Nesting Overlay File Systems with ShadowWhiteout

Kenta Ishiguro
Grenoble INP - UGA

Kohei Hayama
Hosei University

Ayase Yokoyama
Hosei University

Hiroshi Yamada
TUAT

Toshio Hirotsu
Hosei University

Abstract

Container technology improves storage efficiency by sharing base images through overlay file systems (overlayfs). Overlayfs enables temporary modifications while preserving original files by merging multiple directories into a single unified view. However, in container-in-container environments, the current overlayfs approach incurs inefficiency in terms of storage usage, as it does not support nesting due to file deletion operations. To delete a preserved original file, a special file is created indicating that the file is deleted in the overlayfs, but this mechanism lacks expressiveness for nesting. This paper introduces ShadowWhiteout, a new representation for file deletion in overlayfs. We identify key technical challenges and implement a ShadowWhiteout-capable overlayfs. Our system allows nesting overlayfs while maintaining reasonable runtime overhead.

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

Keywords: Containers, Overlay File Systems

ACM Reference Format:
Kenta Ishiguro, Kohei Hayama, Ayase Yokoyama, Hiroshi Yamada, and Toshio Hirotsu. 2025. Nesting Overlay File Systems with ShadowWhiteout. In *16th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '25), October 12–13, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3725783.3764408

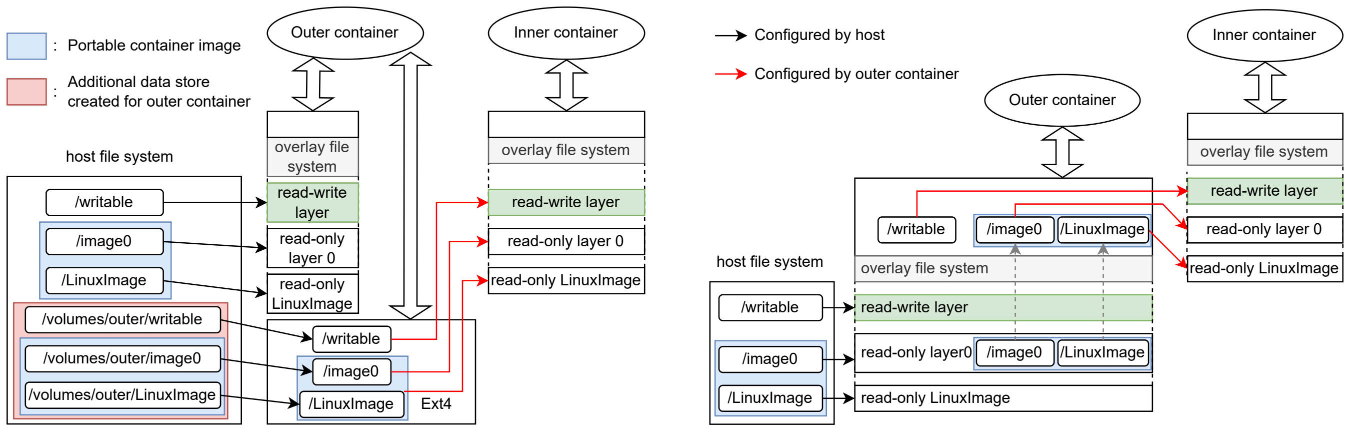
1 Introduction

The overlay file system (overlayfs) [2] is now widely used for container environments [8], enabling the compact construction of container images. Container engines use overlayfs to present a unified file system view as a virtual root file system to the container instance. This root file system combines a series of read-only layers (files and directories) with a single read-write layer. Any changes made by the container instances are stored in the read-write layer using a file granularity copy-on-write approach, as deltas from the underlying read-only layers. This allows container instances to have individual file systems while sharing a large portion of the base container images, optimizing storage efficiency that contributes to a higher degree of container consolidation.

However, due to file deletion operations in overlayfs, it does not support nesting because it cannot use another overlayfs for its read-write layer. This limitation incurs storage inefficiency in container-in-container (CinC) environments, which are becoming increasingly popular. CinC setups, such as Docker-in-Docker (DinD) [12] and Kubernetes-in-Docker (KinD) [14], are in high demand due to their many use cases in continuous integration and testing of container-based development systems, among others [11, 13]. Fig. 1a shows the current CinC setup. The container engine creates file systems other than overlayfs (e.g., Ext4 [9]) in the outer container instance at launch by mounting a directory on the host through Docker volumes, and all data associated with the inner containers is stored in this additional data store. Since the inner container images are stored outside the outer container images, this approach leads to storage inefficiency by preventing the sharing of inner container images across multiple outer container instances.

To deal with this issue, we introduce ShadowWhiteout (S-whiteout for short). The original overlayfs performs the deletion operation by creating a special file called *whiteout* in the read-write layer, which hides the deleted file from the unified view. Unfortunately, the design of the whiteout file type does not account for nesting, preventing each overlayfs layer from independently creating its own dedicated whiteout file (§ 2.1). In contrast, S-whiteout functions as a whiteout file for a specific overlayfs layer while acting as a normal file for other layers. We design the S-whiteout file type by building upon xwhiteout [10], an extended whiteout type specifically designed for read-only layers (§ 2.2), by addressing two key aspects: 1) identifying the file operations in a read-write layer overlayfs that must support the creation of S-whiteout files instead of normal whiteout files and 2) selecting an appropriate file type to facilitate efficient cache management for directory reads in overlayfs. We design and implement an S-whiteout capable overlayfs (§ 3) to achieve the CinC setup shown in Fig. 1b while keeping reasonable performance overhead (§ 4).

ACM ISBN 979-8-4007-1572-3/2025/10…$15.00
https://doi.org/10.1145/3725783.3764408



(a) CinC environments expose host file systems to allow inner containers to use overlayfs. This additional data store is created for each outer container instance at launch time. The images of inner containers cannot be included in the portable outer container image.

(b) Our goal is to enable inner containers to use overlayfs on the overlayfs mount for outer containers in CinC environments. This will allow the images of inner containers to be included in a portable and sharable image of the outer container.

Figure 1. File system construction for containers in CinC environments. The arrows in this figure indicate which directory corresponds to each layer in the overlayfs mount.

2 Challenges to nest overlayfs

This section describes the challenges of making overlayfs nestable through the architecture of overlayfs.

2.1 Overlayfs

Fig. 2 illustrates how overlayfs presents a unified view by combining multiple layers. On the left side of Fig. 2, a typical usage of overlayfs is presented, which combines three directories from the host file system (e.g., Ext4). In this configuration, `/image0` and `/image1` serve as read-only layers, while `/writable` serves as the read-write layer. Since all changes in overlayfs are stored in the read-write layer, the read-write layer is positioned at the top among all layers while the read-only layers can consist of multiple layers. In overlayfs terminology, the read-only layers are referred to as lower file systems, and the read-write layer is referred to as an upper file system. To avoid confusion, we use the terms "read-only layers" and "read-write layer" in this paper.

Lookup. When an object (e.g., a file) with the same name exists in multiple layers, the version in the topmost layer is visible in the unified view. For example, `file0` in the read-only layer 1 and `file1` in the read-only layer 0 are visible in the unified view on the left side of Fig. 2 because they are the files in the topmost layer for this overlayfs mount.

Read a directory. When a `readdir` request is made on a directory in the unified view, overlayfs reads the directory from all layers and creates a combined name list from the results of all layers. This combined name list is cached until the file descriptor for the directory is closed. In the left overlayfs mount shown in Fig. 2, once the root directory is read, four files are cached in the name list.

Modification. When creating a new file in the unified view, overlayfs creates the file in the read-write layer. If a file in one of the read-only layers is modified in the unified view, a file granularity copy-on-write operation is performed. On the left side of Fig. 2, when the container modifies `file2`, the file is first copied into the read-write layer from the read-only layer 0, and then the contents of `file2` in the read-write layer are updated accordingly.

Remove. Removing a file that exists only in the read-write layer from the unified view is straightforward; overlayfs simply removes the file from the read-write layer. However, if a file with the same name exists in one or more read-only layers, overlayfs needs to create a whiteout file in the read-write layer to hide the file from the unified view. The whiteout file in the read-write layer is represented as a character file with 0/0 device number. During file lookups, if the topmost file is a whiteout file, it becomes invisible in the unified view. For example, if `file3` is a whiteout file in the read-write layer (as shown on the left side in Fig. 2), it will be hidden in the unified view, even though `file3` exists in the read-only layer 0.

Challenge 1: Fixed type of whiteout file is not suitable for nesting overlayfs. As mentioned, whiteout files are represented as a specific character file in the read-write layers. However, this binary information (i.e., whether a given file is a whiteout or not) lacks the expressiveness needed for nesting overlayfs. The right side of Fig. 2 illustrates how `file3` appears in the unified view of the top overlayfs when the underlying overlayfs is used as the read-write layer. Since `file3` is a whiteout file, it is hidden from the unified view of the underlying overlayfs. Consequently, when the top

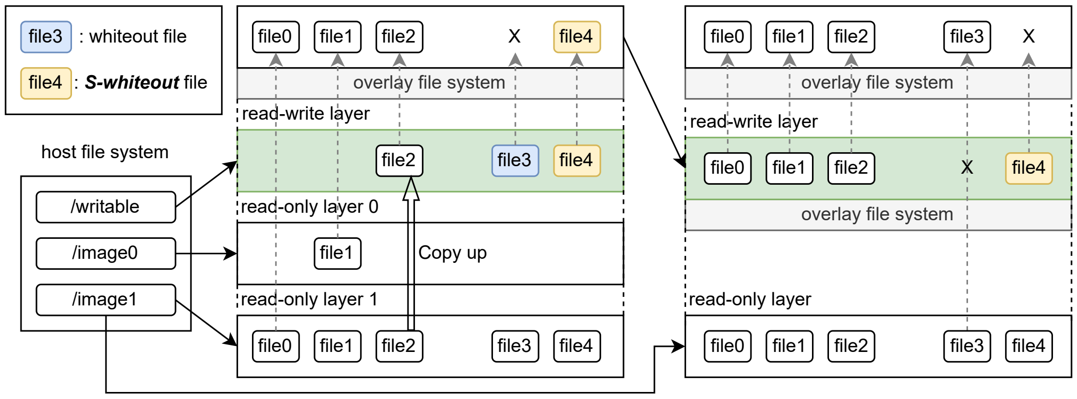


Figure 2. Overview of overlayfs architecture. The left side is a typical setup of overlayfs. The right side shows the case of nesting overlayfs. The top overlayfs uses the underlying overlayfs as its read-write layer. `file3`, a normal whiteout, is hidden by the underlying overlayfs resulting in `file3` in the readonly layer is visible to the top overlayfs. `file4`, an S-whiteout, is visible to the top overlayfs and hidden by it in the unified view of the top overlayfs.

overlayfs looks up `file3`, it finds it in the read-only layer but not in the read-write layer, causing `file3` from the read-only layer to appear in the top overlayfs. Even if the top overlayfs creates a whiteout file, it remains invisible because it is interpreted by the underlying overlayfs. In this example, `file3` cannot be removed from the top overlayfs.

Overlayfs attribute. Overlayfs uses extended attributes that start with "overlay." to manage metadata by associating these attributes with files and directories. For example, overlayfs uses an "opaque" attribute for directories. When a directory is created after removing a directory of the same name that exists in one of the read-only layers, overlayfs sets the "overlay.opaque" attribute to "y" for the newly created directory. This information is crucial because overlayfs needs to determine whether to look up the read-only layers. If the value of "opaque" is "y", overlayfs only needs to look up the read-write layer; if "opaque" is not set for the directory, overlayfs needs to combine results from all layers.

Overlayfs for container environments. The overlayfs architecture aligns well with container image construction. One benefit of using containers to build applications is that each container has its own root file system, which enhances reproducibility. While storage usage could become inefficient if every container included all files in its root file system, containers avoid the inefficiency by sharing a base image, for example, a Linux distribution like Ubuntu image. To achieve both objectives simultaneously, overlayfs maintains base images as immutable by keeping them in read-only layers, while allowing containers to update their content at runtime by saving changes in a dedicated read-write layer for each container. In academic research, container storage has been studied mainly for performance [1, 4, 6, 15–22], but our work focuses on extending the capabilities of overlayfs.

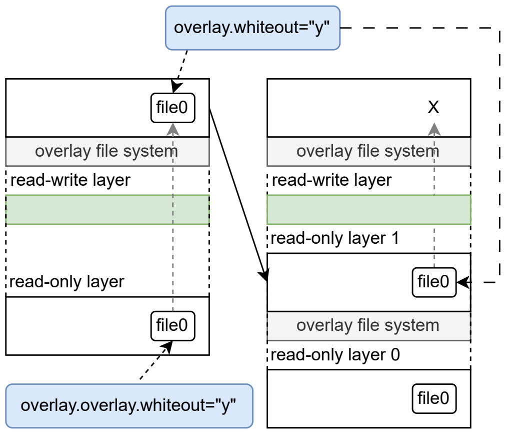


Figure 3. Extended whiteout (xwhiteout) and escaping overlayfs extended attributes.

2.2 Extended whiteout for overlayfs

Although overlayfs did not initially assume the use case of nesting overlayfs, it now supports being used as a **read-only layer** for another overlayfs [10], driven by use cases like making overlayfs mount on ComposeFS [5]. To enable this usage, two major changes have been introduced into the mainline Linux kernel. The first major change involves escaping overlayfs extended attributes. The extended attributes that start with "overlay." are handled by the underlying overlayfs because they only contain the binary information whether those attributes are for overlayfs or not, similar to the whiteout scenario. To address this issue, overlayfs uses a special prefix of "overlay.overlay." instead of "overlay". This prefix allows the attributes to be interpreted by a specific layer of overlayfs, with the layer being determined by the number of prefixes used. Fig. 3 shows how overlayfs handles the overlayfs extended attribute. The underlying overlayfs identifies that the attribute starts with "overlay." and removes the first "overlay." from the "overlay.overlay.whiteout". The top overlayfs also recognizes that the attribute starts with "overlay." and interprets it as "overlay.whiteout" (regarding whiteout attribute described later) because there are no additional prefixes in the attribute.

The second major change is the introduction of extended whiteout (xwhiteout for short) for read-only layers. Unlike the original whiteout file (which is represented as a character file), xwhiteout is represented as a zero-size regular file with the attribute "overlay.whiteout". By combining this approach with the escaping of overlayfs extended attributes, xwhiteout files can be interepreted by a specific layer of overlayfs based on the number of prefixes in the attribute. For example, in Fig. 3, `file0` can work as a whiteout file for the top overlayfs while being treated as a regular file for the underlying overlayfs. The underlying overlayfs handles `file0` as a regular file because its attribute is "overlay.overlay.whiteout". In contrast, for the top overlayfs, the attribute of `file0` is "overlay.whiteout", as the underlying overlayfs removes one "overlay" from its attribute. Consequently, `file0` is recognized as a whiteout file by the top overlayfs.

Even though xwhiteout is available instead of the original whiteout, overlayfs is still not usable as a read-write layer for another overlayfs. This limitation arises because xwhiteout was designed to enable overlayfs to work as a read-only layer for another overlayfs. The two challenges remain in applying the xwhiteout design to nesting overlayfs by using another overlayfs for the read-write layers.

Challenge 2: No file operations in overlayfs create xwhiteout files. Unlike the normal whiteout file, xwhiteout files are never created by overlayfs. Overlayfs assumes that xwhiteout files are created during the construction of read-only layers by userspace tools or may be manually created by maintainers as needed. In contrast, when overlayfs is used as a read-write layer for another overlayfs, the top overlayfs attempts to create whiteout files in various scenarios at runtime. For example, if the inner container deletes a file while it is still retained in the outer container, the nestable overlayfs needs to indicate that the file is invisible to the inner container but visible to the outer container. It must provide a means to handle such situations at runtime without resorting to the creation of the normal whiteout files.

Challenge 3: Cache management of name lists during directory reads incurs a performance overhead. To avoid the overhead of checking xwhiteout files during directory reads, overlayfs employs two optimizations. The first optimization limits the directories that require xwhiteout checks by setting the extended attribute "overlay.opaque" to "x" on directories containing xwhiteout files. With this information, overlayfs only performs xwhiteout checks when reading those specific directories. Note that this setting can also be applied manually by userspace tools or maintainers. The second optimization defers the check for xwhiteout files until the directory is actually read. However, this can unintentionally block the removal of a directory if a file is hidden by an xwhiteout file. During the removal of a directory, overlayfs reconstructs the caches of files in the name list for that directory without checking xwhiteout files. As a result, when determining if the directory is empty, the presence of the xwhiteout file in the name list leads overlayfs to incorrectly conclude that the directory is not empty. To avoid this issue, overlayfs needs to check for xwhiteout files for every regular file during cache construction. However, this check introduces significant overhead, especially if the directory contains many files. As shown in Fig. 5, the execution time of the `ls dir` command is 6.6× slower when a single xwhiteout file is present in a directory containing 800,000 regular files.

3 ShadowWhiteout-capable overlayfs

We introduce ShadowWhiteout (S-whiteout) to enable overlayfs to use another overlayfs for its read-write layer by addressing the following three challenges described in § 2.

- Challenge 1: Fixed type of whiteout file is not suitable for nesting overlayfs.
- Challenge 2: No file operations in overlayfs create xwhiteout files.
- Challenge 3: Cache management of name lists during directory reads incurs a performance overhead.

In summary, S-whiteout is a character file that includes the extended attribute "overlayfs.whiteout". This file type combines the advantages of both normal whiteouts and xwhiteouts. Our S-whiteout-capable overlayfs creates S-whiteout files instead of normal whiteout files in three file operations. As a result, the top overlayfs can remove files from its unified view by creating S-whiteout files in its read-write layer, as demonstrated by `file4` in Fig. 2, even when it uses another overlayfs mount for the read-write layer.

Using the extended attributes to identify whiteout files. To address Challenge 1, we adopt the concept of the xwhiteout file. Although xwhiteout files were originally proposed for read-only layers, the nestable extended attribute is effective for read-write layers as well.

Three file operations support the creation of S-whiteout. To address Challenge 2, we identify which file operations in overlayfs need to support the creation of S-whiteout files through unit tests and real-world workloads (in § 4). Based on this analysis, three file operations are deemed necessary for support. First, when the top overlayfs removes a file using whiteout files, it creates an S-whiteout file instead of a normal whiteout file. Since the top overlayfs can determine whether its read-write layer is an overlayfs through the `dentry`, it simply creates an S-whiteout file upon identification.

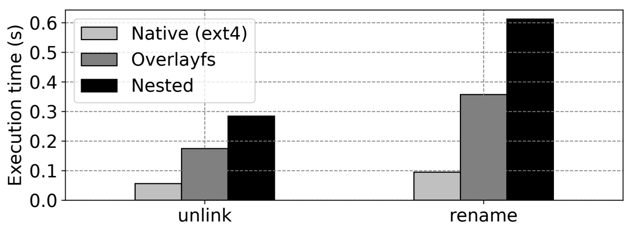


Figure 4. Execution time of `unlink` and `rename`.

Second, the underlying overlayfs needs to support rename whiteout requests. The top overlayfs issues this request when renaming a file that exists in the read-only layer to hide the source file name from its unified view. In response, the underlying overlayfs creates an S-whiteout file with the same name as the source file.

Third, the underlying overlayfs needs to support requests to create special files. Some userspace tools directly create normal whiteout files by using `mknod` during image construction. When the underlying overlayfs receives this request, it checks whether the special file corresponds to a normal whiteout file; if so, it creates an S-whiteout file instead of a normal whiteout file.

Representing S-whiteout as a character file for efficient cache management when reading directories. To address Challenge 3, we use a character file for S-whiteout representation, unlike xwhiteout. Xwhiteout files are represented as size-zero regular files, which require optimizations such as deferring checks during directory reads. In contrast, checks for normal whiteout files are not deferred in overlayfs thanks to their file type. For this reason, S-whiteout inherits the file type of normal whiteout files.

4 Experimental results

We evaluate the performance overhead associated with S-whiteout and nesting of overlayfs in CinC environments. We implement S-whiteout-capable overlayfs on Linux kernel 6.7.11. For this experiment, we execute workloads on a server featuring two 48-core Intel Xeon Platinum 8468 CPUs, 512 GB of RAM, and 800 GB of KIOXIA CD8 mixed use NVMe SSD and we use Ext4 for the root file system.

4.1 Microbenchmark: `unlink` and `rename`

In our microbenchmark, we measure execution time of `unlink` and `rename` for 8,192 files, comparing Ext4, single overlayfs, and nested overlayfs. We use the original overlayfs for the single overlayfs setup and use S-whiteout to nest overlayfs. Fig. 4 shows the execution time for each operation. Adding a single overlayfs increases execution time by 0.12 and 0.11 seconds for `unlink`, and by 0.26 seconds for `rename`. The overhead from S-whiteout is negligible compared to the impact of adding one overlayfs layer.

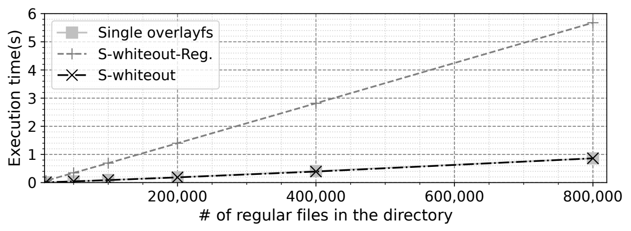


Figure 5. Execution time of `ls dir` with different number of regular files in a directory that has one whiteout file.

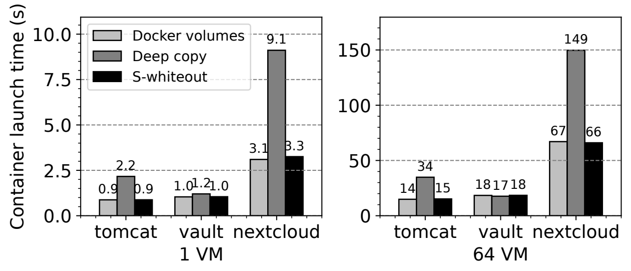


Figure 6. Container launch time in Docker-in-Docker.

4.2 Performance impact on directory reads

To evaluate the effectiveness of S-whiteout against Challenge 3, we measure the execution time of the command `ls dir` with varying numbers of files in the directory. The target directory contains one removed file, indicating the presence of a single whiteout file in the read-write layer. We compare the performance of nested overlayfs with S-whiteout to that of native overlayfs and nested overlayfs with S-whiteout-regular, which uses a regular file instead of a character file to represent S-whiteout. Fig. 5 shows nesting overlayfs with S-whiteout does not incur significant overhead, whereas nesting overlayfs with S-whiteout-regular incurs a worst-case overhead of 6.6×. This overhead arises from the need to check whiteout files for cache construction. When using S-whiteout-regular, overlayfs must perform a whiteout check for every regular file, leading to increased costs as the number of files in the directory grows.

4.3 Containers launch time in Docker-in-Docker

To see the performance impact of nesting overlayfs in CinC environments, we measure the launch time of Docker containers in a Docker container by porting HelloBench [7] with three images selected based on their popularity by the previous study [6]. We compare nesting overlayfs with S-whiteout to two different setups. The one setup uses Docker volumes, allowing the inner container to access the exposed Ext4. The other setup uses deep copy, which does not require the volumes but necessitates copying every image to launch inner containers due to the absence of overlayfs in the container. As shown in Fig. 6, nesting overlayfs with S-whiteout does

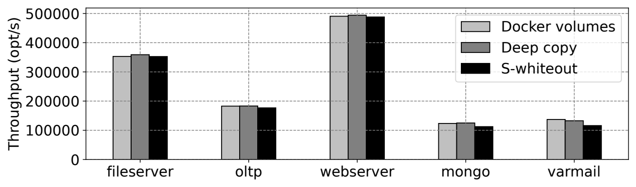


Figure 7. Filebench performance in Docker-in-Docker.

not incur significant overhead thanks to sharing container images, while the deep copy setup incurs overhead of up to 2.75× due to many file copies.

4.4 Filebench performance in Docker-in-Docker

To evaluate the I/O performance of inner containers, we use several workloads from Filebench. We adjust the number of files based on previous studies [3, 6]. The file counts are 200,000 for `fileserver`, 440 for `oltp`, 1.25 M for `webserver`, and 250,000 for both `mongo` and `varmail`. We observe a performance overhead of up to 15% in `mongo` and `varmail` when using S-whiteout. Since only the S-whiteout setup uses nested overlayfs, `read`, `write`, and `unlink` incur overhead due to the additional recursive operation introduced by nesting overlayfs compared to other setups. In the remaining workloads, the impact of synchronization operations (e.g., locks) outweighs that of the recursive operations, making the performance overhead from nesting overlayfs negligible.

5 Discussion

Use cases of CinC and S-whiteout capable overlayfs. One of the primary use cases of CinC setups is testing systems or applications that use container technology. For instance, KinD enables users to test Kubernetes in Docker, while DinD allows users to test Dockerized applications in Docker. However, the original overlayfs does not support nesting, which means that outer Docker containers must reserve a read-write data store (i.e., volumes) outside of read-only images for inner container engines. As a result, images for inner containers are not included in portable outer container images (Fig. 1a). S-whiteout capable overlayfs addresses this issue by allowing nesting, thereby enabling the inclusion of inner container images in portable outer container images (Fig. 1b).

Security. S-whiteout capable overlayfs inherits the security model of the original overlayfs. Since any modifications made by containers to the unified view of the original overlayfs are stored in the read-write layer on the host, the host has visibility into all container files. Conversely, containers are only able to see files that are exposed by the host. When nesting overlayfs for inner containers, the outer containers have greater capabilities than the inner containers, and the host retains the highest level of capability, still having greater visibility and control than the outer containers.

6 Conclusion

This paper presents ShadowWhiteout for optimizing storage efficiency in container-in-container environments by nesting overlayfs. By examining the architecture of overlayfs, we identify three challenges associated with nesting overlayfs and demonstrate how S-whiteout-capable overlayfs addresses them.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was supported in part by Japan Science and Technology Agency (JST CREST JPMJCR21M4).

References

[1] Janki Bhimani, Zhengyu Yang, Ningfang Mi, Jingpei Yang, Qiumin Xu, Manu Awasthi, Rajinikanth Pandurangan, and Vijay Balakrishnan. 2018. Docker Container Scheduler for I/O Intensive Applications Running on NVMe SSDs. *IEEE Transactions on Multi-Scale Computing Systems* 4, 3 (2018), 313–326. https://doi.org/10.1109/TMSCS.2018.2801281
[2] Neil Brown. [n. d.]. *Overlay Filesystem.* https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt
[3] Kyu-Jin Cho, Jaewon Choi, Hyungjoon Kwon, and Jin-Soo Kim. 2024. RFUSE: Modernizing Userspace Filesystem Framework through Scalable Kernel-Userspace Communication. In *22nd USENIX Conference on File and Storage Technologies (FAST 24).* USENIX Association, Santa Clara, CA, 141–157. https://www.usenix.org/conference/fast24/presentation/cho
[4] Kihan Choi, Hyungseok Seo, Hyuck Han, Minsoo Ryu, and Sooyong Kang. 2023. CredsCache: Making OverlayFS scalable for containerized services. *Future Generation Computer Systems* 147 (2023), 44–58. https://doi.org/10.1016/j.future.2023.04.027
[5] composefs. 2023. *composefs: The reliability of disk images, the flexibility of files.* https://github.com/composefs/composefs
[6] 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* (Santa Cruz, CA, USA) *(SoCC '19).* Association for Computing Machinery, New York, NY, USA, 325–336. https://doi.org/10.1145/3357223.3362718
[7] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Slacker: Fast Distribution with Lazy Docker Containers. In *14th USENIX Conference on File and Storage Technologies (FAST 16).* USENIX Association, Santa Clara, CA, 181–195.
[8] Docker Inc. 2025. *OverlayFS storage driver.* https://docs.docker.com/engine/storage/drivers/overlayfs-driver/
[9] Aneesh Kumar KV, Mingming Cao, Jose R Santos, and Andreas Dilger. 2008. Ext4 block and inode allocator improvements. In *Linux Symposium*, Vol. 1.
[10] Alexander Larsson. 2023. *Support nested overlayfs mounts with xattrs and whiteous.* https://lore.kernel.org/all/cover.1694512044.git.alexl@redhat.com/
[11] Chris Maki and Rodny Malina. 2023. Docker-in-Docker: Containerized CI Workflows. https://www.docker.com/resources/docker-in-docker-containerized-ci-workflows-dockercon-2023/. dockercon 23.
[12] Jerome Petazzoni. [n. d.]. *DinD.* https://github.com/moby/moby/blob/master/hack/dind
[13] Alessandro Sassi, Chris Jensen, and Richard Mortier. 2025. Reckoning Kubernetes at the Edge using Emulated Clusters. In *Proceedings of the 3rd International Workshop on Testing Distributed Internet of Things Systems* (Rotterdam, Netherlands) *(TDIS '25).* Association for

Computing Machinery, New York, NY, USA, 7–12. https://doi.org/10.1145/3719159.3721222

[14] Kubernetes SIGs. [n. d.]. *kind*. https://kind.sigs.k8s.io/

[15] 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 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 90–104. https://doi.org/10.1145/3419111.3421291

[16] 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 *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. 199–206. https://doi.org/10.1109/FAS-W.2017.148

[17] Song Wu, Zhuo Huang, Pengfei Chen, Hao Fan, Shadi Ibrahim, and Hai Jin. 2022. 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* (Virtual, Switzerland) *(VEE 2022)*. Association for Computing Machinery, New York, NY, USA, 18–30. https://doi.org/10.1145/3516807.3516818

[18] Xingbo Wu, Wenguang Wang, and Song Jiang. 2015. TotalCOW: Unleash the Power of Copy-On-Write for Thin-provisioned Containers. In *Proceedings of the 6th Asia-Pacific Workshop on Systems* (Tokyo, Japan) *(APSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 15, 7 pages. https://doi.org/10.1145/2797022.2797024

[19] Qiumin Xu, Manu Awasthi, Krishna T. Malladi, Janki Bhimani, Jingpei Yang, and Murali Annavaram. 2017. Docker characterization on high performance SSDs. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 133–134. https://doi.org/10.1109/ISPASS.2017.7975282

[20] Qiumin Xu, Manu Awasthi, Krishna T Malladi, Janki Bhimani, Jingpei Yang, Murali Annavaram, and Ming Hsieh. 2017. Performance analysis of containerized applications on local and remote storage. In *Proceedings of the Symposium on Mass Storage Systems and Technologies (MSST'17)*, Vol. 3. 24–28.

[21] Frank Zhao, Kevin Xu, and Randy Shain. 2016. Improving copy-on-write performance in container storage drivers. In *Proceedings of the Storage Developer Conference (SDC'16)*.

[22] Nannan Zhao, Muhui Lin, Hadeel Albahar, Arnab K. Paul, Zhijie Huan, Subil Abraham, Keren Chen, Vasily Tarasov, Dimitrios Skourtis, Ali Anwar, and Ali Butt. 2024. An End-to-end High-performance Deduplication Scheme for Docker Registries and Docker Container Storage Systems. *ACM Trans. Storage* 20, 3, Article 18 (June 2024), 35 pages. https://doi.org/10.1145/3643819