https://github.com/sslab-gatech/fxmark.

[4] Min Changwoo, Kashyap Sanidhya, Maass Steffen, Kang Woonhak, and Kim Taesoo. 2016. Understanding Manycore Scalability of File Systems. In *Proceedings of the USENIX Annual Technical Conference (ATC'16)*. 71–85.

[5] Google Cloud. 2021. Kubernetes. https://kubernetes.io/.

[6] Docker Docs. 2021. Overlay2. https://docs.docker.com/storage/storagedriver/overlayfs-driver/.

[7] Hao Fan, Shengwei Bian, Song Wu, Song Jiang, Shadi Ibrahim, and Hai Jin. 2021. Gear: Enable Efficient Container Storage and Deployment with a New Image Format. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS'21)*. 115–125. https://doi.org/10.1109/ICDCS51616.2021.00020

[8] Hao Fan, Song Wu, Shadi Ibrahim, Ximing Chen, Hai Jin, Jiang Xiao, and Haibing Guan. 2019. NCQ-Aware I/O Scheduling for Conventional Solid State Drives. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'19)*. 523–532. https://doi.org/10.1109/IPDPS.2019.00062

[9] Fan Guo, Yongkun Li, Min Lv, Yinlong Xu, and John C. S. Lui. 2019. HP-Mapper: A High Performance Storage Driver for Docker Containers. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC'19)*. 325–336. https://doi.org/10.1145/3357223.3362718

[10] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Slacker: Fast Distribution with Lazy Docker Containers. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'16)*. 181–195.

[11] Docker Inc. 2021. Docker. https://www.docker.com/.

[12] Junbin Kang, Benlong Zhang, Tianyu Wo, Chunming Hu, and Jinpeng Huai. 2014. MultiLanes: Providing Virtualized Storage for OS-Level Virtualization on Many Cores. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'14)*. 317–329.

[13] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. 2015. SpanFS: A Scalable File System on Fast Storage Devices. In *Proceedings of the USENIX Annual Technical Conference (ATC'15)*. 249–261.

[14] Jongseok Kim, Cassiano Campes, Joo-Young Hwang, Jinkyu Jeong, and Euiseong Seo. 2021. Z-Journal: Scalable Per-Core Journaling. In *Proceedings of the USENIX Annual Technical Conference (ATC'21)*. 893–906.

[15] Chu Li, Dan Feng, Yu Hua, and Fang Wang. 2016. Improving RAID Performance Using an Endurable SSD Cache. In *Proceedings of the International Conference on Parallel Processing (ICPP'16)*. 396–405. https://doi.org/10.1109/ICPP.2016.52

[16] Feifei Li. 2019. Cloud-Native Database Systems at Alibaba: Opportunities and Challenges. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2263–2272. https://doi.org/10.14778/3352063.3352141

[17] Huiba Li, Yifan Yuan, Rui Du, Kai Ma, Lanzheng Liu, and Windsor Hsu. 2020. DADI: Block-Level Image Service for Agile and Elastic Application Deployment. In *Proceedings of the USENIX Annual Technical Conference (ATC'20)*. 727–740.

[18] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. 2021. Max: A Multicore-Accelerated File System for Flash Storage. In *Proceedings of the USENIX Annual Technical Conference (ATC'21)*. 877–891.

[19] Sean McDaniel, Stephen Herbein, and Michela Taufer. 2015. A Two-Tiered Approach to I/O Quality of Service in Docker Containers. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'15)*. 490–491. https://doi.org/10.1109/CLUSTER.2015.77

[20] Daejun Park and Dongkun Shin. 2017. iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call. In *Proceedings of the USENIX Annual Technical Conference (ATC'17)*. 787–798.

[21] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2005. Analysis and Evolution of Journaling File Systems. In *Proceedings of the USENIX Annual Technical Conference (ATC'05)*. 196–215.

[22] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-tree Filesystem. *ACM Transactions on Storage (TOS)* 9, 3 (2013), 1–32. https://doi.org/10.1145/2501620.2501623

[23] Kai Shen and Stan Park. 2013. FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs. In *Proceedings of the USENIX Annual Technical Conference (ATC'13)*. 67–78.

[24] Yongseok Son, Sunggon Kim, Heon Y. Yeom, and Hyuck Han. 2018. High-Performance Transaction Processing in Journaling File Systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'18)*. 227–240.

[25] Yu Sun, Jiaxin Lei, Seunghee Shin, and Hui Lu. 2020. Baoverlay: A Block-Accessible Overlay File System for Fast and Efficient Container Storage. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC'20)*. 90–104. https://doi.org/10.1145/3419111.3421291

[26] Sysdig. 2021. 2021 Container Security and Usage Report. https://sysdig.com/wp-content/uploads/2021-container-security-and-usage-report.pdf.

[27] Vasily Tarasov, Lukas Rupprecht, Dimitris Skourtis, Amit Warke, Dean Hildebrand, Mohamed Mohamed, Nagapramod Mandagere, Wenji Li, Raju Rangaswami, and Ming Zhao. 2017. In Search of the Ideal Storage Configuration for Docker Containers. In *Proceedings of the IEEE International Workshops on Foundations and Applications of Self* Systems (FAS*W'17)*. 199–206.

[28] Vasily Tarasov, Erez Zadok, and Spencer Shepler. 2016. Filebench: A Flexible Framework for File System Benchmarking. *USENIX login* 41, 1 (2016), 6–12.

[29] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gómez-Luna, and Onur Mutlu. 2018. FLIN: Enabling Fairness and Enhancing Performance in Modern NVMe Solid State Drives. In *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'18)*. 397–410. https://doi.org/10.1109/ISCA.2018.00041

[30] Charles P. Wright, Jay Dave, Puja Gupta, Harikesavan Krishnan, David P. Quigley, Erez Zadok, and Mohammad Nayyer Zubair. 2006. Versatility and Unix Semantics in Namespace Unification. *ACM Transactions on Storage (TOS)* 2, 1 (2006), 74–105. https://doi.org/10.1145/1138041.1138045

[31] Xingbo Wu, Wenguang Wang, and Song Jiang. 2015. Totalcow: Unleash the Power of Copy-on-Write for Thin-Provisioned Containers. In *Proceedings of the Asia-Pacific Workshop on Systems (APSys'15)*. 1–7. https://doi.org/10.1145/2797022.2797024

[32] Qiumin Xu, Manu Awasthi, Krishna T. Malladi, Janki Bhimani, Jingpei Yang, and Murali Annavaram. 2017. Performance Analysis of Containerized Applications on Local and Remote Storage. In *Proceedings of the International Conference on Massive Storage Systems and Technology (MSST'17)*. 1–12.