# Flexible, Modular File Volume Virtualization in Loris

Raja Appuswamy
Computer science Dept.
Vrije Universiteit
Amsterdam, Netherlands

David C. van Moolenbroek
Computer science Dept.
Vrije Universiteit
Amsterdam, Netherlands

Andrew S. Tanenbaum
Computer science Dept.
Vrije Universiteit
Amsterdam, Netherlands

***Abstract*—Traditional file systems made it possible for administrators to create file volumes, on a one-file-volume-per-disk basis. With the advent of RAID algorithms and their integration at the block level, this "one file volume per disk" bond forced administrators to create a single, shared file volume across all users to maximize storage efficiency, thereby complicating administration. To simplify administration, and to introduce new functionalities, file volume virtualization support was added at the block level. This new virtualization engine is commonly referred to as the *volume manager*, and the resulting arrangement, with volume managers operating below file systems, has been referred to as the *traditional storage stack*.**

**In this paper, we present several problems associated with the compatibility-driven integration of file volume virtualization at the block level. In earlier work, we presented Loris, a reliable, modular storage stack, that solved several problems with the traditional storage stack by design. In this paper, we extend Loris to support file volume virtualization. In doing so, we first present "File pools", our novel storage model to simplify storage administration, and support efficient file volume virtualization. Following this, we will describe how our single unified virtualization infrastructure, with a modular division of labor, is used to support several new functionalities like 1) instantaneous snapshotting of both files and file volumes, 2) efficient snapshot deletion through information sharing, and 3) open-close versioning of files. We then present "Version directories," our unified interface for browsing file history information. Finally, we will evaluate the infrastructure, and provide an in-depth comparison of our approach with other competing approaches.**

## I. Introduction

Over the past few decades in the evolution of file and storage systems, storage virtualization techniques have played a crucial role in improving efficiency and manageability. Traditional file systems provided the first layer of virtualization. File systems were used to create hierarchies of files and directories, also referred to as file volumes[1], on dedicated disk drives. Disks were small enough that administrators could create one file volume per logical unit (per user or per project for instance), and apply administrative policies on these file volumes. As disks grew larger, administrators were forced to use a single file volume across all users to improve storage efficiency. Thus, this "one file volume per device" bond complicated administration, as administrators could no longer use file volumes as the unit of administration.

[1]Throughout this paper, we will use the term file system to refer to the operating system code that implements a persistent name space of files and directories, and *file volumes* to refer to an instantiation of a file system

978-1-4577-0428-4/11/$26.00 © 2011 IEEE

Volume managers [22] solved this problem by virtualizing file volumes. Similar to RAID algorithms, volume managers were integrated at the block level to retain compatibility with existing installations. The resulting arrangement, with volume managers operating below file systems, has been referred to as the *traditional storage stack*. In this stack, file systems are used to create file volumes on logical disks exposed by the volume manager. Thus, file systems translate file requests to logical block requests. The volume manager transparently maps these logical blocks to blocks on physical devices it manages. As a result, multiple logical disks, and hence multiple file volumes, could now share the same set of physical disk drives, thus improving storage efficiency. Volume managers also simplified administration, as administrators could now create and manage file volumes in logical units.

In this paper, we examine the block-level integration of file volume virtualization, and we highlight several problems along two dimensions: flexibility and heterogeneity. In prior work, we outlined several fatal flaws that plague the traditional stack [3], and presented Loris [4], our complete redesign of the storage stack. Our first prototype, which we refer to henceforth as Loris-V1, had the "one file volume per device" bond, similar to traditional file systems. In this paper, we add support for file volume virtualization to the Loris stack. In doing so, we present *File pools*, a new model for managing storage devices. We show how the new model simplifies management by automating several mundane chores, supports heterogeneous device configurations, and provides file volume virtualization in Loris. We then show how our unified infrastructure, with a modular division of labor among layers, supports 1) instantaneous snapshotting of both files and file volumes, 2) efficient deletion of snapshots by sharing information between layers, and 3) version creation policies, like open-close versioning, on a per-file basis. We also present *Version directories*, our unified interface for browsing file history information. We will show how version retention policies can be implemented as simple shell scripts.

The rest of the paper is organized as follows. Sec. 2 outlines problems caused by the compatibility-driven, block-level integration of volume managers. In Sec. 3, we present a quick overview of Loris. Sec. 4 presents the design of file volume virtualization in Loris. In Sec. 5, we present the design of efficient file volume snapshotting in Loris. Sec. 6 presents a modular division of labor in Loris that integrates support for both individual file snapshotting, and open-close versioning. Sec. 7 presents our virtual directory interface. We then evaluate Loris using a series of micro and macro benchmarks in Sec. 8. An in-depth comparison of Loris with other systems is presented in Sec. 9. We finally discuss future work in Sec. 10 and conclude in Sec. 11.

## II. Problems with existing approaches

In this section, we will outline the problems that plague the traditional approach to file volume virtualization. Several commercial and research projects have taken other approaches for virtualizing file volumes. A detailed comparison of our approach with other competing approaches is presented in Sec. 9. We will now present problems along two dimensions, namely, flexibility and heterogeneity.

### A. Lack of Flexibility

An ideal storage stack must 1) provide flexible configuration and management of devices, and 2) support policy assignment at a range of granularities, from individual files or file types, to entire file volumes. In this section, we will highlight how inflexibility in the traditional stack complicates both device management and file management.

*1) Complicated device management:* Traditional volume managers used the level of indirection introduced by logical devices to support new functionalities, like file volume snapshotting and cloning. However, this level of indirection also introduced additional administrative operations. Even a simple task, such as adding a new disk to an existing installation, requires a series of steps, at least one for each level in the stack, to be performed by the administrator. This is because, any change in device configuration results in changes, not only in the volume manager's data structures, but also in file system data structures (for instance, any change in the size of a logical disk requires changes to the file system block management data structures), as file systems continue to work with the one file volume per logical disk assumption. Each and every one of these newly added steps is error prone, and a simple error could result in extensive data loss [5]. An ideal system would allow the administrator to just state the intent, like "add a new disk to an existing installation for increasing storage space," and automate implementation details (like expanding volumes). Traditional block-level volume management fails to meet this requirement.

*2) Coarse-grained file management:* Administrators manage data at the granularity of file volumes. For instance, an enterprise administrator could create one file volume per project, and encrypt certain file volumes, while compressing others. Administrators also take snapshots of entire file volumes, and use the snapshot for initiating periodic backups. Thus, at the enterprise level, policy specification at the granularity of file volumes is required. Block-level volume managers can easily provide such policies at a file volume granularity [22].

However, end users tend to associate policies with individual files or file types. The set of files over which a policy must be applied is typically much smaller in number than a file volume. For instance, a user might want open-close versioning on a source file, and no versioning for an object file. Thus, end-users require the ability to specify policies on a per-file basis. Since traditional volume managers operate below a strict block interface, they are semantically unaware, and thus are unable to provide fine-grained file management.

### B. Lack of support for heterogeneous devices

An ideal volume virtualization solution should be designed to work with heterogeneous device types. In this section, we will explain why heterogeneity should be considered a first class citizen during system design. We will show how the traditional approach fails to support devices other than conventional disk drives with a block interface.

*1) Heterogeneity across device families:* New devices are emerging with completely new storage interfaces. A common approach to integrating these devices into the traditional stack involves building file systems for each device family [10]. These file systems communicate directly with the device, using device-specific interfaces. However, traditional volume managers can work only with file systems that translate file requests to logical block requests. As a result, the traditional approach of virtualizing file volumes at the block level is not portable across heterogeneous device families.

*2) Heterogeneity within device families:* Even devices within the same family sometimes differ starkly in their performance characteristics. It is a well known fact that SSDs can be optimized to have different performance characteristics depending on certain firmware design choices [2]. For instance, Intel X25-V SSD provides very good random write IOPS, but its sequential write throughput suffers due to a price/performance tradeoff (only half the channels are populated with NAND). As a result, X25-V provides the same throughput for both large sequential writes, and small random writes [20]. Intel X25-M, on the other hand, provides higher throughput under large sequential writes than small random writes. While it would certainly be beneficial to opt for a log-structured layout on X25-M, it would provide little benefit when a X25-V is used, as it delivers the same throughput for both sequential and random writes. It might even prove to be detrimental due to the unnecessary cleaning overhead.

Thus, device-specific layout requirements create heterogeneity even within device families. Accommodating this kind of heterogeneity is impossible with the traditional stack, where multiple file systems, with even competing layout designs, could share the same physical device. Thus, any layout specific optimizations employed by file systems are rendered futile.

## III. The Loris storage stack

In prior work, we highlighted several issues that plague the traditional storage stack. We proposed Loris, a fresh redesign of the stack and showed how the right division of labor among

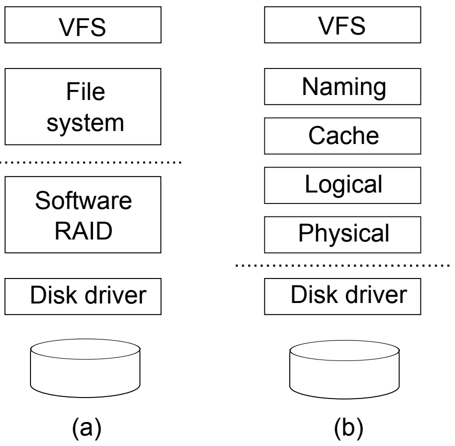

**Fig. 1:** The figure depicts (a) the arrangement of layers in the traditional stack, and (b) the new layering in Loris. The layers above the dotted line are file aware; the layers below are not.

layers in Loris solves all problems by design. In this section we provide a brief overview of Loris.

Loris is made up of four layers as shown in Figure 1. The interface between these layers is a standardized file interface consisting of operations such as *create*, *delete*, *read*, *write*, and *truncate*. In addition to file operations, the interface also contains *attribute* manipulation operations—*getattribute* and *setattribute*. Attributes are associated with files and have two purposes in Loris: 1) they enable information sharing between layers, and 2) they store out-of-band file metadata. We will now detail the division of labor between layers in a bottom-up fashion.

### A. Physical layer

The physical layer implements device-specific layout schemes, and provides persistent storage for files and their attributes. It exports a "physical file" abstraction to its client, the logical layer, and by doing so, abstracts away any device-specific interfaces or protocols. Each storage device is managed by a separate instance of the physical layer, and we call each instance, a *physical module*. The physical layer is also in charge of data verification. Being file aware, physical layer implementations can support parental checksumming [12]. By being the lowest layer in the stack, the physical layer verifies both application requests, and requests from other Loris layers alike, thus acting as a single point of data verification.

Our Loris-V1 physical layer implementation was based on the original MINIX 3 file system layout scheme [21], with added support for parental checksumming. Inodes are used to realize the physical file abstraction, and files are referred to using their inode numbers. Each inode stores a fixed number of persistent attributes, as well as 7 direct, one single indirect and one double indirect *safe block pointers*. Each safe block pointer contains a 32-bit block number as well as a checksum of the data block. Single and double indirects also store such pointers. Blocks belonging to both the block bitmap and inode bitmap are checksummed, and their checksums are stored in *block bitmap file* and *inode bitmap file* respectively. Each inode is individually checksummed, and these checksums are stored in the *root file*. The checksum of the root inode itself is stored in the superblock. Thus, the resulting parental checksumming hierarchy, as shown in Figure 2, ensures that all blocks, both data and metadata, are verified without any exceptions.

### B. Logical layer

The logical layer's responsibility is to combine multiple physical files and provide a virtualized *logical file* abstraction. It also supports RAID algorithms on a per-file basis. From the point of view of the cache layer, the logical layer's client, a logical file appear to be a single, flat file. Details such as the RAID algorithm used, and the physical files that constitute a logical file, are all abstracted away by the logical file abstraction.

The central data structure in the logical layer is the *mapping file*. The mapping file stores an array of entries, one per logical file, each containing the *configuration information* of that file. This configuration information is 1) the RAID level used, 2) the stripe size used, and 3) the set of physical files that make up the logical file, each specified as a physical module ID and inode number pair. For instance, a file mirrored on two devices could have the following configuration information in its mapping entry: F1=<raidlevel=1, stripesize=INVALID, physicalfiles=<D1:I1, D2:I2>>. The entry tells the logical layer that this file is a RAID1-type file, and inodes I1 on module D1, and I2 on module D2, form the physical files that store F1's data. The mapping file is itself a logical file with a static configuration. The same static inode number is reserved on all physical modules, and the mapping file is mirrored across all these physical files for improved reliability.

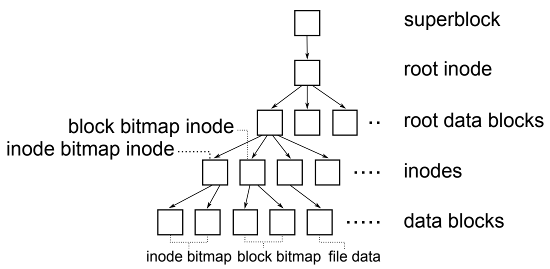

**Fig. 2:** Parental checksumming hierarchy used by the physical layer prototype. With respect to parental checksumming, the two special bitmap files are treated as any other files. Indirect blocks have been omitted in this figure.

### C. Cache and naming layers

The cache layer's responsibility is to provide data caching. Our Loris-V1 cache layer is file aware, and it performs data readahead and eviction on a per-file basis. The cache layer uses a static set of buffer pages to hold cached data.

The naming layer acts as the interface layer. Our Loris-V1 naming layer implements the traditional POSIX interface, and translates virtual file system (VFS) requests into corresponding file operations. All POSIX semantics are confined to the naming layer. For instance, none of the layers below the naming layer know about directories, or the format of directory entries.

All other layers treat directories as regular files. The naming layer uses the attribute infrastructure in Loris to store POSIX attributes. The naming layer is also in charge of assigning a unique *file identifier* for each file at file creation time. This identifier is passed as a parameter in all file and attribute operations to identify the target file.

## IV. File volume virtualization in Loris

Like traditional file systems, Loris-V1 does not support virtualized file volumes. However, unlike traditional file systems, the logical layer supports RAID algorithms, and hence can work with multiple devices. Thus, Loris-V1 has a "one file volume per set of devices" bond. In this section, we detail the design and implementation of file volume virtualization in Loris. We first present *file pools*, our new storage model for simplifying and automating the management of devices. We then describe changes to the infrastructure for supporting file volume virtualization.

### A. File pools: Our new storage model

As we mentioned earlier, the standardized file interface above the physical layer abstracts away device-specific details from the logical layer. Thus, from the point of view of the logical layer, each physical module is a source of physical files. Thus, multiple physical modules can be combined together to form a collection of files we call a *file pool*.

*1) Simplified device management:* File pools are the unit of storage management. A Loris installation can have one or more file pools. Administrators create a file pool by specifying the set of devices that form the pool. Each device can be a part of only one file pool. Multiple file pools can be created to provide performance isolation for each pool. For instance, an enterprise administrator could create two file pools, each having its own dedicated set of devices, to host the departmental file server and web server.

New devices can be added to, and old devices be removed from, existing file pools. Addition/removal of devices to/from a file pool is completely automated. When a device is added to a pool, a device-specific physical module is started. This new physical module registers itself with the logical layer as a new source of files. Once registration is complete, the logical layer can start creating new files on this module. Thus, unlike traditional volume managers, adding a new device to an existing file pool is a single step process, and space on the newly added device is immediately available for use.

Any device can be removed from a file pool by just moving all the files on that device to a spare device, or in some cases, even distributing the files among other existing devices. Since the logical layer has complete knowledge of the file-device mapping, supporting this is trivial. Furthermore, since the logical layer is file aware, copying file data ensures that only live data is moved over to the spare disk. This is a huge benefit compared to block-level volume managers, which do a block by block copy of the entire disk due to the absence of block liveliness information [19].

*2) Supporting heterogeneous installations:* As mentioned earlier, when a device is added to a file pool, a device-specific physical module is started. Since the physical module exposes a physical file abstraction to its clients, it can completely abstract away heterogeneous device interfaces. As the physical module is in charge of providing device-specific layout, it is possible to support different layout schemes for different devices even within the same device family. Thus, the file pool model permits pairing devices with customized physical modules, thereby exploiting heterogeneity both within and across device families.

*3) Thin provisioning with file pools:* We will describe support for file volumes in detail in the next section, but we would like to point out now that file pools also make thin provisioning [6] of file volumes possible. Thin provisioning refers to the ability to create dynamically filled, sparse file volumes. This ability can be used as a planning tool to determine new storage requirements, and hence derive a storage budget. With the file pool model, storage space need not be reserved for file volumes ahead of time. As a result, administrators can create multiple file volumes without committing physical storage space. As users create files and directories in these file volumes, storage space is automatically allocated from the set of physical modules that constitute the file pool. As we will see later, snapshots and clones also utilize this functionality. Multiple snapshots of a file volume can coexist together, but most files and data blocks will be shared among snapshots. Thus, file pools provide natural support for thin provisioning.

### B. Infrastructure support for file pools

We extended the logical layer to support the new storage model. The new logical layer can be considered to be made up of two sublayers, namely, the *volume management/RAID sublayer*, and the *file pool sublayer*. The file pool sublayer sits below the volume management sublayer and provides device management services, like creating and deleting file pools, adding and removing devices from existing file pools, etc.

The file pool sublayer is also responsible for satisfying file allocation requests from the shared pool of files it manages. It can employ several algorithms for satisfying file allocation requests. For instance, it could maintain a utilization summary of each physical module and provide load leveling, or it could monitor file access patterns and provide workload-aware file allocation. Our file pool implementation just rotates file allocation requests across physical modules. Exploiting device-specific characteristics, and matching device types with file types is a part of ongoing research.

The volume management sublayer operates above the file pool sublayer. We will describe the design details of this sublayer in the next subsection, but it suffices to say now that it supports RAID and volume management algorithms. It utilizes the allocation services of the file pool sublayer to satisfy file allocation requests. For instance, when a new file create request arrives at the logical layer, it is forwarded to the volume management sublayer. The volume management

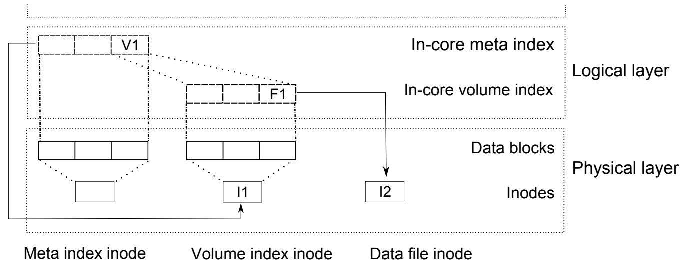

**Fig. 3:** The figure shows the relationship between meta index and volume index, the two datastructures that support file volume virtualization in Loris. The meta index file itself is a static file, and static inode is reserved to store its data(an array of file volume metadata entries). The file volume metadata entry for volume V1 shown in the figure could be <V1, REGULARVOL, volume index configuration=<raidlevel=1, stripesize=INVALID, physicalfiles=<D1:I1>>. Thus, inode I1 in physical module D1 is used to store the volume index file data (an array of logical file configuration entries) for file volume V1. The logical file configuration entry for file <V1, F1> could be <raidlevel=1, stripesize=INVALID, physicalfiles=<D1:I2>>. Thus, inode I2 in physical module D1 is used to store file data.

sublayer makes a file create request to the file pool sublayer, passing the number of physical files required (depending on the RAID type chosen for the file) as a parameter. The file pool sublayer allocates the required number of physical files and returns physical module–inode number pairs, which the volume management sublayer then records in its data structures.

### C. *Infrastructure support for file volume virtualization*

As described earlier, the mapping file in the logical layer stores configuration information for each logical file, and multiplexes file requests across physical files. We decided to extend the logical layer to support multiple virtualized file volumes, since it was a natural extension to the logical layer's data structures.

In the new infrastructure, multiple file volumes can be created in a single file pool, and multiple files can be created in each file volume. As a result, each file is now referenced in Loris using a pair of identifiers, namely, a new *volume identifier*, and the traditional file identifier. The logical layer assigns a new volume identifier to each file volume during volume creation time. Associated with each file volume is a *volume index* file. The format of this file is identical to the original mapping file. Each volume index file contains an array of entries, one per logical file belonging to that volume, and each entry contains contains configuration information, similar to the mapping file's configuration information we described earlier. The volume index file is also mirrored on all physical modules belonging to the file pool for improving reliability.

Since volume index files are created during volume creation, the configuration information of the volume index file itself is not static, that is, the volume index file can use inodes with different inode numbers on different physical modules. Hence, this configuration information is stored in the *meta index* file, with other file volume metadata. Each file pool has only one meta index file, which is also mirrored on all physical modules for improving reliability. This file contains an array of entries, one per file volume, containing *file volume metadata*. This metadata consists of 1) configuration information for the file volume's volume index file, 2) the type of the file volume, and 3) the volume id for this file volume. Thus, while the volume index file tracks files within a volume, the meta index tracks file volumes themselves. When the new logical layer receives a call to perform any file operation, it uses the volume identifier to first retrieve the volume metadata from the meta index file. After this, it uses the file identifier to retrieve the target file's configuration information, following which, it performs the requested operation. Thus, these two data structures make it possible for multiple file volumes to share a file pool, as shown in Figure 3, effectively breaking the one file volume per set of devices bond.

## V. New functionality: file volume snapshotting in Loris

Loris supports an extremely flexible snapshotting facility. snapshotting is efficient and instantaneous in Loris. We added a new *snapshot* operation to the standardized file interface described earlier. The operation carries a parameter, which is either the target file identifier for an individual file snapshot, or the file volume identifier for a file volume snapshot.

### A. *Division of labor*

Space-efficient snapshotting requires fine-grained, block-level data sharing to avoid making unnecessary copies of unchanged blocks. After investigating several possibilities, we assigned the responsibility for providing data sharing to the physical layer. This labor assignment maximizes storage efficiency without sacrificing modularity, as it is possible to support different physical layer implementations, with different mechanisms for data sharing, without affecting the logical layer algorithms. Thus, each physical layer must provide support for physical file snapshotting. In this section, we will describe two such physical layer implementations—a copy-based physical layer that lacks storage efficiency, but is extremely simple to implement, and a copy-on-write physical layer that supports fine-grained data sharing.

With individual file snapshotting and data sharing mechanism provided by the physical layer, the logical layer acts as a policy engine. It decides when a snapshot operation should be invoked on which physical file, and supports file volume snapshotting using individual file snapshotting provided by the physical layer. In this section, we will describe the logical layer data structures that support file volume snapshotting after describing the two physical layer implementations.

### B. *Physical layer(1): Copy-based snapshotting*

To implement copy-based snapshotting, we retained the layout design of the Loris-V1 physical layer, and we added support for the new snapshot operation. In both copy-based and copy-on-write-based physical layers, we distinguish between two types of inodes, namely, *current inodes*, and *snapshot inodes*. Current inodes can be considered to be the active version of a file which is used to satisfy normal read/write requests.

Snapshot inodes, on the other hand, are read-only, historical versions, that act as point-in-time snapshots of a current inode. A snapshot inode is created as a result of a snapshot operation on a current inode. We will now describe how snapshot creation and deletion work in the copy-based physical layer.

*1) Snapshot creation:* Since the physical layer is responsible for storing both data and attributes (POSIX attributes for instance), it must preserve their old values after a snapshot. So, the copy-based physical layer performs the following steps during a snapshot call. It first retrieves the inode corresponding to the inode number passed in as a parameter to the snapshot call, which we will refer to henceforth as the target inode. It then allocates a new inode, and copies over all the attributes from the target inode to the new inode. Following this, data belonging to the target inode is also copied over, allocating new data blocks during the process, to the new inode. Thus, after a snapshot operation, the new inode and target inode are independent copies, *not sharing* any data blocks. Finally, the new inode number is returned back to the the logical layer. From here on, the target inode becomes a snapshot inode, and the new inode becomes the current inode. It is important to note here that the level of indirection provided by the logical layer makes it possible to switch inodes without changing the file identifier. As a result, higher layers in the stack, like the naming layer, can continue using the same file identifier even after a snapshot operation.

*2) Snapshot deletion:* Deletion of a snapshot inode is a trivial operation. Since no data is shared between snapshots, the deletion operation deallocates all data, single and double indirect blocks, and then the inode itself, by marking them free in their corresponding bitmaps. Thus, while copy-based snapshotting suffers from inefficient storage utilization, its conceptual simplicity makes it a good mechanism for some personal and enterprise computing environments, where accesses to small files dominate the workload.

## *C. Physical layer(2): Copy-on-write-based snapshotting*

Our copy-on-write layout is a natural extension of the Loris-V1 layout. There are two major requirements for supporting copy-on-write-based snapshots as we will see in this section. The first requirement is that, for each data block, we need to identify if the block is shared with a previous snapshot. This is required to be able to perform a copy-on-write operation *only* when required. Second, for each snapshot inode, we need to know the chronological successor and predecessor to provide efficient snapshot deletion.

To support the former, we changed the definition of a safe block pointer. As we mentioned earlier, both inodes and indirect blocks contain a number of safe block pointers, and each safe block pointer contains a 32-bit block number–checksum pair. On an installation with a 4 KB block size, the largest disk size that can be supported with a 32-bit block number is 16 TB. For our prototype, we borrowed a bit from the block number, and the resulting safe block pointer contains a 31 bit block number, a 1- bit status field, and the block checksum. The resulting layout can now support a maximum disk size

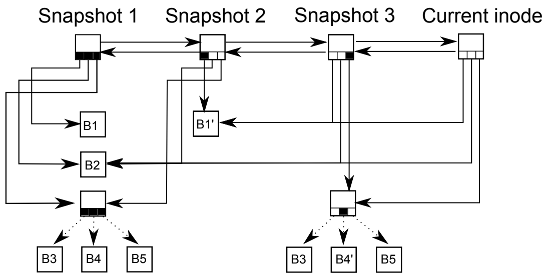

**Fig. 4:** The figure shows physical layer data structures after performing the following operations: 1) Create file (write B1, B2, B3, B4 and B5) 2) Snapshot file 3) Modify B1 to B1' 4) Snapshot file. 5) Modify B4 to B4'. 6) Snapshot file. A black rectangle represents a status bit that is set (an unshared block), and a white rectangle represents a cleared status bit(a shared block).

of 8 TB. Setting the status bit marks a data block as being newly allocated in this snapshot, and hence unshared with the previous snapshot. Clearing the status bit marks a data block as shared with the previous snapshot. We adopted this approach of borrowing a bit only for reducing the implementation effort. It is always possible to make the safe block pointer larger and overcome this space limitation. To maintain a chronological relationship between snapshots, we added two new fields to the inode, the *previous snapshot* and *next snapshot*. These fields are the inode numbers of the previous and next snapshot inodes respectively, and they link snapshots together in a bidirectional list.

*1) Snapshot creation:* When the copy-on-write physical layer receives a snapshot request, it performs the following steps. It first allocates a new inode, and copies over all the attributes from the target inode, just like the copy-based snapshotting approach. However, unlike the copy-based approach, it then copies over only the safe block pointers from the target inode, instead of allocating new blocks. While copying over the safe block pointers, it clears the status bit in each pointer to indicate that the data blocks are shared between the new and snapshot inodes. The physical layer then adds both inodes to the bidirectional list of snapshots by setting the previous and next snapshot fields. Finally, it returns back the new inode number to the logical layer. From here on, the target inode becomes a snapshot inode, and the new inode becomes the current inode. Figure 4 illustrates the snapshot operation with an example.

*2) Copy-on-write mechanism:* When the physical layer receives a write request, it first retrieves the target inode. For each block being written, the physical layer then retrieves the corresponding safe block pointers from either the inode, or from one of the indirects. If the status bit in the safe block pointer is cleared, the data block is shared with the previous snapshot. Hence, the physical layer allocates a new block, and the data is written out to this new location. However, if the status bit is set, no allocation happens, and the data is written to the block address contained in the block pointer.

If a new data block was allocated, the physical layer must update its corresponding block pointer in either the inode, or one of the indirects, to reflect 1) the new location of this data

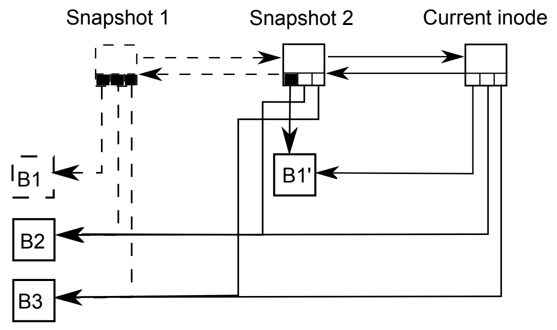

**Fig. 5:** The figure shows the physical layer data structures after the following operations are performed: 1) Create file 2) Snapshot 3) Modify B1 4) Snapshot 5) Delete the first snapshot

block, and 2) the new unshared state of the block by setting the status bit. If the safe block pointer is in the inode, updating this information is trivial, as shown for snapshot 2 in Figure 4. However, if the block pointer is in an indirect block, then the physical layer must allocate a new indirect, since the old one is being referenced by the previous snapshot.

In Figure 4, snapshot 3 illustrates the indirect block update process with an example. First, the old indirect block pointed to by the inode is read in. Following this, a new indirect block is allocated, and it is populated with safe block pointers from the old indirect block. During this process, the status bits in the safe block pointers are cleared. Following this, the safe block pointer for the newly allocated data block is updated in the indirect block. Finally, the safe block pointer for the indirect itself is updated in the inode. As shown in the figure, by clearing the status bit for all shared data pointers, we postpone block allocation until the very last moment, thereby providing efficient data sharing. As allocation of blocks takes place in a dynamic fashion, the amount of space taken by a snapshot is proportional to the amount of data overwritten.

*3) Snapshot deletion:* Deleting a snapshot inode is more complicated, since blocks pointed to by an inode could be shared with other snapshot inodes. A block can be freed only if it is not shared with any other inode. We use two facts to help us make this decision. For any given snapshot inode,

1) any data block not shared with the immediate predecessor is also not shared with any other predecessor.
2) any data block not shared with the immediate successor is also not shared with any other successor.

Thus, a block that is not shared with the immediate predecessor and successor snapshots are blocks that are unshared with any other snapshots, and hence by definition, are blocks that can be deleted. We can easily find blocks of the former kind using the status bits in the safe block pointers associated with the target inode. We can find blocks of the latter kind by reading in the safe block pointers associated with the target's successor inode, and examine their status bits. However, there are two interesting boundary conditions that deserve a special mention.

The first condition is when a file is truncated in the successor. In such a case, some of the successor's safe block pointers would be invalid, as the truncation code zeroes out these block pointers (by setting them to NOBLOCK). The second condition occurs when the target of deletion is the head of the snapshot list. Consider the situation depicted in Figure 5. When the first snapshot is deleted, block B1 is freed, but B2 is not freed as it is shared with the second snapshot. After deleting the first snapshot, the second snapshot is at the head of the snapshot list. However, the status bit for block B2 is cleared in the snapshot inode, as B2 was not overwritten during the snapshot lifetime. Thus, if the second snapshot is deleted, B2 will not be freed. Thus, considering both boundary conditions, we adopt the following algorithm for deleting snapshot inodes.

```
for each block offset in inode_being_deleted do
  if pointer_in_successor.block_number = NOBLOCK or
  pointer_in_successor.status = 1 then
    if pointer_in_deleted_inode.status = 1
    or inode_being_deleted.predecessor = NONE then
      delete the block
    end if
  end if
end for
```

## D. File volume snapshotting in the logical layer

Having explained the mechanism for block sharing in the physical layer, we will now explain the snapshot operation at the logical layer. We will also show how information sharing between the logical and physical layers makes it possible to support efficient snapshot deletion.

In order to support snapshotting, each logical file is associated with a *file epoch number*. This epoch number is stored together with other logical file configuration information in the corresponding volume index file. Each file volume is also associated with a *volume epoch number*. This epoch number is stored together with other file volume metadata in the meta index file. Each file volume metadata entry also contains previous and next snapshot fields. These fields store the volume identifiers of preceding and succeeding snapshot volumes, thus linking snapshots in a bidirectional list, similar to the inode snapshots in the physical layer. As we will see later, version directories utilize this bidirectional linking to enumerate the list of snapshots.

*1) Snapshot creation:* When the logical layer receives a request to snapshot a file volume, which we will henceforth refer to as the target volume, it first retrieves the volume metadata from the meta index file. The logical layer then creates a new *snapshot volume*. A snapshot volume is a read-only file volume. No file operations, except read and getattribute, are permitted on any files in a snapshot volume. The type field in the volume metadata indicates whether a volume is a regular or a snapshot volume. The process of creating a snapshot volume involves 1) assigning a new volume identifier, 2) allocating an entry for storing the new volume's metadata in the meta index file. Following this, the logical layer copies over all metadata fields from the target volume entry to the snapshot volume entry. Once this step is completed, both the

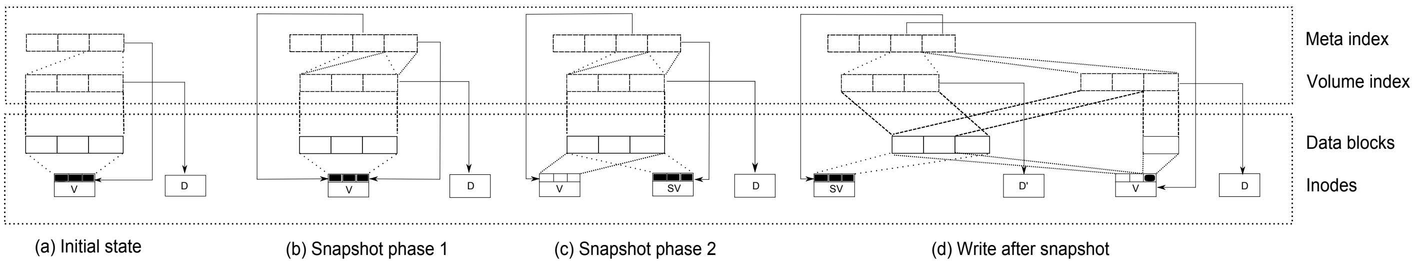

**Fig. 6:** The figure shows the different phases of a snapshot operation, and the interaction between logical and physical layer data structures during, and after a snapshot operation. A block labeled D in the figure represents a data file inode, V represents a current/regular volume's volume index inode, and SV represents a snapshot volume's volume index inode. Arrows in the figure connect a file volume metadata entry with the volume index inode, and a logical file configuration entry with its data file inode. Blocks in the logical layer show the logical layer's view of the meta and volume index files, while blocks in the physical layer represent the actual blocks storing data corresponding to those file. Dotted lines between the two layers show the mapping between the two views. The figure is divided into four parts. Part (a) shows the state of the stack before a snapshot. Part (b) shows the first phase of the snapshot operation, where a new snapshot volume is created. Part (c) shows the second phase of the operation, where a snapshot of the volume index file itself has been performed. Part (d) shows a new data file inode being allocated due to a write operation after a snapshot, and the new volume index file pointing to the new data file.

snapshot and target volumes share the same volume index file. Figure 6(b) illustrates the first step with an example.

Next, the logical layer snapshots the volume index file. It does this by making a snapshot call to each physical module, passing in the relevant inode number as a parameter. Each physical module snapshots the inode as explained in the previous section, and returns back a new inode number. The logical layer updates only the target volume's volume index configuration information with this new inode number. At this point, the snapshot volume's configuration points to the old volume index inodes, and the target volume's configuration points to the new inodes, as shown in Figure 6(c). Finally, the epoch number in the target volume is incremented by 1 to reflect a completed file volume snapshot operation. Thus, snapshotting is an instantaneous operation, and it involves only a snapshot call to freeze the volume index file.

snapshotting of individual files happens dynamically at the next operation that modifies the data or metadata of the file. When the logical layer receives a write, delete, setattr or truncate operation on a file, it compares the file's epoch number with the corresponding file volume's epoch number. If the file has a smaller epoch number, the logical layer snapshots the file before performing the operation. Read, create and getattr Loris operations do not incur this check, as they do not modify a file or its metadata in any way. For instance, a read request from the application gets transformed into a Loris read operation to retrieve the data, and a Loris setattr operation to update the access time. While the Loris read operation bypasses the check, the setattr operation results in a snapshot being created. Finally, the file's epoch number is set to the file volume's epoch number. Future operations on the file proceed without snapshotting the file again as the epoch number test fails. The logical layer snapshots a file by calling the snapshot operation on all associated physical modules. The file's configuration information is updated with the new inode numbers returned by the physical modules. When the data block containing this new file entry is written out to the physical layer, the data sharing mechanism in the physical layer ensures that only the current volume index file stores this new configuration. Thus, as illustrated in Figure 6(d), any snapshot of a file is reachable through that snapshot volume's volume index.

*2) Efficient deletion support through information sharing:* The algorithm for deleting snapshot volumes in the logical layer is very similar to the algorithm for deleting blocks in the physical layer. In the pseudocode for deleting snapshots given below, target snapshot refers to the snapshot volume being deleted.

```
for each file in the target snapshot do
    if file does not exist in the next snapshot or file has been
    modified in the next snapshot then
        if file has been modified in the target snapshot or target
        snapshot has no preceding snapshot then
            call delete on this file's physical modules
        end if
    end if
end for
```

Similar to block deletion, files that are not shared between snapshots are the files that are deletable. A file is not shared by two snapshots if it has been modified between snapshots. As mentioned earlier, a file is modified if the file receives a setattribute, delete, truncate or write operation after a file volume snapshot. Any such modification operation results in a new file entry, with a new configuration information, and an updated epoch number. Thus, unshared files are ones for which file epoch number is the same as the snapshot volume's epoch number. Each such file identified by the algorithm is deleted by making a delete call to each corresponding physical module, which deletes the snapshot inode as mentioned earlier.

It is a well-known fact that file access distribution is heavily skewed, with a very small percentage of files getting a large percentage of accesses [16]. Thus, it is very likely that only a relatively small number of files, and hence file entries, are modified between any two snapshots. The deletion efficiency could be improved significantly if we process only these changed file entries. Thinking about this, we realized that the copy-on-write-based physical layer already stores this information in the form of status bits associated with each block. Thus, we added a new operation which the physical layer could use to communicate a snapshot inode's modified block offsets to the logical layer. When the logical layer receives a delete request, it retrieves the file volume configuration information as usual. It then makes a call to retrieve the set of modified file offsets for the snapshot's volume index file. Equipped with this information, the logical layer executes the algorithm mentioned earlier, but only for file entries in these modified offsets thus avoiding a linear scan.

## VI. New functionality: Unifying file snapshotting and version creation policies

In this section, we will describe the infrastructure support for providing per-file snapshotting and open-close versioning. Per-file snapshotting and open-close versioning introduce an interesting problem in the design of file volumes. With both these features, multiple snapshots of a file can be taken between any two file volume snapshots. Each such snapshot requires the configuration information at the time of snapshot to be recorded. In our file volume design, each configuration information is always tracked by a volume index belonging to either a snapshot volume or a current volume. For instance, after the first operation that modifies file data or metadata following a snapshot, the snapshot volume's volume index entry tracks the old inodes that existed at the time of snapshot, as we already described earlier. However, with individual file snapshots, no volume index is available to track multiple snapshot configurations. Hence, without added support, we lose the ability to access all individual snapshots created between two file volume snapshots.

Solving this problem requires a way to track version history, in the form of configuration information for each file snapshot. Thinking about this, we realized that we could create a new file volume for each file, and use its volume index file to store this configuration information.

### *A. Version volumes*

To support per-file snapshotting, we define a new volume type, called *Version volume*. The fundamental idea behind version volume is to group all individual file snapshots together in a file volume. Each logical file in Loris can be conceptually seen as being associated with a version volume, and each version volume stores the version history of its parent file. A version volume is linked to its parent file by storing its volume identifier with the file's configuration information. We will now explain how version volumes are created, and how they support per-file snapshotting.

Initially, all files start out without a version volume. A version volume is created during the first individual file snapshot operation. Creating a version volume is similar to creating a regular volume–a new volume identifier is assigned, a new metadata entry is created in the meta-index file, a new volume index file is created, and its configuration information is stored with other details in metadata entry. An important piece of metadata specific to version volumes is the *next version number* field. This field is incremented every time a new file version is added to the version volume. The logical layer also stores the version volume's identifier with the file's configuration information, thereby linking the file with its version volume.

Every individual file snapshot operation proceeds as follows. The target file's configuration information is retrieved from the corresponding volume index file. The version volume's next version number is incremented, and this value is used to determine the version volume's volume index entry where this old configuration information is stored. Following this, a snapshot call is made to all associated physical modules, and the target file's configuration information is updated in the current volume. Figure 7 illustrates this with an example.

Figure 7 illustrates the interaction between a file volume snapshot and an individual file snapshot. Version volumes are used only to store configuration information for snapshots created by user-initiated per-file snapshots, or auto-generated open-close versioning based snapshots. File snapshots created as a side effect of a file volume snapshot are tracked by the respective snapshot volume's volume index.

While it might appear at first thought that creating a file volume for each file might be expensive, such is not the case due to several reasons. First of all, files start out initially without a version volume. As we mentioned earlier, version volumes are created on-the-fly during the first individual file snapshot operation. Thus, all files that are not modified after a snapshot do not have version volumes. Second, each volume requires space proportional to the number of configuration entries it stores. As version volumes are created on a per-file basis, only files that are snapshotted very frequently end up with version volumes containing many configuration entries. Third, as all individual file versions share unmodified data blocks using the physical layer's copy-on-write functionality, the only information that is not shared between versions is each version's configuration entry, which by itself is very small (roughly 100 bytes) compared to the modified data blocks. Thus, version volumes provide a light-weight mechanism for tracking file history.

### *B. Open-close versioning in the naming layer*

We will now illustrate how the same infrastructure that supports individual file snapshotting supports open-close versioning as well. The naming layer, being aware of open-close sessions, acts as the policy enforcement layer. It creates new versions of files that have been modified in an open-close session, by making a snapshot call following the close operation. The logical layer processes this snapshot call, as explained earlier, using its version volume infrastructure, and forwards the snapshot call to each associated physical module for snapshotting the inodes. The next setattr, write, truncate or delete operation on this file will result in the physical layer performing block-granular copy-on-write. Thus, as the naming layer only specifies policies, plugging in a different version creation policy, like provenance-based version creation [13], is very simple. The only code change required would be to

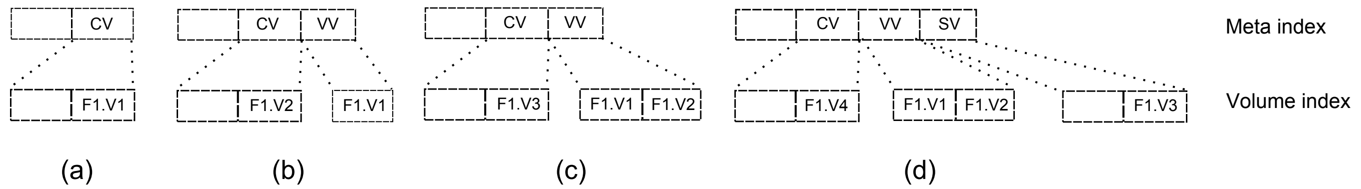

**Fig. 7:** The figure illustrates individual file snapshotting using version volumes. Part(a) in the figure shows a current volume(CV) containing configuration information for a file F1. Part (b) in the figure shows the data structures after an individual snapshot of file F1. It shows that a new version volume(VV) has been created, and F1's V1 configuration information is stored as an entry in VV. Part (c) shows the state after another individual file snapshot. Now, VV has two entries, one configuration per snapshot. Part (d) shows the state after the following operations: 1) snapshot volume CV, 2) write F1. A snapshot volume(SV) has been created, and it contains the configuration information for version V3 of F1. It is important to note that VV stores only individual file snapshot history(V1 and V2 of the file). File volume snapshot history (V3 in this case) is recorded by snapshot volumes, SV in this case.

figure out the exact time and place where a snapshot call must be made. Thus, using a unified infrastructure, we are able to support 1) per-file version creation policies, 2) per-file snapshotting, and 3) file volume snapshotting.

## VII. New functionality: Version directories– a unified interface for browsing history

snapshotting and versioning systems have provided several different interfaces for browsing file history information. The design requirements we had for our interface were threefold: 1) the interface should be simple and natural to use, 2) applications should be able to access file versions without any modifications, 3) the same interface should be used for accessing individual file snapshots, open-close version based snapshots, and file volume snapshots. We now present *version directories*, our new unified interface for browsing file history information.

### A. Version directories – interface specification

With most versioning systems, users can access file versions by suffixing a file name with a *version specifier*. The version specifier consists of a *syntax token*, which is a special character (like "!" in CedarFS [8]), and a *version sequence number*, which identifies the target version from a list of available file versions. Most snapshotting systems, on the other hand, require users to mount a file volume snapshot, or auto mount the snapshot at a designated mount point. We rejected the mount-based history access as it violated our third requirement.

In our interface, the "@" character acts as the syntax token. Thus, for a file foo, the name foo@N can be used to access the Nth version of foo. Version specifiers can also be used with directories to achieve *version inheritance*. Version inheritance refers to the mechanism by which files and directories are automatically scoped using their parent's sequence number. Version inheritance can be used to simulate a mount-based interface. For instance, if /home/user1 is a file volume, an administrator could create a symbolic link to /home/user1@1 at any location, and scope the entire subtree to the first snapshot.

Yet another interesting feature of this inheritance mechanism is its use in recovering deleted files. Since we do not support name versioning yet, once a file is deleted, its name is removed from its parent directory by the naming layer, the file entry is purged from the current file volume by the logical layer, and the inodes storing file data are deleted by the corresponding physical modules. However, a user could scope the parent directory to a snapshot that has the file name intact, and by inheritance, access the file version at that snapshot. For instance if a file foo has been removed from directory bar, the name bar/foo@2 cannot be used to access the snapshot version of foo, as we do not support name versioning yet. However, the name bar@2/foo could be used to achieve the same effect. As an aside, the name bar@2/foo@1 resolves to the same version as the name bar@1/foo, and the name bar@1/foo@2 resolves the same version as the name bar@2/foo. Thus, multiple scope specifiers can be used in a single path name.

Having described the interface for accessing old versions, we will now describe our interface to enumerate the list of all file versions. Most systems add an explicit library call, that in turn forwards an ioctl to a versioning file system to retrieve such details. Previous research has already suggested that overloading file system semantics improves uniformity when compared to creating new interfaces [7]. In our interface, by suffixing any file or directory name with *just* the syntax token, the user can treat it as a *version directory*. A version directory is a virtual directory, in the sense that its directory entries are created on the fly. Virtual directories meet all the three requirements we mentioned earlier: 1) It is a simple and natural technique, as it overloads a well known file system construct—directories, 2) one can use shell utilities and applications unmodified to access old versions, and 3) every snapshot—irrespective of how it is created—is presented as a virtual directory entry, thus providing an integrated access interface.

A directory entry enumeration operation on a version directory results in all versions of the target file being displayed to the user as individual file entries in the directory. For instance, a user could perform a "cd foo@", followed by an "ls –l" to view both dynamically generated file names and POSIX attributes of each individual version. Each file name in a version directory has two parts: a type specifier, and a version sequence number. The type specifier identifies whether the version was created by an individual file snapshot (SNAP), an open-close versioning snapshot (VERSION), or a file volume snapshot (VOLSNAP). As an aside, users can also access versions by using directory entries instead of suffixing file names with sequence numbers. For instance, /usr/foo.txt@1, and /usr/foo.txt@/VolSnap_1 resolve to the same file version.

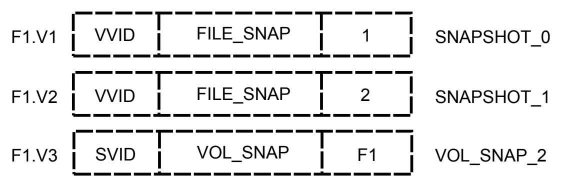

**Fig. 8:** The figure shows the vstat structures for file F1 shown in Figure 7(d). The columns from left to right contain the volume identifier, volume type, and file identifier for the file version shown to the left of the vstat structure. The names on the right of each vstat structure are the names assigned by the naming layer, for each virtual directory entry, when the file is treated as a version directory.

Another advantage of using version directories is the fact that retention policies can be implemented as simple shell scripts. For instance, a script that implements a number based retention policy could access each file as a version directory, and delete oldest N versions using standard UNIX utilities. A landmark based retention policy could diff two versions (diff foo@1 foo@2), and pick versions with minimal changes to discard. Furthermore, different policies can be applied to different files or file types, thus providing highly flexible version management. It is important to note here that only snapshots created by open-close versioning or individual file snapshotting can be explicitly deleted. File snapshots created as a side effect of volume snapshotting can be deleted only by deleting the file volume. Retention scripts can use the type specifier part of the file name to identify deletable versions.

### B. *Version directories – implementation details*

We will now describe the infrastructure support for implementing version directories. When an applications performs a directory listing operation on a virtual directory, the naming layer needs to enumerate the list of snapshots for the target file. Since the task of tracking snapshots is provided by the logical layer, a new *version stat* call was added to the standardized file interface to communicate this information to the naming layer. The naming layer makes the version stat call, passing in the target file identifier as a parameter.

When the logical layer receives a version stat call, it first pulls up the volume metadata from the meta index file. It then walks through the set of snapshots associated with this volume, using the previous and next snapshot fields we described earlier, and checks each volume index for the target file. For each valid configuration entry, the logical layer populates a new *vstat structure*. Each vstat structure contains several fields, like the volume identifier, volume type, and file identifier. After checking all snapshot volumes, the logical layer retrieves the file's configuration information from its current volume. If the configuration information records the presence of a version volume, the logical layer retrieves the version volume's volume index file. For each configuration entry in this volume index, it populates a new vstat structure. After processing all entries, the logical layer returns back the list of vstat structures to the naming layer, as shown in Figure 8.

The naming layer uses these vstat structures to build virtual directory entries in the version directory. It uses the volume type field to choose a type specifier, and a zero based counter to assign the version sequence number for each directory entry. When the user access a particular file version using one of these entries, the naming layer uses the version sequence number to retrieve the appropriate vstat structure. It then uses the volume and file identifiers specified in the vstat structure to identify the target file in any file operation.

This approach does have the disadvantage that a version's name might change as other versions are deleted. However, we really do not consider this to be an issue due to two reasons. First of all, as users can easily view POSIX metadata associated with each version, they can identify the target version using its metadata, thus obviating the need for consistent system-generated names. Second, if name-based identification is required, the proper approach would be to enable tagging of individual versions. Once users tag versions with user-friendly names, they can easily identify target versions using their tags. We are working on extending our system to support tagging of both individual file versions and file volume snapshots.

It is important to note in Figure 8, that the file identifiers in the first two vstat structures are not F1, the target file's identifier. Since these two versions were created by individual file snapshotting, their configuration information resides in the version volume's volume index. Since the version volume can be accessed as a regular file volume, one could directly access a version by using its position within the volume index as the file identifier. For instance, when the logical layer gets the identifier pair <VVID, 1>, it first retrieves the volume with identifier VVID, which in our case is the file's version volume. It then uses 1 as the file identifier, and retrieves the first file entry from the volume index, which would be the configuration information for the first file version. Thus, by grouping all file snapshots in a version volume, we are able to use the same mechanism for accessing file snapshots, irrespective of how they are created.

## VIII. Evaluation

In this section we will present our evaluation of the Loris prototype which supports all new functionality presented in this paper. We implemented our Loris prototype on the Minix 3 multiserver operating system [9]. We will first evaluate the overhead of open-close versioning and snapshotting using micro-benchmarks. We will then present an evaluation of the infrastructure using two macro-benchmarks, and show that our file volume virtualization approach has no overhead.

### A. *Test Setup*

All tests were conducted on an Intel Core 2 Duo E8600 PC, with 4 GB RAM, and one 500 GB 7200RPM Western Digital Caviar Blue SATA hard disk (WD5000AAKS). We ran all tests on 8 GB test partitions at the beginning of the disk. Loris was set up to work with a 32 MB buffer cache.

### B. *Copy-based and copy-on-write snapshotting comparison*

We will first compare the performance of copy-based and copy-on-write-based snapshotting using a custom micro-benchmark. The micro-benchmark stresses file system snapshotting by first creating either 500 1 MB files, or one 500 MB file, in a single file volume. Following this, we perform ten rewrite runs, where we truncate and rewrite all the files, snapshotting the entire file volume after each run. We measure the total time taken to overwrite all the files in each run, and the median of these ten values is shown in Table I. As the file volume is snapshoted, each rewrite run results in a new snapshot of all files being created.

| Benchmark | No Snapshot | Copy-based | Copy-on-write-based |
|---|---|---|---|
| 500 1 MB files | 7.30 | 17.95 | 7.31 |
| 1 500 MB file | 7.50 | 21.01 | 7.95 |

**TABLE I:** Time in seconds for file volume snapshotting using copy-based and copy-on-write-based physical layer implementations.

As shown in Table I, file volume snapshotting has very little overhead with the copy-on-write-based physical layer. The copy-based physical layer however has a significant overhead. This is due to the fact that each file is copied over in its entirety after every snapshot operation. As each copy operation reads and writes 1 MB per file snapshot in the first case, and 500 MB in the second case, it causes excessive delay in the mainline write path leading to poor performance.

## C. Open-close versioning evaluation

We will now present an evaluation of our open-close versioning implementation using the same micro-benchmark that was used to evaluate snapshotting, with a minor modification. We no longer snapshot the file volume at the end of each run. Instead, we enable open-close versioning for each file. As shown in Table II, the copy-based physical layer suffers due to the copying overhead, as we explained earlier.

| Benchmark | No versioning | Copy-based | Copy-on-write |
|---|---|---|---|
| 500 1 MB files | 7.30 | 21.26 | 10.53 |
| 1 500 MB file | 7.48 | 20.80 | 7.95 |

**TABLE II:** Time in seconds for open-close versioning using copy-based and copy-on-write-based physical layer implementations.

The copy-on-write-based physical layer on the other hand incurs an overhead only when 500 1 MB files are are individually versioned. Versioning of a single 500 MB file does not exhibit any overhead. We examined this further, and we found individual flushing of metadata blocks to be responsible for this performance loss. As all files are open-close versioned, every file has an associated version volume. Each version volume's volume index file contains logical configuration entries that track file history. Thus, for 500 files, there exist 500 version volumes, each having a volume index file containing one data block with logical configuration entries. In our current implementation, these 500 blocks are flushed using individual write operations. This results in multiple, small, random writes at the disk, and the resulting seeks result in performance loss. We are working on fixing this problem by vectoring these write requests in a single write operation using the vwrite call we introduced to solve a similar problem with small files [4].

## D. Overhead of file volume virtualization

We now evaluate the overhead of our new infrastructure using two macro-benchmarks: (1) PostMark, configured to perform 20,000 transactions on 5,000 files, spread over 10 subdirectories, with file sizes ranging from 4 KB to 1 MB, and read/write granularities of 4 KB, and (2) an application-level macro-benchmark, which we will refer to henceforth as *Applevel*, which consists a set of very common file system operations including copying (a complete MINIX 3 source tree), compiling (running "make clean world"), and running find and grep (searching for a keyword in all source and header files).

| Benchmark | Loris-V1 | Loris(new) |
|---|---|---|
| Postmark | 686.00 | 693.00 |
| Applevel (copy) | 124.00 | 134.00 |
| Applevel (build) | 112.00 | 113.00 |
| Applevel (find and grep) | 20.00 | 19.00 |

**TABLE III:** Transaction time in seconds for Postmark and wall clock time in seconds for Applevel tests

Table III shows PostMark and Applevel results for both Loris-V1 and our latest Loris version that supports all the new functionalities described in the paper. As can be seen, file volume virtualization in Loris has very little overhead, if any. Most other systems maintain elaborate block mapping information to virtualize file volumes, and hence suffer from performance degradation due to increased metadata footprint. As no such mapping information is maintained by Loris, there is no performance impact.

| Benchmark | No Snap | Copy Snap | COW Snap |
|---|---|---|---|
| Applevel (build) | 123.00 | 131.00 | 124.00 |
| Applevel (find and grep) | 21.53 | 38.68 | 21.60 |

**TABLE IV:** Wall clock time in seconds for applevel tests using copy-based and copy-on-write-based physical layer implementations.

Table IV shows an interesting comparison of the two snapshotting approaches using the Applevel benchmark. To perform this evaluation, we modified our test suite to take a file volume snapshot after the copy phase. As can be seen copy-on-write snapshotting does not suffer from any overhead in both build and find phases.

Copy-based snapshotting on the other hand suffers from a small overhead during the build operation, and a huge overhead during the find and grep operation. The find and grep operations result in the access time of all source and header files being updated. Since the access time is stored as an attribute by the physical layer, setting a new access time is done by making a setattr call. This call triggers a file snapshot operation at the logical layer. The copy-based physical layer copies over the entire file data to create a new current version, while the copy-on-write physical layer just allocates a new inode and marks data blocks as shared. This is the reason behind the poor performance of the copy-based physical layer. Turning off access time updates resulted in similar performance figures for both copy-based and copy-on-write physical layer implementations.

We are working on a new naming layer that provides structured data storage to applications. The new naming layer will provide a new directory storage and indexing scheme. It will also be responsible for storing attributes with directory entries, and providing snapshotting of attributes. With this new naming layer, the physical layer will be in charge of snapshotting only file data, and thus the copying overhead will not be incurred for attribute changes.

## IX. Comparison with other approaches

Several commercial and academic projects have taken other approaches toward virtualizing file volumes. We will first discuss device management alternatives, and compare file pools with other approaches. Then, we will discuss file volume virtualization, and present the advantages that Loris has over other techniques. We discussed logical volume managers in great detail earlier in this paper. Most block-level virtualization solutions suffer from problems similar to the ones mentioned in Sec.II, and so we will not discuss them in further detail here.

### *A. Device management*

Sun's ZFS [1] proposed refactoring the traditional storage stack to solve many problems we presented earlier. ZFS introduced storage pools, a new storage model for simplifying device management. Storage pools are based on the idea that block allocation decision is made by the wrong layer in the traditional stack—the file system layer. In the ZFS stack, a separate storage pool allocation (SPA) layer manages a pool of storage devices, and provides an interface for allocating and freeing virtual blocks, similar to the malloc() and free() interface for virtual memory. As the SPA provides a virtualized block address space, multiple file volumes can share a single storage pool. The SPA also simplifies and automates addition and removal of devices.

Our approach (file pools) offers advantages in addition to the benefits offered by storage pools. Providing RAID and volume management services at a file-level makes it possible to support advanced functionalities, like snapshotting, at several granularities, thus improving flexibility. Rather than bundling allocation and storage management together like ZFS, Loris makes a clean split between the two functionalities, by assigning block allocation to the physical layer, and storage management to the logical layer. The improved modularity makes it possible to support heterogeneous device configurations using custom layout designs without affecting RAID and volume management algorithms.

### *B. File management and file volume virtualization*

AFS[18] was one of the first projects to promote the use of file volumes as administrative units. AFS was a client-server system, and AFS clients accessed files using a <volume identifier, file identifier > pair similar to Loris. The volume identifier was used to locate the server housing the file volume. On the server side, many volumes share a single disk partition, and administrators could associate usage quotas with file volumes. File volumes were supported by modifying the 4.2BSD on-disk file system, to include per-volume inode tables that translated the <volume identifier, file identifier> pair to an inode. snapshotting was a nightly operation that was implemented by incrementing the link count on all inodes associated with a file volume, and block-level data sharing was not supported. In contrast, file volume virtualization Loris is layout independent, snapshotting is a flexible and instantaneous operation, and if required, a copy-on-write based physical module can be used to support fine grained block-level data sharing.

There are several file systems, both on-disk and stackable ones, that support snapshotting and versioning. Most versioning file systems, like ElephantFS [17], support only open-close versioning and do not support file system snapshots. File systems that do support both, like ext3cow [15] are on-disk file systems, and do not support virtualized file volumes. Stackable file systems like RAIF [11], and VersionFS [14] are extremely flexible, portable, and support open-close versioning and RAID algorithms on a per-file basis. However, unlike on-disk file systems, they suffer from performance problems due to double buffering, and data copying. Loris, on the other hand, implements portable file volume virtualization in the logical layer by relegating storage-efficient block-level data sharing to the physical layer. It supports flexible snapshotting and versioning with its policy-mechanism split. Thus, Loris has the advantages of all these systems with its modular division of labor without the disadvantages.

Flexol [6] is the file volume virtualization system from NetApp. The basic idea adopted by FlexVol is to virtualize file volumes by creating a file volume inside a file, in a lower file system. The recursive use of the WAFL file system provided file awareness to the virtualization layer, making it possible to support several advanced functionalities like snapshotting and cloning. However, the dual mapping information that needs to be maintained by FlexVol causes some performance degradation. Unlike FlexVol, Loris approach does not suffer from overhead due to metadata footprint as seen earlier. Furthermore, while FlexVol supports file volume snapshotting and cloning, it does not support individual file snapshotting or open-close versioning. We will discuss support for cloning in Loris in the next section.

## X. Future work

The design of file volume virtualization in Loris opens up several areas of future work. We will now discuss two main avenues of ongoing research.

### *A. Flexible cloning in Loris*

As we mentioned in the previous section, a number of commercial projects have introduced file volume cloning in addition to snapshotting. We are currently working on adding support for flexible copy-on-write-based cloning at both per-file and file volume granularities. The mechanism that supports snapshotting can also be used to support cloning with minor modifications. A new *clone* operation will be added to the standardized file interface. When the logical layer receives a request to clone a volume, it first picks a new volume identifier, and allocates a new metadata entry in the volume index file. It then instantiates a *clone volume*, that is a writable clone of the parent, by copying the target volume's metadata to this new entry. A new field in the snapshot volume's metadata links it with the new child volume. Following this, the clone volume's epoch number is set to one higher than its parent's epoch number. Individual files themselves are cloned on demand, just like snapshotting.

Copy-based snapshotting physical layer can support cloning without any modification, as each inode is an independent copy, not sharing any data blocks with other snapshots. However, supporting copy-on-write based cloning requires some changes to the physical layer to make sure that data blocks belonging to snapshot inodes are not freed while there are clones using those blocks.

### B. Hybrid file pools

As we mentioned earlier when discussing file pools, several file allocation algorithms can be employed by the file pool sublayer to satisfy file creation requests. In addition, we are investigating the addition of device-aware migration algorithms at the file-pool sublayer. Since the file pool manages devices of multiple types, it can collect aggregate performance statistics of these devices. Based on these metrics, it can classify each device as belonging to a particular device type category. Similarly, since the file pool sublayer works at a file-level, it can also collect file access patterns on a per-file basis, using which, it can classify each file as belonging to a particular file type category. An easily configurable rule table can then be used to match file types with device types.

As an example configuration, small, read-only files could be positioned on a device with good random read performance, while small, write-mostly files could be positioned on a device with a log-structured layout. Large files that are both randomly read and written, on the other hand, could use two physical modules, one with a log-structured layout for absorbing writes, and one on a device with good random read performance. The logical layer would direct writes to the write-optimized physical module, and migrate data in the background to the read-optimized device. We use the term *Hybrid file pools* to refer to this new storage model, as file pools can accommodate heterogeneous devices in hybrid configurations.

## XI. Conclusion

Virtualizing file volumes makes it possible to retain administrative flexibility without sacrificing storage efficiency. In this paper, we examined the traditional approach of virtualizing file volumes along two dimensions, namely, flexibility, and heterogeneity. We illustrated several problems associated with the traditional approach of virtualizing file volumes. We then presented our file volume virtualization design based on Loris, our fresh redesign of the traditional storage stack. We showed how Loris, with its unified, modular infrastructure, supports file volume snapshotting, per-file snapshotting, and open-close versioning.

## References

[1] Sun microsystems, solaris zfs file storage solution. solaris 10 data sheets, 2004.

[2] Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J. D., Manasse, M., and Panigrahy, R. Design tradeoffs for ssd performance. In *USENIX Ann. Tech. Conf.* (2008), USENIX Association, pp. 57–70.

[3] Appuswamy, R., van Moolenbroek, D. C., and Tanenbaum, A. S. Block-level raid is dead. In *Proc. of the Second USENIX Workshop on Hot topics in Storage and File systems* (2010), USENIX Association, pp. 4–4.

[4] Appuswamy, R., van Moolenbroek, D. C., and Tanenbaum, A. S. Loris - a dependable, modular file-based storage stack. In *Proc. of the The 16th IEEE Pacific Rim Intl. Symp. on Dependable Computing* (2010), USENIX Association, pp. 4–4.

[5] Brown, A. B., and Patterson, D. A. To err is human. In *Proc. of the First Workshop on Evaluating and Architecting System dependability* (2001).

[6] Edwards, J. K., Ellard, D., Everhart, C., Fair, R., Hamilton, E., Kahn, A., Kanevsky, A., Lentini, J., Prakash, A., Smith, K. A., and Zayas, E. Flexvol: Flexible, efficient file volume virtualization in wafl. In *USENIX Ann. Tech. Conf.* (2008), pp. 129–142.

[7] Gifford, D. K., Jouvelot, P., Sheldon, M. A., and O'Toole, Jr., J. W. Semantic file systems. In *Proc. of the 13th ACM Symp. on Oper. Syst. Prin.* (1991), pp. 16–25.

[8] Gifford, D. K., Needham, R. M., and Schroeder, M. D. The cedar file system. *Commun. ACM 31* (March 1988), 288–298.

[9] Herder, J. N., Bos, H., Gras, B., Homburg, P., and Tanenbaum, A. S. Construction of a highly dependable operating system. In *Proc. of the Sixth European Dependable Computing Conf.* (2006), IEEE Computer Society, pp. 3–12.

[10] Josephson, W. K., Bongo, L. A., Flynn, D., and Li, K. Dfs: A file system for virtualized flash storage. In *Proc. of the Eigth USENIX Conf. on File and Storage Tech.* (2010), USENIX Association.

[11] Joukov, N., Krishnakumar, A. M., Patti, C., Rai, A., Satnur, S., Traeger, A., and Zadok, E. RAIF: Redundant Array of Independent Filesystems. In *Proc. of 24th IEEE Conf. on Mass Storage Systems and Tech.* (September 2007), IEEE, pp. 199–212.

[12] Krioukov, A., Bairavasundaram, L. N., Goodson, G. R., Srinivasan, K., Thelen, R., Arpaci-Dusseau, A. C., and Arpaci-Dussea, R. H. Parity lost and parity regained. In *Proc. of the Sixth USENIX Conf. on File and Storage Tech.* (2008), USENIX Association, pp. 1–15.

[13] Muniswamy-Reddy, K.-K., and Holland, D. A. Causality-based versioning. In *Proc. of the Seventh USENIX Conf. on File and Storage Tech.* (2009), pp. 15–28.

[14] Muniswamy-Reddy, K.-K., Wright, C. P., Himmer, A., and Zadok, E. A versatile and user-oriented versioning file system. In *Proc. of the Third USENIX Conf. on File and Storage Tech.* (2004), pp. 115–128.

[15] Peterson, Z., and Burns, R. Ext3cow: a time-shifting file system for regulatory compliance. *Trans. Storage 1* (May 2005), 190–212.

[16] Roselli, D., Lorch, J. R., and Anderson, T. E. A comparison of file system workloads. In *USENIX Ann. Tech Conf.* (2000), pp. 4–4.

[17] Santry, D. S., Feeley, M. J., Hutchinson, N. C., Veitch, A. C., Carton, R. W., and Ofir, J. Deciding when to forget in the elephant file system. In *Proc. of the 17th ACM Symp. on Oper. Syst. Prin.* (1999), pp. 110–123.

[18] Sidebotham, B. Volumes: the andrew file system data structuring primitive. Tech. Rep. CMU-ITC-053, 1986.

[19] Sivathanu, M., Bairavasundaram, L. N., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Life or death at block-level. In *Proc. of the Sixth Conf. on Symp. on Opearting Systems Design & Impl.* (2004), USENIX Association, pp. 26–26.

[20] StorageReview. http://www.storagereview.com/intel_x25v_ssd_review_40gb.

[21] Tanenbaum, A. S., and Woodhull, A. S. *Operating Systems Design and Impl. (Third Edition)*. Prentice Hall, 2006.

[22] Teigland, D., and Mauelshagen, H. Volume managers in linux. In *USENIX Ann. Tech. Conf., FREENIX Track* (2001), pp. 185–197.