BAOVERLAY: A Block-Accessible Overlay File System for Fast and Efficient Container Storage

Yu Sun, Jiaxin Lei, Seunghee Shin, Hui Lu
Binghamton University
{ysun59,jlei23,sshin,huilu}@binghamton.edu

ABSTRACT

Container storage commonly relies on overlay file systems to interpose read-only container images upon backing file systems. While being transparent to and compatible with most existing backing file systems, the overlay file-system approach imposes nontrivial I/O overhead to containerized applications, especially for writes: To write a file originating from a read-only container image, the whole file will be copied to a separate, writable storage layer, resulting in long write latency and inefficient use of container storage. In this paper, we present BAOverlay, a *lightweight, block-accessible* overlay file system: Equipped with a new block-accessibility attribute, BAOverlay not only exploits the benefit of using an asynchronous copy-on-write mechanism for fast file updates but also enables a new file format for efficient use of container storage space. We have developed a prototype of BAOverlay upon Linux Ext4. Our evaluation with both micro-benchmarks and real-world applications demonstrates the effectiveness of BAOverlay with improved write performance and on-demand container storage usage.

CCS CONCEPTS

• Information systems → Storage virtualization; **• Software and its engineering → File systems management**.

KEYWORDS

Virtualization, Containers, Overlay File Systems

ACM Reference Format:
Yu Sun, Jiaxin Lei, Seunghee Shin, Hui Lu. 2020. BAOVERLAY: A Block-Accessible Overlay File System for Fast and Efficient Container Storage. In *ACM Symposium on Cloud Computing (SoCC '20), October 19–21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3419111.3421291

1 INTRODUCTION

As an alternative to virtual machines (VM), containers [10, 24, 57] offer a much lightweight, operating system (OS) level virtualization approach, leading to higher server consolidation density and lower operational cost in cloud data centers [9]. Further, containers allow developers to easily pack and ship an application as a self-sufficient container image, able to be deployed and run virtually anywhere, making it extremely *portable* in a standardized and repeatable manner [52].

A container image is compactly built with a series of *read-only* layers (files and directories) – with each being a set of deltas from the layer before it [14]. Latching a layered container image onto backing file systems (e.g., Linux Ext3/4) to make it serve as the (virtual) root file system of a container instance needs additional effort: Container storage typically repurposes an *overlay* approach [59], wherein a shim layer (i.e., an overlay file system) sits upon backing file systems to combine all layers of a container image and presents a unified file-system view to the container instance.

While being transparent and compatible with most existing backing file systems, the overlay file-system approach, yet another level of indirection, does impose nontrivial I/O overhead. Most noticeably, a mechanism commonly used in practical overlay file systems [15, 19, 42] – making the read-only layers of container images immutable – is a simple (yet costly) copy-on-write (CoW) operation: To write a file in a read-only container image layer, a copy of the *whole* file must be first made to a separate, per-instance writable layer.

Such a CoW mechanism makes practical sense, as an overlay file system, working upon backing file systems, simply relies on their exposed *file-level* I/O interface and hence intuitively works at a *file* granularity. However, the file-based CoW mechanism negatively impacts the performance of containerized applications, as a write, even to a tiny portion of a file, might trigger a long "read-and-then-write" I/O operation. Our investigation demonstrates that due to the file-based CoW, the performance (queries per second) of a containerized MySQL database decreases by up to 50%; it also leads to slow startup time – up to 28 seconds to start a containerized MongoDB. This is precisely why the existing "best practice" of

SoCC '20, October 19–21, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8137-6/20/10…$15.00
https://doi.org/10.1145/3419111.3421291

containers suggests storing read-write data outside read-only container images (e.g., via volumes [16]). However, distinguishing types of file data and storing them separately incur considerable management/configuration effort and greatly impair containers' portability.

To this end, we introduce BAOverlay, a *lightweight, block-accessible* overlay file system for fast and efficient container storage. The key idea of BAOverlay lies in providing fine-grained accessibility – at the *block* level – to overlay files. With block-accessibility, BAOverlay further enables a fast CoW operation and efficient use of container storage space.

Specifically, to enable block-accessibility, BAOverlay logically partitions an overlay file into a sequence of equally-sized blocks. Thus, a CoW to an overlay file only involves copying a number of blocks – where the write targets – rather than the whole file. To further hide the read latency in a slow, synchronous CoW operation, BAOverlay decomposes it into two portions: a *fast-and-synchronous* write (putting only the new updates into the writable layer) and a *slow-and-asynchronous* read (copying necessary data from the read-only layers). This division hides any latency of the read in a CoW from containerized applications, which only wait for the completion of the write. As a write is usually to local memory (e.g., page cache), it is extremely fast. Additionally, BAOverlay provides a new file format to compactly store a sparse overlay file – allocation of storage space is delayed until blocks of an overlay file are being updated – and explores various mapping mechanisms to strike a good balance between performance and space efficiency.

We know of no other container storage techniques supporting block-accessibility for fast I/O and efficient use of storage space. BAOverlay materializes the above ideas by interposing new features and functions upon a clearly-defined file-system interface – the inode data structure in POSIX-compatible file systems. It does not alter any semantic-level activities of backing file systems and thus can be interposed upon a variety of existing file systems. Our evaluation of BAOverlay with Linux Ext4 as the backing file system using micro-benchmarks and real-world applications shows that BAOverlay significantly improves applications' I/O performance (e.g., 32x for a 1KB file, and 64x for a 4MB file), and saves storage space (the file size grows only as needed).

Road map: Section 2 motivates the problem, followed by related work (Section 3), detailed design (Section 4), implementation (Section 5), and empirical evaluation (Section 6). Section 7 concludes the paper.

2 MOTIVATION

2.1 Container Storage

One driving factor for the popularity of containers lies in *portability* – the ability to move an application from one host environment to another, seamlessly. With containers, one can build an application in the form of a container image and execute it on any host supporting containers. Compared to VMs, the portability provided by containers is much lightweight, as a container image contains minimally what are needed to run a containerized application – e.g., its binary code, libraries, and storage data – without the need of including a bulky OS kernel (e.g., VMs).

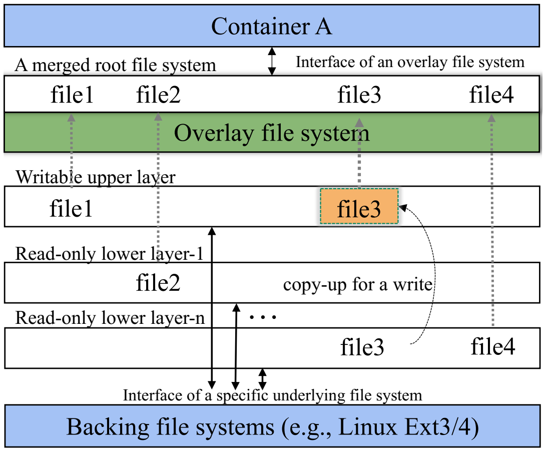


Figure 1: An overlay file system combines all layers of a container image and presents a unified virtual file-system view to a container instance.

Container Image: A container image provides a reproducible virtual file-system environment that a containerized application runs within. As practically driven by Docker [24], a container image is made up of a series of *immutable* read-only layers, with each storing a hierarchy of files and directories. Being immutable, layers can be shared by an unlimited number of images, resulting in saved storage space and fast creation of new container images. For instance, one can create a new container image of a web service application in a Linux environment by creating a new layer that contains the code and data of the web application and composing it with an existing Ubuntu image (i.e., layers).

Overlay File System: To provision a container instance from a specific container image, the layered image needs to be mounted and served as the (virtual) root file system of the container instance. In this way, a container instance can execute with a different file system from the host's (i.e., a lightweight isolation in data storage). Container storage commonly leverages an overlay approach [15, 42, 59] to achieve this, which interposes an overlay (or union[1]) file system upon backing file systems. In general, an overlay file system combines multiple sub-directories – stored on the same or

[1] Note that, in container terminologies, overlay file system (overlayFS) sometimes refers to one implementation of union file systems. Yet, we use overlay to generally refer to union file systems [55].

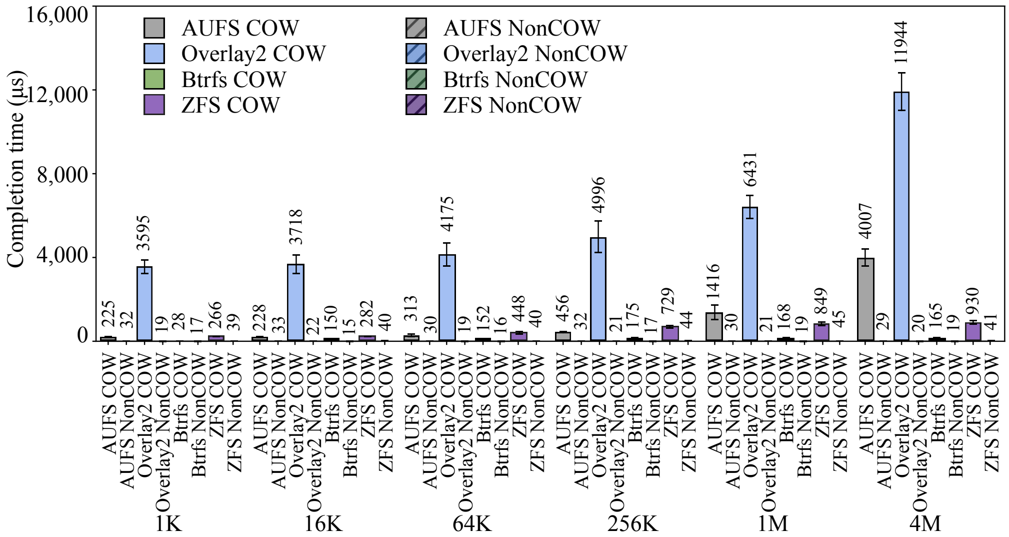


Figure 2: Completion time of an update to lower-layer files of different sizes under various container storage approaches (e.g., AUFS, Overlay2, Btrfs, and ZFS).

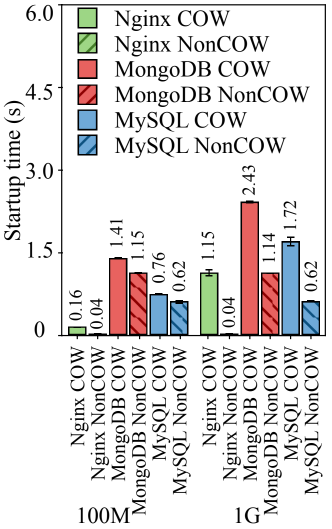


Figure 3: Startup time of containerized applications.

different backing file system(s) and organized in a particular stacking order – into a single merged sub-directory: When the name of an object (e.g., a file) exists in multiple sub-directories, the one in the topmost sub-directory is visible.

An overlay file system can be straightforwardly applied to mount a layered container image. As depicted in Figure 1, during the creation of a container instance, a container storage driver exposes its container image to an overlay file system in a way that each image layer represents a *read-only* sub-directory stacked in the same order as that in the container image (referred to as *lower* layers). In addition, an initially empty *writable* sub-directory is stacked at the top (referred to as the *upper* layer). The overlay file system then presents them as a single root file system to the container instance. During runtime, for a *read*, the overlay file system searches from the upper to the lower layers in sequence until the target file is first found; the path information is then cached to accelerate future accesses. For a *write*, if it is the first time to a file in the lower, read-only layer, a costly copy-up operation (i.e., copy-on-write) is performed to copy the *full* file from the lower layer to the upper layer. All subsequent updates are then applied to the new copy of the file. Therefore, all changes made to the container instance (e.g., writing new files, modifying existing files, and deleting files) are confined within the upper, per-instance writable layer.

2.2 Overlay Overhead Illustration

To illustrate the overhead incurred in a multilayered overlay architecture (in Figure 1), we first compare the I/O latency, in terms of the completion time of a write I/O operation – to lower-layer files of various sizes – issued by a containerized application. More specifically, we ran Docker container [24] (version 19.03) on a Linux host machine (with kernel 5.3). The container storage was layered upon the Linux Ext4 file system backed by an NVMe SSD (Samsung 970 EVO). We ran a simple containerized application that interacted with the underlying container storage with a series of I/Os: It opened a file (of various sizes), updated a tiny portion of the file (i.e., one byte), and then closed the file. We considered two types of updates: 1) copy-on-write (CoW) – the file was stored in the lower layer and accessed for the first time by the application and 2) non-CoW – the file resided in the upper layer. We measured the total completion time of the above three consecutive operations. We ran each case for ∼100 times to take latency variations into account.

Write Latency: We investigated two state of the art overlay file systems – Overlay2 [15] and AUFS [19]. As plotted in Figure 2, under the CoW case with Overlay2 (default in Docker container), it takes 3,595 μs to update a 1-KB lower-layer file. The time increases to 11, 944 μs for a 4-MB file. Such update latency keeps growing as we further increase the size of the file (not shown in the figure). It is simply because, before applying a write, the overlay file system copies up the whole file from the lower, read-only layer to the upper, writable layer; as the file size increases, it takes increased time to complete the copy-up operation. We found that Overlay2 buries such a copy-up operation amidst file `open` – when an application opens a lower-layer file for writing (i.e., specified in the `flags` parameter), Overlay2 directly copies the file up, resulting in long blocking time during the file open – even for a file that is never modified but "mistakenly" opened with a "write" flag. In contrast, under the non-CoW case, regardless of file sizes, the completion time of an update is constant (around *22 μ*s) – once a file resides in the upper layer, the write operation of Overlay2 is fast and nearly the same as the native (i.e., without containers). Figure 2 shows the similar trend under AUFS – another overlay file-system implementation – but with shorter I/O latency under CoW and slightly higher I/O latency under non-CoW.

These results clearly demonstrate that, as a result of CoW, an update to a read-only, lower-layer file causes the overlay file system to take significant time for completion. In practice, a container instance is provisioned from a container image; its application suffers from the prolonged write latency for updating any data originating from the container image.

Startup Time: The write latency can be more significant as a container image contains larger files. For instance, to start a containerized MongoDB database (a No-SQL database) with a 40-GB data store committed in advance, it takes ∼28 seconds until it can start serving any requests (i.e., *startup time*). It is because, when the MongoDB database boots up, it opens its data store – consisting of several large files – with a *write-mode* flag. Indicated by such a flag, Overlay2 copies these large files from the lower to the upper layer, leading to long startup time. Indeed, the startup time of a containerized application can be prolonged by any files opened in the write mode. Figure 3 shows that, while starting, the web server (e.g., Nginx) and database servers (e.g., MongoDB and MySQL) open up a log file stored in the lower layer. It takes roughly 120 ms *extra* startup time with a 100-MB log file, compared with the non-CoW cases. The startup time becomes much longer with a larger log file – e.g., more than one second with a 1-GB log file. As reported in [41], slow container initialization can severely hurt common-case latency on emerging container-based cloud platforms. For example, in serverless computing [3, 37], a monolithic application is decomposed to multiple containerized serverless tasks, where startup cost is amplified. In addition, it is also *space-inefficient* to store the entire file in the upper layer, even only a portion of the file is updated.

These observations precisely reflect why the current "best practice" of containers suggests storing *read-write* data outside the container's file system (e.g., using container volumes or data containers) [16]. Unfortunately, it not only increases management/configuration complexity (e.g., one has to delicately distinguish read-write data from read-only data and create a volume for each unique directory), but also remains limiting as storing read-write data in a volume still needs to *pre-copy* the data from the container image to the volume directory. Further, though one is advised to use volumes, read-write data still commonly exists in container images. Over the analysis of 500,000 public container images [63], 44% are document files such as Microsoft office files, texts, source code, scripts, etc., many of which are supposed to be modified. It also reports a certain amount of database-related files in container images indicating that Docker developers run databases inside containers.

3 RELATED WORK

CoW-featured File Systems: The I/O overhead and space inefficiency, resulted from the CoW mechanism of overlay file systems, could be mitigated by *CoW-featured* file systems, like Btrfs [46] and ZFS [43]. They perform CoW at the *block* (rather than the file) granularity with CoW friendly data structures – e.g., a B+ tree in Btrfs and a Merkle tree in ZFS. Ideally, only the modified blocks (a portion of a file) need to be copied to new locations, leading to increased CoW performance and efficient storage usage. As illustrated in Figure 2, the CoW-featured file systems do perform better than overlay file systems in terms of CoW performance. For example, Btrfs takes 165 *μ*s to complete an update to a 4-MB file, while we observed that Overlay2 takes 11,944 *μ*s.

Though bringing performance benefit to CoW-related operations, the CoW-featured file-system approach, unfortunately, turns out to be cumbersome in practice. First, it trades performance of certain I/O operations for fast CoW – it would be very expensive for random updates, as an update affects many on-disk structures (e.g., in Btrfs, an entire path from root to the updated leaf node needs to be changed [46]). Second, it experiences high fragmentation due to fine-grained CoW, where defragmentation is required and costly [27]. Last, CoW-featured file systems are still not prevalent due to immaturity, instability, and others [45, 51], making it less accessible and adoptable especially for systems with other pre-installed file systems. As we will report in Section 6, Btrfs generally performs worse than non-CoW-featured file systems (e.g., Linux Ext4).

I/O Stack Optimization: The I/O latency brought by the virtualized I/O stacks has long been mitigated via both software techniques (e.g., para-virtualization [8, 33, 54, 62]) and hardware-assisted solutions (e.g., device passthrough [17, 39, 48]). Further, to avoid I/O delays caused by contention, many I/O scheduling [6, 21, 22, 26, 28, 29, 35, 56] and placement approaches [5, 12, 13, 40, 44, 47, 50, 60] were also proposed. For example, vTurbo [56] accelerated I/O processing by offloading I/O processing to a designated core, while Vanguard [47] eliminated I/O performance interference with dedicated I/O resources. These techniques mostly focus on I/O performance improvement for VMs. In contrast, BAOverlay mitigates the I/O performance overhead caused by an overlay file system for containers.

File Systems Optimization: Many specialized file systems were designed to improve I/O performance as well,

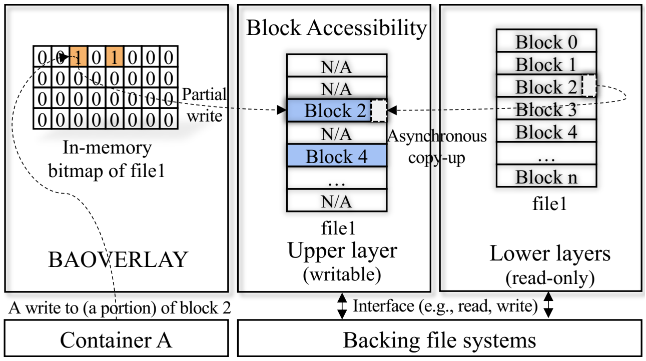


Figure 4: BAOverlay logically partitions overlay files into same-sized blocks to enable block-accessibility.

such as write-optimized file systems [2, 32, 34, 58], copy-on-write based file systems [7, 31, 43, 53], and others [38, 61]. Particularly, BetrFS[25] reduced write amplification and adopted buffer-level indexes for efficient look-up operations. PCOW [53] optimized CoW with a pipelined CoW to decrease performance penalty to first write. The composite-file file system [61] decoupled the one-to-one mapping between files and metadata for better performance. BAOverlay, as an overlay file system, builds upon existing backing file systems and can benefit from their improvements.

4 DESIGN OF BAOVERLAY

We design and develop BAOverlay, a lightweight, block-accessible overlay file system. First, BAOverlay introduces *fine-grained, block-level* accessibility to overlay files by logically partitioning an overlay file as a sequence of equally-sized blocks, thus being able to be accessed at the block level (Section 4.1). With the block accessibility, BAOverlay further enables a *non-blocking* CoW mechanism, wherein only updated blocks – instead of a full file – are asynchronously copied up from the lower layer to the upper layer (Section 4.2). Last, BAOverlay provides a new file format, B-CoW, for compactly storing overlay files to save storage space – allocation of storage space is delayed until blocks of the overlay files are actually being updated (Section 4.3).

4.1 Enabling Block Accessibility

The overlay approach builds upon the clear definition of the interface between the OS and file systems. In particular, in POSIX-compatible file systems, the representation of a file (or directory) is associated with an `inode` data structure. It stores the information (metadata) of a file's attributes (e.g., file ownership, access mode, and data location in disk) as well as an operations vector – specifically defining several operations (e.g., open, read, and write) that the OS can invoke when an application accesses the file. An overlay file system can interpose itself on a backing file system by creating an `overlay inode` for each overlay file (visible in the merged file-system view in Figure 1). The overlay inode contains overlay specific metadata (e.g., layers), the operation vector of the overlay file system, and its associated file on the backing file system – i.e., the inode of the file in the upper or the lower layer(s). Note that, unlike an inode from the backing file system, an overlay inode is a pure in-memory data structure to be transparent to the backing file system – no on-disk modifications are needed.

We use the open function in Overlay2 to demonstrate such an interposition mechanism. When a containerized application opens an overlay file, an open request is directed through the virtual file system (i.e., a uniform interface between the OS kernel and a concrete file system) to Overlay2 by invoking Overlay2's open function. Then, the open function of Overlay2 creates an overlay inode associated with the overlay file (if not created yet) and populates it with needed information. One main step is to figure out which layer the associated underlying file resides, fetch the actual file's inode data (via the backing file system's inode fetch operation), and put it into one field of the overlay inode. In the end, the Overlay2 open function invokes the backing file system's open function that takes the associated file's inode as an input to actually open the file. Similarly for `read` and `write` functions, Overlay2 injects its operations before invoking the backing file system's read/write functions.

As stated in Section 2.2, the costly CoW mechanism is injected in the Overlay2 open function: If a lower-layer file is opened with a write-mode flag, the whole file is copied from the lower layer to the upper layer, notwithstanding the fact that the file would not be updated at all. Intuitively, this (in)efficiency issue can be mitigated by moving the CoW operation to the overlay write operation – when the file is actually updated, a copy-up is applied. However, it will instead stall a write request – issued by the application – due to possibly long copy-up latency.

Instead, we observe that CoW is ideally not needed, as long as an overlay file system can keep track of the *file update information* – i.e., the offset and length of each write to overlay files. With such information, the overlay file system knows which portions of a file are updated and which portions are intact. Hence, the overlay file system can directly write the updated portion(s) (new data) to the upper layer without copying lower-layer data, though additional logic is needed for read/write operations to locate the correct portions. For instance, a read could involve reading certain portions of an overlay file from the upper layer and certain portions from the lower layer, by referring to the file update information. To achieve this, one simple way is to use a *dynamic* data structure (e.g., a linked list or treed structure) to track the file update information. However, as applications could issue massive, arbitrary-length writes, such a dynamic approach might be very costly in practice.

Block Accessibility: To this end, BAOverlay favors a *static* approach with a simple (and lightweight) data structure to track the locations of the updated portions in an overlay file. As depicted in Figure 4, BAOverlay logically partitions an overlay file (as well as its associated upper-layer file and/or lower-layer file) into a sequence of equally-sized blocks (e.g., 4 KB)[2] and tracks the locations of each block of the latest version using a bitmap (i.e., a new field added in the overlay inode) – e.g., bit `0` indicates a block remaining intact and in the lower layer, while bit `1` indicates a block updated and moved to the upper layer.

With this, *block-accessibility* is enabled by BAOverlay as follows (as illustrated in Figure 4): To write a lower-layer file for the first time – instead of conducting a full file copy-up – BAOverlay creates a *sketchy* file in the writable, upper layer with its size initially set to zero. This process is very fast, as it only involves metadata (i.e., the sketchy file's inode) to be asynchronously written to the upper layer's backing file system. Meanwhile, a bitmap associated with the file's overlay inode is allocated (and initialized to zeros) with each bit representing the location of one block. After this, the original write – to a continuous file range (i.e., represented by offset and length) – is converted to a sequence of block-based *small* writes directly mapped to the blocks within the file range being accessed.

There are two types of small writes: a *full-block* write (i.e., the length is aligned with the block size) and a *partial-block* write (i.e., not aligned). Note that, after conversion, only the first and the last small writes could be partial-block writes, while the ones in between are full-block writes. For a full-block access, the data is directly written to the sketchy file (i.e., filling up or overwriting the whole block) – no copy-up is needed. In contrast, for a partial-block write to an intact block (i.e., updated for the first time), in addition to writing the new data to the mapped portion of the block in the sketchy file, BAOverlay needs to copy the remaining portion(s) from the lower-layer file – a copy-up operation is needed. The reason that BAOverlay still needs to conduct CoW lies in that it uses a coarse-grained (and lightweight) approach to track file update information. To mitigate this overhead, BAOverlay provides an efficient asynchronous CoW approach (in Section 4.2).

As plotted in Figure 4, after a block of the sketchy file has been completely updated, its bit in the bitmap flips, indicating the block moved to the upper layer. As the overlay file is continually updated, its sketchy file is filled up with more blocks. Gradually, all blocks in the sketchy file are filled up, at which time the sketchy file becomes a complete upper-layer file. Before the sketchy file completes, for any read, BAOverlay coverts it to a sequence of block-based small reads (to mapped blocks). For each small read, BAOverlay looks up the bitmap to locate and read the latest version of the block – from the sketchy file or the lower-layer file.

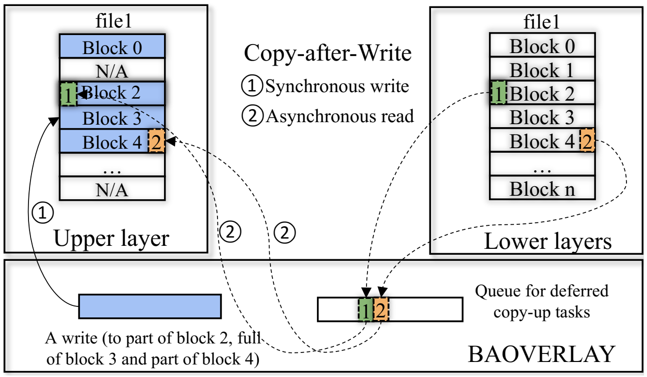


Figure 5: Copy-after-Write consists of two decoupled parts: (1) a synchronous and fast write and (2) a deferred and slow read.

To sum up, with block-accessibility, BAOverlay allows updating an overlay file at the *block* granularity. The block-granularity update prevents the overlay file system from copying the entire file at an update. This design provides several benefits: First, due to the small block size, a containerized application does not experience a long stall. In addition, the small block size allows useful blocks staying in memory (i.e, page cache) – benefiting subsequent reads to the same blocks. In contrast, a full file copy-up can easily fill the page cache and evict needed data blocks. Finally, the block-granularity update further allows the "copy" operation to be removed from the critical path of a write (as presented shortly in Section 4.2). As BAOverlay uses a bitmap to quickly and efficiently track the physical locations of blocks, it incurs little overhead to read/write operations (as shown in Section 6).

4.2 Taking "C" out of "CoW"

As we have seen, a copy-up operation is still needed in BAOverlay for a partial-block update. Recall that, the traditional CoW mechanism requires performing a blocking read (i.e., copy) before a write. Unlike the write that can leverage the page cache to buffer data and quickly return, the read usually triggers a slow path – to fetch data from the slow backing storage (e.g., HDD or SSD). Hence, a partial-block update may still suffer from long I/O latency due to the slow "copy" operation in the traditional CoW.

Copy-after-Write: To overcome this, BAOverlay introduces a *non-blocking* CoW mechanism, called copy-after-write (CaW). The key idea is that the "copy" operation can be taken out of a synchronous CoW operation and deferred

[2] A padding might be applied to the last block to be block aligned.

to a later appropriate time. As presented in Figure 5, CaW hides the slow read latency in a synchronous CoW operation by decomposing it into two portions: a *fast-and-synchronous* write – putting the updates (i.e., a portion of a block) into the upper sketchy file; and a *slow-and-asynchronous* read – fetching data from the backing file systems to the remaining portion(s) of the block in the sketchy file. This division allows BAOverlay to hide any latency of the read operation from applications, which only wait for the completion of the write (as such a write is usually to page cache, it is very fast).

The reasons that the two halves can be separated are: (1) The data to be copied are always stored in the read-only lower layer and do not need to be copied immediately; and (2) the data blocks (to be copied later) can be easily identified using the bitmap.

Handling Reads: Due to CaW, the content of an *in-transit* block – its deferred copy has not yet been completed – may spread in both the lower-layer file and the upper-layer sketchy file. This does not impact future writes, as writes always target the upper sketchy file. However, this may impact future reads: a read to an in-transit block needs to consult both the lower and the upper layers to compose a complete block. To capture this, a new state, *in-transit*, is added to the bitmap[3] For reads to these *in-transit* blocks, it involves two steps: (1) BAOverlay completes the deferred copy operation immediately – fetching data from the lower-layer file to fill up the remaining portion(s) of the block in the sketchy file. (2) Afterward, it reads the full block – which likely resides in page cache due to step (1) – and returns.

BAOverlay appears to transfer the blocking time from partial-block writes to future reads – the reads after writes might be blocked due to the existence of in-transit blocks. However, the amortized blocking time of BAOverlay is much shorter than the conventional CoW approach because: (1) As a read to a block for the first time needs to fetch data from the backing file system anyway, the additional overhead caused by BAOverlay is "filling up" (i.e., a write) the sketchy file (part of the above step 1). Again, this additional write can take advantage of page cache with little I/O latency; and (2) the deferred copy operation of an in-transit block might have sufficient time to complete before a future read arrives. In addition, a significant number of writes (e.g., 80% [36]) are not followed by any future reads to the same blocks at all.

4.3 Storing Overlay Files Efficiently

In BAOverlay, a sketchy file plays a crucial role in enabling block-accessibility and CaW. A sketchy file is analogous to a "jigsaw puzzle" – with writes to disparate portions of its associated lower-layer file, the sketchy file is being assembled and eventually becomes a complete upper-layer file. However, if the size of the associated lower-layer file is large, this assembling process may take a long time to complete, and the same large storage space is likely to be allocated *upfront* for the sketch file.

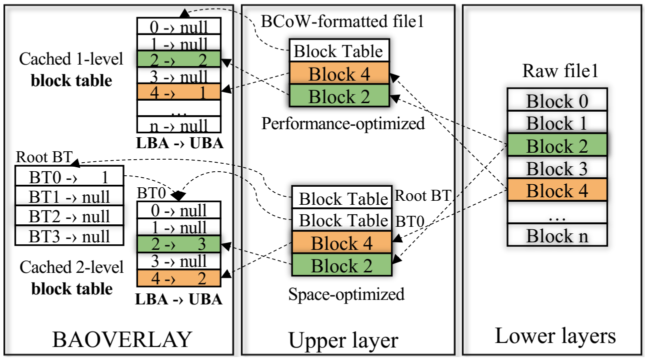


Figure 6: The BCoW file format applies a multi-level block table to store the mappings between the lower-layer addresses and the upper-layer addresses.

Simply using a bitmap (Section 4.1) is hard to achieve a space-efficiency goal, though a bitmap favors performance. It is because, a bitmap provides an inherent one-to-one mapping (of logical blocks) between the lower-layer file and its upper-layer sketchy file: A write access (for the first time) to the *n-th* block of the lower-layer file results in a fill-up of the *n-th* block of the sketchy file. Some file systems (e.g., Linux Ext3/4) support *sparseness* that efficiently use storage space by only updating the metadata of empty blocks (e.g., adjusting file sizes) instead of actually allocating the "empty" space. However, if the underlying file system (e.g., Windows FAT) does not support *sparseness*, the sketchy file's size is determined by the farthest block access (i.e., with the largest access offset). For example, if there is a write to the last block of the lower-layer file, the sketchy file's size will be the same as the lower-layer file, regardless of the actual number of blocks that are filled up – the sketchy file may contain a lot of "empty" holes. Even for the file systems supporting sparseness, filling up a sketchy file (with a lot of "holes") can have unexpected effects, such as disk-full or quota-exceeded errors, as the file size information in the metadata, indicates the total size including empty holes that do not take up any storage space yet.

BCoW Format: To this end, BAOverlay provides a new file format, BCoW, which stands for the BAOverlay Copy-on-Write format. With BCoW, BAOverlay enables sparse (sketchy) files to efficiently use storage space – transparently to the underlying file systems. Inspired by a canonical page-table based virtual memory system [20], BCoW uses a (multi-level) *block table* to maintain the mappings between the blocks in a lower-layer file and its counterparts in the

[3] BAOverlay needs at least two bits to represent the location and in-transit status of a block.

upper-layer, BCoW-formatted file. Simply speaking, it stores the mappings between the lower-layer block addresses (LBA) and the upper-layer block addresses (UBA), as illustrated in Figure 6. Note that, a mapping can be *null*, meaning that the lower-layer block has not been written or copied up to the upper layer. With BCoW, a write access (for the first time) to the *n-th* block of the lower-layer file results in an *append* operation, which simply writes the new block to the end of the upper-layer sketchy file – thus the sketchy file grows as distinct blocks are updated. The append operation in BAOverlay is also conducted in a CaW manner (Section 4.2). In the meantime, the corresponding LBA-to-UBA mapping in the block table is updated. The block table takes up a small portion of a BCoW file's space (e.g., stored in the first several blocks), and loaded to (and cached in) memory when the overlay file is being accessed.

Like a page-table approach, the design of the block table for BCoW needs to consider the space-time tradeoff: To be performance efficient, a single-level block table storing *all* LBA-to-UBA mappings is desired; for each read/write, one look-up is sufficient. On the other hand, to be space-efficient, a multi-level block table storing a portion of LBA-to-UBA mappings – only for the blocks being accessed – is desired; consequently, for each read/write, multiple look-ups are needed. To strike a reasonable balance between performance and storage space efficiency, BAOverlay explores two specific designs:

Performance-optimized Design: In this case, BAOverlay maintains a single-level block table storing all LBA-to-UBA mappings. However, if the block size is small, it might result in a large block table. For example, given an overlay file of 4 MB with the logical block size of 4 KB, it requires a block table with 1,000 entries, and thus 4 KB storage space (assume one entry takes up 32 bits). The block table size grows up to 4 GB for a 4-TB file.

Instead of using a fixed block size across all files, BAOverlay assigns varying block sizes for overlay files of different sizes. Specifically, BAOverlay keeps the single-level block table's size fixed (e.g., 4 KB with 1,000 entries), while adjusting the logical block size accordingly. For example, for a 4-MB overly file, the block size is set to 4 KB, while for a 4-GB overlay file, the block size is set to 4 MB. This design makes intuitive sense as for small overlay files, BAOverlay works at a fine-grained granularity (with small block size), while for large overlay files, BAOverlay works at a coarse-grained granularity (with large block size). More importantly, as the block table is small (e.g., 4 KB), BAOverlay is able to cache the full table in memory – ideally, the performance using a single-level block table is almost the same as that using an in-memory bitmap, except that the block-table approach involves a one-time disk access – fetching the block table (part of a BCoW file) to memory.

Space-optimized Design: The performance-optimized design may unnecessarily grow a BCoW file faster, as a tiny write (e.g., 1 byte) causes a whole block being copied up – the larger the block size, the faster the BCoW file grows. To further enable large files to be more space-efficient (i.e., using a small block size like 4 KB), BAOverlay adopts a multi-level block table. Similar to a multi-level page table, a multi-level block table allows *dynamically* allocating a set of sub-block tables with each covering a disparate *region* of a large file – one region consists of a sequence of adjacent blocks. For example, we can create a 1,000-entry, 4-KB sub-block table to cover the first 4-MB region of a large file, and another one to cover the last 4-MB region. These sub-block tables are only created when those regions are being updated. The locations of these sub-block tables are further tracked by another (root) block table, as depicted in Figure 6 (at the bottom).

Given a 2-level block table, to access a region (e.g., 4 MB) of a file (at most) two extra reads are involved: one fetches the root block table and another fetches a (second-level) sub-block table. Note that, the root block table is shared by all second-level sub-block tables and thus usually cached in the first place. Hence, the overhead to access one region only involves a one-time disk read (of the second-level sub-block table). To summarize, with the space-optimized design, a large overlay file can still save storage space with a small logical block size at the cost of maintaining a relatively complex multi-level block table. As a multi-level block table breaks one large block table into multiple sub-block tables, BAOverlay can efficiently cache them when their regions are being accessed, making the amortized cost of accesses low.

5 IMPLEMENTATION

We have implemented BAOverlay upon Linux's overlay file system (in kernel 5.3) with the focus on the implementation of the presented features: ~700 lines of code (LOC) for enabling block accessibility and CaW, and ~1,200 LOC for supporting BCoW.

BAOverlay Inode: As stated in Section 4.1, the key to interpose an overlay file system upon backing file systems is a clearly-defined interface – e.g., the `inode` in POSIX-compatible file systems. Following the same path, we developed a BAOverlay inode for interposition.

One main data structure that enables block accessibility in BAOverlay is a simple *bitmap*. BAOverlay uses 2 bits to represent the three states of one logical block in an overlay file: (i) in the lower-layer file, (ii) in the upper-layer file, and (iii) in the in-transit state. The bitmap is part of the BAOverlay's *in-memory* inode (i.e., a pointer array), and its memory space is allocated in a "region by region" manner (4 MB per region) for memory efficiency – only there is an access to a new region of the overlay file, that region's bitmap is allocated. This is especially useful for a large file, which needs a large memory space for the bitmap. Once all bits in a region's bitmap (i.e., a region) are flipped to 1, indicating all blocks moved to the upper layer, the region's bitmap can be freed. Further, to eliminate inconsistencies due to a sudden system crash, BAOverlay enables a "write-back" journaling mode for storing bitmap changes in a log file. More specifically, BAOverlay associates the bitmap to a log file – every time there is an update to the bitmap, it will be written to the log file. The log file is protected by the backing file system's journaling mechanism – e.g., writeback, ordered, and data in Linux Ext4, and BAOverlay chose the default "writeback" journaling mode.

When it comes to a BCoW file, an in-memory *block table* is associated with the BAOverlay inode. The in-memory block table is populated from the on-disk block table – part of the BCoW file. BAOverlay stores the single-level block table (or the root block table) in the first 4 KB of the BCoW file. The single-level (or root) block table is loaded/cached to memory when the BCoW file is open for the first time. For a multi-level block table, the sub-block tables' locations are indicated by the root block table, and one sub-block table is created/loaded only when its mapped region is being accessed for storage efficiency. BAOverlay currently supports at most 2-level block table in favor of fast I/O accesses.

Copy-after-Write: To enable CaW, BAOverlay needs to add logic in overlay open, read, and write functions.

In the *open* function, if the "write-mode" flag is specified and the target file resides in the lower layer, BAOverlay creates an "empty" sketchy file (and all the directories along its path if they do not exist) in the upper layer. Yet, the copy-up is not conducted. Meanwhile, the BAOverlay inode is created with the in-memory bitmap (or block table if BCoW is supported) initialized. The inode is cached in the memory (e.g., dCache in Linux) to accelerate future file accesses.

In the *write* function, BAOverlay converts an original write request to a sequence of block-based small writes. For each partial-block write to a block for the first time, BAOverlay generates a copy-up task (i.e., by marking unmodified portions that need to be copied) and places it in a copy-up work queue (implemented using a hash table). Right after this, BAOverlay invokes the underlying file system's write function to perform the original write and returns. The copy-up work queue is processed, asynchronously, by a designated kernel thread (one kernel thread for each mount point). BAOverlay simply schedules the kernel thread when a CPU is available, though we note other strategies, such as invoking the kernel thread when the disk is not busy.

Similarly, in the *read* function, BAOverlay converts each read request to a sequence of block-based small reads. Then, for each read, it locates the actual data block by looking up the bitmap (or block table) and invokes the underlying file system's read function to read the data. If a block's state is *in-transit*, BAOverlay accelerates its copy-up task(s) by directly fetching and completing such tasks from the copy-up work queue. Note that, if the bitmap has not been allocated (i.e., NULL), all data stay in the lower-layer file; if the bitmap has been freed (specified by a magic number), all data reside in the upper-layer file. It means that, when a file is not under CaW, the read performance is the same as the native.

File System Integration: As an overlay file-system approach, BAOverlay does not alter any semantic level activities of underlying file systems and is readily to run upon a variety of existing backing file systems. We have integrated and thoroughly tested BAOverlay with Linux Ext4.

6 EVALUATION

We present our comprehensive evaluation results to demonstrate the benefits and costs of BAOverlay with micro-benchmarks, stress testing, and real-world applications.

Evaluation Setup: Same as Section 2.2, our testbed consisted of a physical machine equipped with a 10-core CPU (E5-2630 v4 @ 2.20 GHz), 64 GB of RAM, and a 500-GB NVMe SSD (Samsung 970 EVO) as the backing store. We used Docker container [24] of version 19.03 on Linux kernel 5.3 and Linux Ext4 as the underlying backing file system. We evaluated BAOverlay mainly with two container storage techniques: Docker container's default overlay file-system solution, Overlay2 [15] and a CoW-featured file system, Btrfs [46]. We also included AUFS [19] (the previous default overlay solution in Docker) for comparison in micro-benchmarks cases. In addition, we compared BAOverlay with the native case (i.e., running applications directly on Linux Ext4 without containers) to demonstrate possible overhead caused by the overlay approach.

6.1 Micro-benchmark

To evaluate the effectiveness of BAOverlay from different angles, we first use our own micro-benchmark tools. In particular, we tested BAOverlay in four separate cases, including write-once, write-many, impact-on-reads, and overhead of BCoW. For the first three cases, we focused on the performance benefit and overhead of BAOverlay and configured BAOverlay to use "bitmap" to track file update information. For the fourth case, we aimed to understand the possible overhead of the BCoW files. Hence, we configured BAOverlay to be space-optimized (with a 2-level block table).

Write-once Case: Let us re-visit the motivational *write-once* case in Section 2.2 – a containerized application accesses a container file of different sizes via three consecutive operations: (1) open the file, (2) write a byte, and (3) close the file. We have added the performance data (i.e., the completion

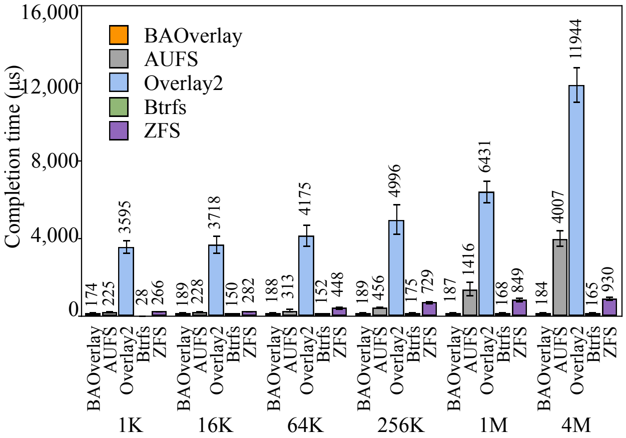


Figure 7: Performance comparisons in the write-once case: Irrespective of the size of the files, the completion time under BAOverlay remains constant and much smaller than Overlay2 and AUFS.

time taken by the three operations in total) under BAOverlay in Figure 7. We observe that, irrespective of the size of the files, the completion time under BAOverlay remains constant (and small) – around 170 ∼ 180 μs. In contrast, under Overlay2, it takes 3,595 μs for a 1-KB file (20x slower), while 11,944 μs for a 4-MB file (70x slower). Though AUFS seems lightweight in comparison with Overlay2 in the write-once case, its performance is worse than BAOverlay– it takes 225 μs for a 1-KB file (2x slower), while 4,007 μs for a 4-MB file (20x slower). The performance benefit is brought by BAOverlay's block-accessibility attribute and CaW mechanism – (1) due to block-accessibility, BAOverlay only copies the blocks being updated (i.e., the first 4-KB block) instead of the whole file; (2) due to CaW, the copy operation is further deferred to an asynchronous background process. In addition, we observe that, BAOverlay achieves very close or even better performance – in terms of low write latency – compared with CoW-featured file systems, like ZFS and Btrfs (e.g., under 4 MB). Note that the advantage of BAOverlay is that it is compatible with existing backing file systems, while the CoW-featured file systems are not.

While the write latency under BAOverlay is significantly reduced compared to Overlay2 and AUFS, it remains longer than the native case (180 μs vs. 27 μs). It is because BAOverlay interposes its overlay inode upon backing file systems' inodes and spends more time populating the overlay inode. As we will see shortly in the write-many test case, this is just a one-time overhead.

Write-many Case: In this case, a containerized application sequentially writes a 2-GB container file (initially in the lower layer) block by block with the 4-KB block size. For each write, it opens the file, moves the file pointer to the current block's position, writes 410 bytes (~10% of 4 KB) to the block, and closes the file.

Test cases	Total time (s)	Ave. write lat. (μs)	First write lat. (μs)
Native (w/o cache)	232.20	442.89	777
Native (w/ cache)	1.39	2.65	27.01
AUFS	9.65	17.50	1, 785, 295
Overlay2	26.20	49.98	24, 096, 098
BAOverlay	**3.39**	6.47	189.15

Table 1: Performance comparisons in the write-many case: BAOverlay completes writing the 2-GB file faster than Overlay2 and AUFS because of the copy-after-write mechanism.

Table 1 summarizes that, under Overlay2 (or AUFS), it takes 26 seconds (or 9.65 seconds) to complete writing the whole file, while under BAOverlay it takes only *3.39* seconds (7.7x or 2.84x faster). This is again due to CaW – BAOverlay takes the copy operation out of CoW, and each write simply puts data in memory (i.e., page cache) and returns. In contrast, Overlay2 needs to copy the full 2-GB file from the lower to the upper layer when it opens the file – that is why the first write latency of Overlay2 is 24 seconds (Table 1 Row 5). Again, under BAOverlay, we observe the similar first write latency (∼ 189 μs) as that in the write-once case. Yet, the subsequent write latency decreases dramatically – the average write latency under BAOverlay is only 6.47 μs (Table 1 Row 6). The reason is that, once the overlay inode is populated (as in the first write), it will be cached in memory and exploited by future file operations.

Surprisingly, we observe that the native case takes a much longer time – 232 seconds (Table 1 Row 2). Upon deeper investigation, we found that this is caused by the read-modify-write (RMW) restriction [18] – partial writes (i.e., unaligned with the cache page size like 4 KB) to non-cached data cause I/O applications to suffer from long I/O blocking time, as a slow page read before a partial write is needed. In the write-many case, each 410-byte write is a partial write and the file is not cached in memory. Thus, for such a partial write, the OS fetches the corresponding page-aligned block from the disk to page cache (which is slow) and then applies the partial update to the cached block. To confirm this, we cached the full file in memory in advance (by reading the full file before the test), and conducted the same experiment. This time, it takes only 1.39 seconds under the native (Table 1 Row 3). In contrast, under BAOverlay, the RMW restriction does not apply, as the partial writes under BAOverlay target *empty* blocks in a sketchy file – instead of fetching on-disk blocks, the OS simply fills "null" in page cache.

These results clearly demonstrate that, compared with Overlay2 (or AUFS), BAOverlay significantly reduces write

Test cases	Total time (s)	Ave. write lat. (*μ*s)	Ave. read lat. (*μ*s)
Native (unlimited cache)	234.81	447.18	3.4
Native (limited cache)	259.27	446.76	238.76
AUFS (unlimited cache)	10.46	18.25	5.4
AUFS (limited cache)	17.71	17.50	77.25
Overlay2 (unlimited cache)	25.93	48.77	3.4
Overlay2 (limited cache)	51.30	48.75	245.53
BAOverlay (a worst case)	50.26	6.53	446.62
BAOverlay (a practical case)	3.76	6.49	3.4

Table 2: Performance impact on reads: In a worst case scenario, BAOverlay does incur longer read latency, which can be mitigated when the CaW completes in a timely manner.

latency. BAOverlay may even outperform the native case under some circumstances (e.g., partial writes).

Impact on Reads: As stated in Section 4.2, BAOverlay may change the behavior of the reads that access *in-transit* blocks, due to CaW. To measure the impact of BAOverlay on such reads, we performed a *worst-case* scenario evaluation: We used the write-many case setup to first sequentially write a 2-GB lower-layer file. Afterward, 20% of these blocks were read (randomly picked) – to simulate a real read-after-write scenario [36]. In the worst-case scenario, we configured BAOverlay to defer CaW infinitely, which means, for each read, BAOverlay triggered a long I/O path: (1) completed the CaW; (2) read the full block; and (3) returned.

Table 2 shows that, as expected, for such reads accessing *in-transit* blocks, BAOverlay spends a longer time (~ 446.62 *μ*s in Row 8) due to the longer I/O path. In contrast, the read latency under Overlay2, AUFS, and the native is very low (3.4 *μ*s for Overlay2 and the native in Row 6 & 2, and 5.4 *μ*s for AUFS in Row 4). It is due to the fact that, in all these cases, after finishing writing the whole file, the file is cached in memory – the following read operations fetch data directly from memory instead of slow disk. However, the cost to enjoy such fast reads is that Overlay2 blocks a containerized application for 24 seconds as a result of CoW. In addition, if the container has limited memory (less than 2-GB) or the file size is too big to fit in the memory cache, the read latency under both native and Overlay2/AUFS increases to more than 77 *μ*s (Row 5). Even with high read latency, BAOverlay achieves the same total completion time as Overlay2 (~ 50 seconds) and continuously outperforms the native case.

Test cases	Ave. write lat. (*μ*s)	Total time (s)
Native (with cache)	2.66	1.39
AUFS	17.50	9.65
Overlay2	49.98	26.20
BAOverlay (Case 1)	5.75	3.01
BAOverlay (Case 2)	3.74	1.97

Table 3: Overhead analysis of BCoW: BAOverlay incurs slight overhead in populating block tables (under Case 1), while little (or no) overhead when the blocks tables are established (under Case 2).

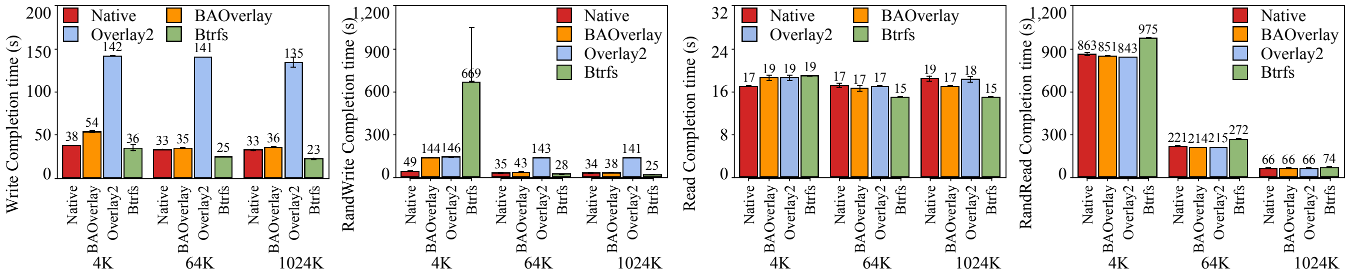
This demonstrates a worst case scenario. In a more practical setting – BAOverlay has sufficient time to complete CaW (i.e., a write is not immediately followed by a read to the same block) – BAOverlay can also return reads quickly (3.4 *μ*s in Table 2, Row 9).

Overhead of BCoW: Section 4.3 presents that to support BCoW, a *block table* is involved. As BAOverlay's implementation supports at most a 2-level block table for fast I/O accesses, we measured the overhead of BCoW by accessing a 2-GB BCoW-formatted file with a configuration of a 2-level block table and 4-KB block size. We, particularly, focused on two representative cases: (1) to create a BCoW-formatted (sketchy) file during CaW; and (2) to access a BCoW-formatted file that has been in the upper layer.

Table 3 shows that, under both cases, BAOverlay leads to low write latency (5.75 *μ*s and 3.74 *μ*s respectively in Row 5 & 6) – it is close to the native case (2.66 *μ* in Row 2) and much better than Overlay2 and AUFS (49.98 *μ* and 17.5 *μ* in Row 4 & 3). Noticeably, BAOverlay achieves lower write latency in Case 2 in contrast to Case 1, because under Case 2, the multi-level block tables are established and stored in the BCoW file and BAOverlay only needs to read it (rather than populate it from scratch as in Case 1) – the cost of a sub-block table read is further amortized over multiple block writes (i.e., 1000). We further observed that the size of the BCoW-formatted file under Case 1, grew as the experiment advanced (i.e., efficient use of storage space due to BCoW).

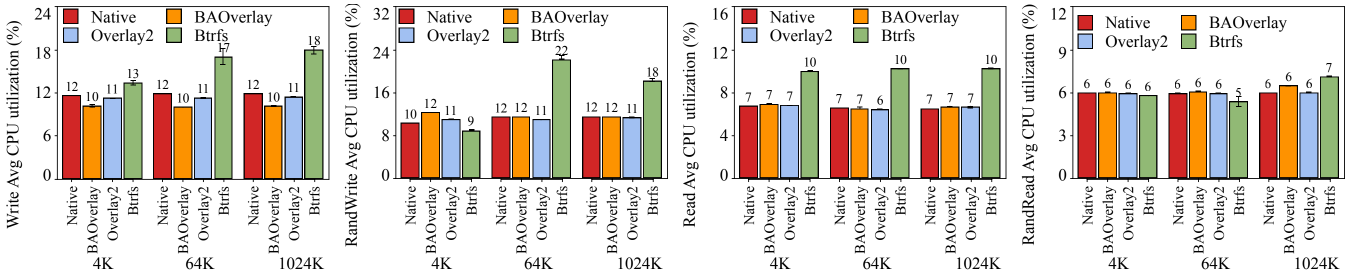
6.2 Stress-testing

To further stress test BAOverlay, we used a public microbenchmark tool, Fio [4]. Specifically, a 40-GB file was created and stored in the Fio's container image. We focused on four typical I/Os: sequential/random writes and sequential/random reads. One thread of the containerized Fio generated I/O requests of the four types to the lower-layer 40-GB file, with the I/O depth being one and the I/O sizes ranging from 4 KB to 1024 KB. We configured BAOverlay to use



(a) Sequential write. **(b) Random write.** **(c) Sequential read.** **(d) Rand read.**

Figure 8: Performance comparisons under four typical I/O types among the native, BAOverlay, Overlay2 and Btrfs: BAOverlay generally outperforms Overlay2 and is very close to the native in most cases.



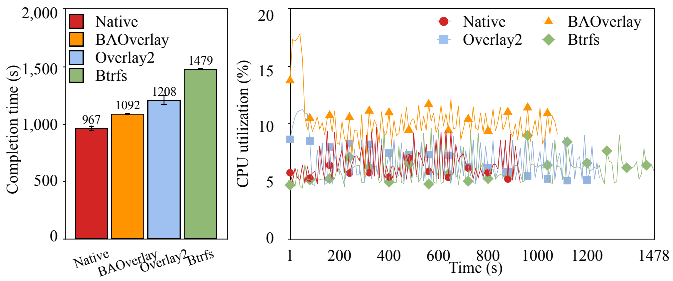
(a) Sequential write. **(b) Random write.** **(c) Sequential read.** **(d) Rand read.**

Figure 9: Comparisons of average CPU utilization among the native, BAOverlay, Overlay2 and Btrfs: the CPU utilization under BAOverlay is almost the same as those under the native and Overlay2.

"bitmap" to track file update information and compared it with Overlay2, Btrfs, and the native.

Figure 8 presents the completion time under each case. Under the sequential and random write cases, BAOverlay outperforms Overlay2 significantly – BAOverlay takes 54 seconds to complete the sequential write under the 4-KB I/O size, while Overlay2 takes 142 seconds (2.62x slower). The performance gap widens with larger I/O size – under the same sequential write with the 1024-KB I/O size, BAOverlay takes 36 seconds to complete, while Overlay2 takes 135 seconds (3.75x slower). It is because, with the I/O sizes being aligned with the block size (i.e., 4 KB), BAOverlay does not involve any copy-up; instead, it simply writes data to the upper sketchy file. This also explains the close performance between BAOverlay and the native under all cases – including the read cases, indicating that BAOverlay incurs *little* overhead to non-CaW reads.

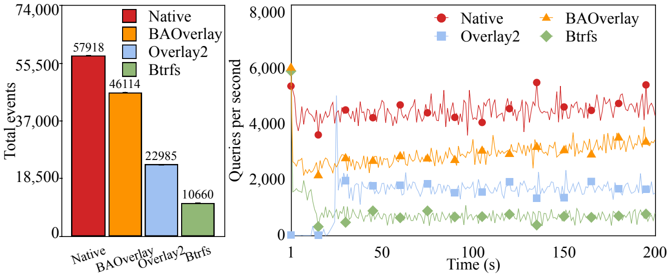
Figure 9 presents the average CPU utilization over the full run under each test case. We find that the CPU utilization under BAOverlay is almost the same as those under the native and Overlay2. Btrfs consumes much more CPU than others while achieving similar or even worse I/O performance (for random write in Figure 9b). It is because, in Btrfs, a single update alters many on-disk structures causing significant overhead. In addition, the performance of random writes under Btrfs varies a lot, as plotted in Figure 9b. These test cases do not involve CaW, as all I/O sizes are aligned with the block size (i.e., 4 KB). It shows that equipped with block accessibility, BAOverlay avoids unnecessary data copy-ups.



(a) Performance. **(b) CPU% over time.**

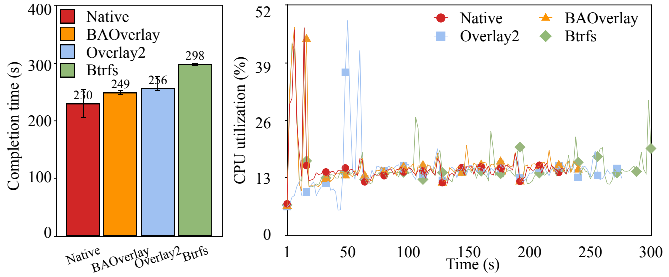
Figure 10: Performance and CPU utilization under partial-block writes: BAOverlay results in less completion time than Overlay2 while slight longer time than the native due to CaW.

To evaluate CaW under Fio, we used Fio to write the same 40-GB file with I/O requests accessing 4-KB blocks sequentially, but leaving a 3-KB hole for each access (i.e., partial-block writes). Thus, BAOverlay needs to create a copy-up task for each write. Under this condition, compared with Overlay2, BAOverlay leads to less completion time (1,092



(a) Total events. **(b) Number of served queries over time.**

Figure 11: Performance comparisons of Sysbench on MySQL: BAOverlay achieves higher throughput in terms of total events compared to Overlay2.



(a) Completion time. **(b) CPU% over time.**

Figure 12: Performance comparisons of YCSB on MongoDB: BAOverlay completes 1-million updates faster than Overlay2 with stable CPU consumption.

seconds vs. 1,208 seconds in Figure 10a), with slightly higher CPU usage (e.g., 2%∼5% more CPU usage in Figure 10b). Again, Btrfs performs poorly among all.

6.3 Application Workloads

We used realistic workloads to test BAOverlay, including Sysbench [30] and YCSB [11]. We configured BAOverlay to use "bitmap" to track file update information and compared it with Overlay2, Btrfs, and the native.

Sysbench (MySQL): Sysbench is an OLTP application benchmark running upon a transactional database. We chose MySQL (version 5.5) with its default storage engine, INNODB, and installed it – consisting of 24 tables, each with 4 million items – in a container image. We ran 16 Sysbench threads remotely in a client for 300 seconds in each run, which generated a mix of queries (e.g, select, insert, and update) to MySQL.

Figure 11a shows that the throughput of MySQL, in terms of total events over the 300-second testing period, increases by 100% under BAOverlay, compared to Overlay2. The reason is that, with CaW, BAOverlay avoids long blocking time caused by CoW and can keep servicing clients (Sysbench) through the whole run, as illustrated in Figure 11b. In contrast, Overlay2 frequently copies up a full MySQL data store file, which blocks a specific Sysbench thread for a long time, resulting in lower overall performance. In addition, Btrfs continues to lead to worst performance.

YCSB (MongoDB): Yahoo Cloud Serving Benchmark (YCSB) is an industry-standard performance benchmark for NoSQL databases. We ran YCSB against a popular document-oriented NoSQL – MongoDB (version 3.0.10). We simply deployed the MongoDB in a container image with a 20-GB data store. We selected "UPDATE" of YCSB as the core workload. For each run, the workload inserted 1 million 1-KB records.

Figure 12a shows that BAOverlay improves the "INSERT" throughput of MongoDB slightly by 3%, compared to Overlay2. Under the YCSB case, we observe that Overlay2 stalls the client (YCSB) in the beginning of each run: When the MongoDB database starts, it opens its data store with a write-mode flag, wherein Overlay2 copies up the whole 20-GB data store, which takes around 14 seconds. This is further confirmed in Figure 12b, which displays the CPU% over time. BAOverlay consumes more CPU resources during MongoDB's startup, due to starting serving requests. In contrast, the CPU utilization under Overlay2 keeps stable in the beginning, partly attributed to the copy-up operation.

Discussions: Compared to other overlay file systems (e.g., Overlay2 and AUFS), BAOverlay significantly reduces the first write latency with both block-accessibility and the CaW mechanism. Yet, it still falls behind the native case (170 μs vs. 22 μs) as BAOverlay takes non-negligible time to populate the overlay inode leading to increased latency. New mechanisms are further needed to either mitigate or hide such high first write latency, especially in a serverless computing scenario where fast container startup time is much necessary [1, 23, 49]. Besides, BAOverlay's BCoW works at a per-instance granularity. To make container storage space more efficient, a BCoW file might need to be further shared by multiple containers, given the fact that a physical host can support hundreds/thousands of containers.

7 CONCLUSIONS

We have presented BAOverlay, a lightweight, block-accessible container overlay file system. Equipped with block accessibility, BAOverlay enables a non-blocking copy-after-write mechanism for accelerated file updates and a new file format for the efficient use of container storage space. Our extensive evaluation with both micro-benchmarks and real-world applications demonstrates the effectiveness and general applicability of BAOverlay.

8 ACKNOWLEDGMENTS

We thank our shepherd, Mahesh Balakrishnan, and the anonymous reviewers for their insightful comments. This work was supported in part by NSF under Award 1909877.

REFERENCES

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 419–434.

[2] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2009. FAWN: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 1–14.

[3] AWS. 2020. AWS Lambda: Run code without thinking about servers. Pay only for the compute time you consume. https://aws.amazon.com/lambda/.

[4] Jens Axboe. 2020. fio(1) - Linux man page. https://hub.docker.com/.

[5] Jon CR Bennett and Hui Zhang. 1996. WF/sup 2/Q: worst-case fair weighted fair queueing. In *INFOCOM'96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE*.

[6] Jon CR Bennett and Hui Zhang. 1997. Hierarchical packet fair queueing algorithms. *IEEE/ACM Transactions on Networking (TON)* 5, 5 (1997), 675–689.

[7] David Bernstein. 2014. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing* 1, 3 (2014), 81–84.

[8] Anton Burtsev, Kiran Srinivasan, Prashanth Radhakrishnan, Kaladhar Voruganti, and Garth R Goodson. 2009. Fido: Fast Inter-Virtual-Machine Communication for Enterprise Appliances.. In *USENIX Annual technical conference*.

[9] Google Cloud. 2020. Containers at Google. https://cloud.google.com/containers/.

[10] Kata Containers community. 2020. The speed of containers, the security of VMs. https://katacontainers.io/.

[11] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[12] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *ACM SIGPLAN Notices*, Vol. 48. 77–88.

[13] Alan Demers, Srinivasan Keshav, and Scott Shenker. 1989. Analysis and simulation of a fair queueing algorithm. In *ACM SIGCOMM Computer Communication Review*, Vol. 19. 1–12.

[14] Docker Docs. 2020. *About storage drivers*. https://docs.docker.com/storage/storagedriver/.

[15] Docker Docs. 2020. Use the OverlayFS storage driver. https://docs.docker.com/storage/storagedriver/overlayfs-driver/.

[16] Docker Docs. 2020. Use volumes. https://docs.docker.com/storage/volumes/.

[17] VMWare Docs. 2020. Single Root I/O Virtualization (SR-IOV). https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.networking.doc/GUID-CC021803-30EA-444D-BCBE-618E0D836B9F.html.

[18] Garth Gibson and Greg Ganger. 2011. Principles of operation for shingled disk devices. (2011).

[19] GitHub. 2020. Aufs5. https://github.com/sfjro/aufs5-linux.

[20] Mel Gorman. 2004. *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River.

[21] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. 2009. PARDA: Proportional Allocation of Resources for Distributed Storage Access.. In *Proceedings of the USENIX Conference on File and Storage Technologies*.

[22] Ajay Gulati, Arif Merchant, and Peter J. Varman. 2007. pClock: an arrival curve based approach for QoS guarantees in shared storage systems. In *SIGMETRICS*. ACM.

[23] 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)*.

[24] Docker Inc. 2020. Get Started with Docker. https://www.docker.com.

[25] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, et al. 2015. BetrFS: A right-optimized write-optimized file system. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. 301–315.

[26] Wei Jin, Jeffrey S Chase, and Jasleen Kaur. 2004. Interposed proportional sharing for a storage service utility. *ACM SIGMETRICS Performance Evaluation Review* 32, 1 (2004), 37–48.

[27] Meaza Taye Kebede. 2012. *Performance comparison of btrfs and ext4 filesystems*. Master's thesis.

[28] Terence Kelly, Ira Cohen, Moises Goldszmidt, and Kimberly Keeton. 2004. Inducing models of black-box storage arrays. *HP Laboratories, Palo Alto, CA, Technical Report HPL-2004-108* (2004).

[29] Linux Kernel. 2020. *CFQ*. https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt.

[30] Alexey Kopytov. 2004. SysBench manual. (2004).

[31] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. 2017. High Performance Metadata Integrity Protection in the {WAFL} Copy-on-Write File System. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. 197–212.

[32] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. 273–286.

[33] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleswar K. Panda. 2006. High Performance VMM-bypass I/O in Virtual Machines. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*.

[34] David Lomet and Mark Tuttle. 1999. Logical logging to extend recovery to new domains. *ACM SIGMOD Record* 28, 2 (1999), 73–84.

[35] Hui Lu, Brendan Saltaformaggio, Ramana Kompella, and Dongyan Xu. 2015. vFair: Latency-aware Fair Storage Scheduling via per-IO Cost-based Differentiation. In *Proceedings of the 6th ACM Symposium on Cloud Computing*.

[36] Hui Lu, Brendan Saltaformaggio, Cong Xu, Umesh Bellur, and Dongyan Xu. 2016. Bass: Improving i/o performance for cloud block storage via byte-addressable storage stack. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*.

[37] Garrett McGrath and Paul R Brenner. 2017. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 405–410.

[38] Athicha Muthitacharoen, Robert Morris, Thomer M Gil, and Benjie Chen. 2002. Ivy: A read/write peer-to-peer file system. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 31–44.

[39] Jun Nakajima. 2007. Intel virtualization technology roadmap and VT-d support in Xen. http://www-archive.xenproject.org/files/xensummit_4/VT_roadmap_d_Nakajima.pdf. (2007).

[40] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. 2010. Q-clouds: Managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European Conference on Computer Systems*.

[41] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 57–70.

[42] Junjiro R. Okajima. 2020. Linux AuFS Examples: Another Union File System Tutorial. http://aufs.sourceforge.net/aufs2/.

[43] Openzfs. 2020. zfs-0.8.3. https://github.com/openzfs/zfs/releases.

[44] Abhay K Parekh and Robert G Gallager. 1993. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM transactions on networking* 1, 3 (1993), 344–357.

[45] Abhishek Prakash. 2020. Don't Use ZFS on Linux. https://itsfoss.com/linus-torvalds-zfs/.

[46] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)* 9, 3 (2013), 1–32.

[47] Yannis Sfakianakis, Stelios Mavridis, Anastasios Papagiannis, Spyridon Papageorgiou, Markos Fountoulakis, Manolis Marazakis, and Angelos Bilas. 2014. Vanguard: Increasing Server Efficiency via Workload Isolation in the Storage I/O Path. In *Proceedings of the ACM Symposium on Cloud Computing*.

[48] Amit Shah, Allen M. Kay, Muli Ben-Yehuda, and Ben-Ami Yassour. 2008. PCI Device Passthrough for KVM. https://www.linux-kvm.org/images/d/d0/KvmForum2008%24kdf2008_14.pdf. (2008).

[49] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 419–433. https://www.usenix.org/conference/atc20/presentation/shillaker

[50] David Shue, Michael J. Freedman, and Anees Shaikh. 2012. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*.

[51] Vasily Tarasov, Lukas Rupprecht, Dimitris Skourtis, Wenji Li, Raju Rangaswami, and Ming Zhao. 2019. Evaluating Docker storage performance: from workloads to graph drivers. *Cluster Computing* 22, 4 (2019), 1159–1172.

[52] Steven J. Vaughan-Nichols. 2020. Docker makes ready-to-run container apps available. https://www.zdnet.com/article/docker-makes-ready-to-run-container-apps-available/.

[53] Zhikun Wang, Dan Feng, Ke Zhou, and Fang Wang. 2008. PCOW: Pipelining-based COW snapshot method to decrease first write penalty. In *International Conference on Grid and Pervasive Computing*. Springer, 266–274.

[54] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. 2008. Scalable Performance of the Panasas Parallel File System.. In *Proceedings of the USENIX Conference on File and Storage Technologies*.

[55] Charles P. Wright and Erez Zadok. 2004. Kernel Korner: Unionfs: Bringing Filesystems Together. *Linux J.* 2004, 128 (Dec. 2004), 8.

[56] Cong Xu, Sahan Gamage, Hui Lu, Ramana Rao Kompella, and Dongyan Xu. 2013. vTurbo: Accelerating Virtual Machine I/O Processing Using Designated Turbo-Sliced Core.. In *USENIX Annual Technical Conference*.

[57] Ethan G Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2019. The true cost of containing: A gVisor case study. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*.

[58] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael Bender, et al. 2016. Optimizing every operation in a write-optimized file system. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. 1–14.

[59] Erez Zadok, Rakesh Iyer, Nikolai Joukov, Gopalan Sivathanu, and Charles P Wright. 2006. On incremental file system development. *ACM Transactions on Storage (TOS)* 2, 2 (2006), 161–196.

[60] Lixia Zhang. 1991. VirtualClock: a new traffic control algorithm for packet-switched networks. *ACM Transactions on Computer Systems (TOCS)* 9, 2 (1991), 101–124.

[61] Shuanglong Zhang, Helen Catanese, and Andy An-I Wang. 2016. The composite-file file system: Decoupling the one-to-one mapping of files and metadata for better performance. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. 15–22.

[62] Xiaolan Zhang, Suzanne McIntosh, Pankaj Rohatgi, and John Linwood Griffin. 2007. XenSocket: A High-throughput Interdomain Transport for Virtual Machines. In *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*.

[63] Nannan Zhao, Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Amit S Warke, Mohamed Mohamed, and Ali R Butt. 2019. Large-Scale Analysis of the Docker Hub Dataset. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 1–10.