FlexVol: Flexible, Efficient File Volume Virtualization in WAFL

John K. Edwards, Daniel Ellard, Craig Everhart, Robert Fair,
Eric Hamilton, Andy Kahn, Arkady Kanevsky, James Lentini,
Ashish Prakash, Keith A. Smith, Edward Zayas
NetApp, Inc.

Abstract

Virtualization is a well-known method of abstracting physical resources and of separating the manipulation and use of logical resources from their underlying implementation. We have used this technique to virtualize file volumes in the WAFL® file system, adding a level of indirection between client-visible volumes and the underlying physical storage. The resulting virtual file volumes, or *FlexVol*® volumes, are managed independent of lower storage layers. Multiple volumes can be dynamically created, deleted, resized, and reconfigured within the same physical storage container.

We also exploit this new virtualization layer to provide several powerful new capabilities. We have enhanced SnapMirror®, a tool for replicating volumes between storage systems, to remap storage allocation during transfer, thus optimizing disk layout for the destination storage system. *FlexClone*® volumes provide writable Snapshot® copies, using a FlexVol volume backed by a Snapshot copy of a different volume. FlexVol volumes also support thin provisioning; a FlexVol volume can have a logical size that exceeds the available physical storage. FlexClone volumes and thin provisioning are a powerful combination, as they allow the creation of light-weight copies of live data sets while consuming minimal storage resources.

We present the basic architecture of FlexVol volumes, including performance optimizations that decrease the overhead of our new virtualization layer. We also describe the new features enabled by this architecture. Our evaluation of FlexVol performance shows that it incurs only a minor performance degradation compared with traditional, nonvirtualized WAFL volumes. On the industry-standard SPEC SFS benchmark, FlexVol volumes exhibit less than 4% performance overhead, while providing all the benefits of virtualization.

1 Introduction

Conventional file systems, such as FFS [18], ext2 [3], or NTFS [9], are allocated on dedicated storage. Each file system, representing a single namespace tree, has exclusive ownership of one or more disks, partitions, and/or RAID groups, which provide the underlying persistent storage for the file system. The controlling file system is solely responsible for all decisions about the allocation and use of individual storage blocks on these devices.

With the continuing growth in disk capacities, this has become an increasingly inefficient way to manage physical storage. Larger storage capacities generate a commensurate pressure to create larger file systems on larger RAID groups. This optimizes for performance and efficient capacity utilization. A large number of spindles provides good performance; combining many disks into one large file system makes it easy to dynamically allocate or migrate storage capacity between the users or applications sharing the storage. But this growth in storage capacity can be challenging for end users and administrators, who often prefer to manage their data in logical units determined by the size and characteristics of their application datasets.

As a result, administrators have faced competing pressures in managing their storage systems: either create large file systems to optimize performance and utilization, or create smaller file systems to facilitate the independent management of different datasets. A number of systems, ranging from the Andrew file system [13, 23] to ZFS [26], have addressed these competing needs by allowing multiple file systems, or namespace trees, to share the same storage resources. By separating the management of file systems from the management of physical storage resources, these systems make it easier to create, destroy, and resize file systems, as these operations can be performed independent of the underlying storage.

NetApp® has followed a similar evolution with its WAFL file system [11]. For most of its history, users have allocated WAFL file systems (or *volumes* in NetApp terminology) on dedicated sets of disks configured as one or more RAID groups. As a result, WAFL presented the same management challenges as many other file systems. Customers who combined separate file sets on a single volume were forced to manage these files as

a single unit. For example, WAFL Snapshot copies operate at the volume level, so an administrator had to create a single Snapshot schedule sufficient to meet the needs of all applications and users sharing a volume.

In this paper we describe *flexible volumes*, a storage virtualization technology that NetApp introduced in 2004 in release 7.0 of Data ONTAP®. By implementing a level of indirection between physical storage containers (called *aggregates*) and logical volumes (*FlexVol volumes*), we virtualize the allocation of volumes on physical storage, allowing multiple, independently managed file volumes, along with their Snapshot copies, to share the same storage.

Adding a level of indirection allows administrators to manage datasets at the granularity of their choice. It also provides a mechanism for seamlessly introducing new functionality. We have used this indirection to implement writable Snapshot copies (called *FlexClone volumes*), thin provisioning, and efficient remote mirroring.

As is often the case, introducing a new level of indirection brings both benefits and challenges. Mapping between virtual block addresses used by FlexVols and physical block addresses used by aggregates can require extra disk I/Os. This could have a significant impact on I/O-intensive workloads. We have introduced two important optimizations that reduce this overhead. First, dual VBN mappings cache physical block numbers in the metadata for a FlexVol volume, eliminating the need to look up these mappings on most I/O requests. Second, lazily reclaiming freed space from a FlexVol volume dramatically reduces the I/O required to update the virtual-to-physical block mappings. With these optimizations we find that the performance of the SPEC SFS benchmark when using a FlexVol volume is within 4% of the performance of a traditional, nonvirtualized WAFL volume.

In the remainder of this paper, we first describe the design and implementation of flexible volumes, including an overview of the WAFL file system and a description of some of the new functionality enabled by FlexVol volumes. Next we describe our tests comparing FlexVol performance to that of traditional WAFL volumes. Finally, we survey other systems with similar goals and ideas and present our conclusions.

2 Background

This section provides a brief overview of the WAFL file system, the core component of NetApp's operating system, Data ONTAP. The original WAFL paper [11] provides a more complete overview of WAFL, although some details have changed since its publication. In this section, we focus on the aspects of WAFL that are most important for understanding the FlexVol architecture.

Throughout this paper we distinguish between file systems and volumes. A file system is the code and data structures that implement a persistent hierarchical namespace of files and directories. A volume is an instantiation of the file system. Administrators create and manage volumes; users store files on volumes. The WAFL file system implements these volumes.

WAFL uses many of the same basic data structures as traditional UNIX® style file systems such as FFS [18] or ext2 [3]. Each file is described by an *inode*, which contains per-file metadata and pointers to data or indirect blocks. For small files, the inode points directly to the data blocks. For large files, the inode points to trees of indirect blocks. In WAFL, we call the tree of indirect blocks for a file its *buftree*.

Unlike FFS and its relatives, WAFL's metadata is stored in various metadata files. All of the inodes in the file system are stored in the *inode file*, and the block allocation bitmap is stored in the *block map file*.

These data structures form a tree, rooted in the *vol_info* block. The vol_info block is analogous to the superblock of other file systems. It contains the inode describing the inode file, which in turn contains the inodes for all of the other files in the file system, including the other metadata files.

WAFL can find any piece of data or metadata by traversing the tree rooted at the vol_info block. As long as it can find the vol_info block, it doesn't matter where any of the other blocks are allocated on disk. This leads to the eponymous characteristic of WAFL—its *Write Anywhere File Layout*.

When writing a block to disk (data or metadata), WAFL never overwrites the current version of that block. Instead, the new value of each block is written to an unused location on disk. Thus, each time WAFL writes a block, it must also update any block that points to the old location of the block (which could be the vol_info block, an inode, or an indirect block). These updates recursively create a chain of block updates that reaches all the way up to the vol_info block.

If WAFL performed all of these updates for each data write, the extra I/O activity would be crippling. Instead, WAFL collects many block updates and writes them to disk *en masse*. This allows WAFL to allocate a large number of blocks to a single region in RAID, providing good write performance. In addition, many of the written blocks are typically referenced from the same indirect blocks, significantly reducing the cost of updating the metadata tree. Each of these write episodes completes when the vol_info block is updated, atomically advancing the on-disk file system state from the tree rooted at the old vol_info block to the tree rooted at the new one. For this reason, each of these write episodes is called a

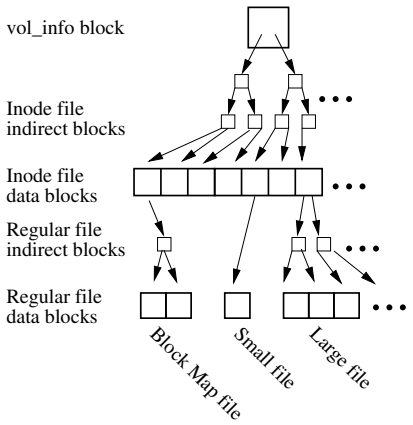

Figure 1: WAFL data structures

consistency point or *CP* for short.

WAFL uses non-volatile memory to log all incoming requests. This ensures that no data is lost in the event of a failure and allows WAFL to acknowledge write requests as soon as they are logged, rather than waiting until the next CP has completed and the data is on disk. After a failure, WAFL returns to the most recently committed CP and replays the contents of the NVRAM log, similar to error recovery in a journaling file system [22].

One of the benefits of the WAFL write anywhere allocation scheme is that it can create point-in-time Snapshot copies of a volume almost for free. Each CP results in a completely consistent on-disk file system image rooted at the current vol_info block. By preserving an old vol_info block and the tree rooted from it, we can create a Snapshot copy of the file system at that point in time. These Snapshot copies are space efficient. The only differences between a Snapshot copy and the live file system are the blocks that have been modified since the Snapshot copy was created (and the metadata that points to them). In essence, WAFL implements copy-on-write as a side effect of its normal operation.

3 FlexVol Architecture

Prior to the introduction of FlexVol volumes, Data ONTAP statically allocated WAFL volumes to one or more RAID groups. Each disk and each RAID group would belong exclusively to a single volume. We call this style of configuration, which is still supported in Data ONTAP today, a *traditional volume*, or a *TradVol*.

Our goal, in creating FlexVol volumes, was to break this tight bond between volumes and their underlying storage. Conceptually, we wanted to aggregate many disks into a large storage container and allow administrators to create volumes by carving out arbitrarily sized logical chunks of this storage.

Thinking about this problem, we realized the relationship between volumes and physical storage is the same as that between files and a volume. Since we had a perfectly good file system available in WAFL, we used it to implement FlexVol volumes. This is the essence of the architecture: a FlexVol volume is a file system created *within* a file on an underlying file system. A hidden file system spans a pool of storage, and we create externally visible volumes inside files on this file system. This introduces a level of indirection, or virtualization, between the logical storage space used by a volume and the physical storage space provided by the RAID subsystem.

3.1 Aggregates

An aggregate consists of one or more RAID groups. This storage space is organized as a simple file system structure that keeps track of and manages individual FlexVol volumes. It includes a bitmap indicating which blocks in the aggregate are allocated. The aggregate also contains a directory for each FlexVol volume. This directory serves as a repository for the FlexVol volume and associated data, and it provides a uniform repository for volume-related metadata. It contains two important files for each FlexVol volume, the *RAID file* and the *container file*.

The RAID file contains a variety of metadata describing the FlexVol volume, including its volume name, file system identifier, current volume state, volume size, and a small collection of other information. We call this the RAID file because the corresponding information for a TradVol is stored as part of the RAID label (along with RAID configuration information).

The container file contains all the blocks of the FlexVol volume. Thus, the block addresses used within a FlexVol volume refer to block offsets within its container file. In some respects, the container file serves as a virtual disk containing the FlexVol volume, with significant differences as described in later sections.

Since the container file contains every block within a FlexVol volume, there are two ways to refer to the location of a given block. The physical volume block number (PVBN) specifies the block's location within the aggregate. This address can be used to read or write the block to RAID. The virtual volume block number (VVBN) specifies the block's offset within the container file.

To better understand VVBNs and PVBNs, consider the process of finding the physical disk block given an offset into a file within a FlexVol volume. WAFL uses the file's buftree to translate the file offset to a VVBN—a block address within the FlexVol volume's virtual block address space. This is the same as the way a traditional file system would translate a file offset to a disk

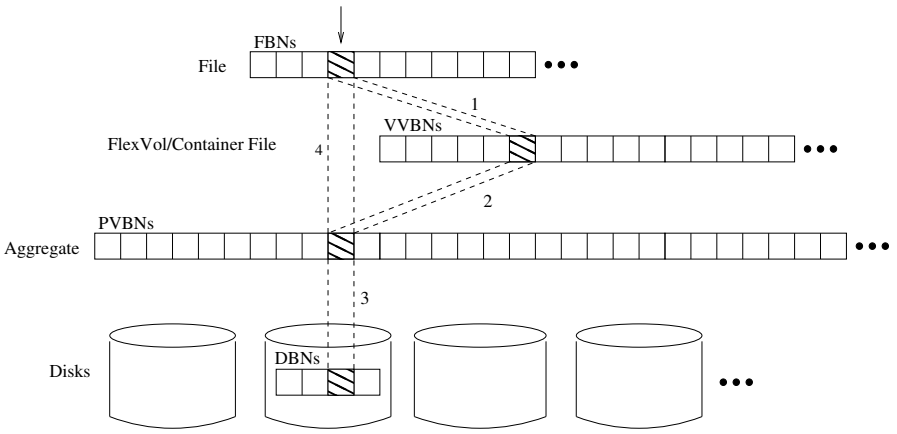

Figure 2: Mapping a block from file to disk. *We show a single block as part of several logical and physical storage containers—a file, a container file holding a FlexVol volume, an aggregate, and a disk. Each provides an array of blocks indexed by the appropriate type of block number. The file is indexed by file block number (FBN), the container file by VVBN, and the aggregate by PVBN. Finally, the disks are indexed by disk block number (DBN). To translate an FBN to a disk block, WAFL goes through several steps. In step 1, WAFL uses the file's inode and buftree to translate the FBN to a VVBN. In step 2, WAFL translates the VVBN to a PVBN using the container file's inode and buftree. Finally, in step 3, RAID translates the PVBN to a DBN. Step 4 shows the short-cut provided by dual VBNs (see Section 3.3). By storing PVBNs in the file's buftree, WAFL bypasses the container map's VVBN-to-PVBN translation.*

address. The FlexVol volume's block address space is defined by the container file. WAFL uses the container file's buftree to translate the VVBN to a block address in the aggregate's block address space. This provides a PVBN, which WAFL can give to the RAID subsystem to store or retrieve the block. Figure 2 displays this mapping process and the relationship between the different types of block numbers.

Observe that in the container file's buftree, the first level of indirect blocks list all of the PVBNs for the container file. Together, these blocks form an array of PVBNs indexed by VVBN. We refer to the VVBN-to-PVBN mapping provided by this first level of indirect data in the container file as the *container map*.

3.2 Volumes

Because FlexVol volumes are implemented by container files, they inherit many characteristics of regular files. This provides management flexibility, which we can expose to users and administrators.

When a FlexVol volume is created, its container file is sparsely populated; most of the logical offsets have no underlying physical storage. WAFL allocates physical storage to the container file as the FlexVol volume writes data to new logical offsets. This occurs in much the same way as hole-filling of a sparse file in a conventional file system. This sparse allocation of container files also allows the implementation of thin provisioning, as described in Section 4.3.

The contents of a FlexVol volume are similar to those of a traditional WAFL volume. As with a TradVol, there is a vol_info block, located at well-known locations within the container file space. Within the volume are all of the standard WAFL metadata files found in a TradVol.

A FlexVol volume contains the same block allocation files as a traditional volume. While the aggregate-level versions of these files are indexed by PVBN, in a FlexVol volume these files are indexed by VVBN. Thus, the files and the associated processing scale with the logical size of the volume, not with the physical size of the aggregate.

3.3 Dual Block Numbers

The use of VVBNs introduces a layer of indirection which, if not addressed, may have significant impact on read latencies. In a naïve implementation, translating a file offset to a physical block (PVBN) would require WAFL to read two buftrees—one for the file to find the VVBN and one for the container file to find the PVBN. In the worst case, this overhead could be quite high, as WAFL might have to perform this translation for each indirect block as it traverses the file's buftree.

To address this problem, FlexVol volumes use *dual VBNs*. Each block pointer in a FlexVol volume contains two block addresses, the VVBN and its PVBN translation. For normal read operations, WAFL never needs to look up PVBNs in the container file buftree. It just uses the PVBN values it finds in the dual VBNs stored in the inodes and indirect blocks of the FlexVol volume. Figure 3 illustrates this process.

For writes, WAFL allocates a VVBN from the FlexVol volume's container file and a PVBN from its ag-

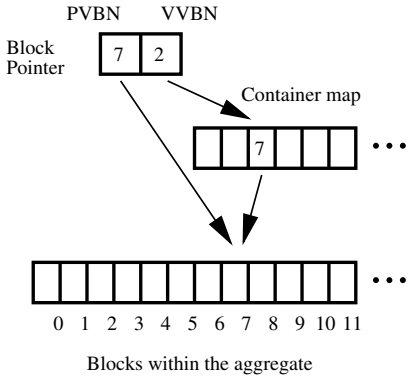

Figure 3: Dual volume block numbers in indirect blocks. *A block pointer within a FlexVol volume contains two block addresses. One is the FlexVol volume's VVBN, the location of the block within the flexible volume's container file. The other is the aggregate's PVBN, the physical location of the block. In this example the read path bypasses the container map and goes directly to PVBN #7. The VVBN of 2 is the index into the container map that can be used as an alternate means to find the PVBN.*

gregate, and it updates the file and container file buftrees accordingly. This requires extra processing and I/O. Because WAFL delays write allocation until it writes a consistency point, this work occurs asynchronously and does not add latency to the original write request. In workloads with good locality of reference, the overhead is further reduced by amortizing it across multiple updates to the same buftrees.

3.4 Delayed Block Freeing

When a block is freed—for example, by a file deletion—we would like to mark it as free in the aggregate as well as in the FlexVol volume. In other words, we want FlexVol volumes to return their free space to the underlying aggregate. Thus, when a block is no longer referenced from the active file system or from any Snapshot copies we not only mark it free in the FlexVol volume's block map, we also free the block in the aggregate's block map and mark the corresponding location in the container file as unallocated, effectively "punching a hole" in the container file.

The fact that unused blocks are eventually returned from the FlexVol volume to the aggregate is a key feature of the FlexVol design. Without this functionality, freed blocks would remain allocated to their FlexVol. The aggregate would not know that these blocks were free and would not be able to allocate them to other volumes, resulting in artificial free space fragmentation. In contrast, in our implementation, free space is held by the aggregate, not the FlexVol volume; the free space in the aggregate can be made available to any volume within the aggregate. Most importantly, the unrestricted flow of free space requires no external intervention and no management.

This mechanism also motivates an important performance optimization for freeing blocks. Since WAFL always writes modified data to new locations on disk, random overwrites on a large file tend to produce random frees within the VVBN space of the FlexVol volume. This results in random updates of the container file's indirect blocks, adding an unacceptable overhead to random updates of large files.

WAFL avoids this problem by delaying frees from the container file in order to batch updates. WAFL maintains a count of the number of *delayed free* blocks on each page of the container map. Up to two percent of a FlexVol volume's VVBN space can be in the delayed free state. Once the number of delayed free blocks crosses a one percent threshold, newly generated delayed frees trigger background cleaning. The cleaning of the container file is focused on regions of the container block file that have larger than average concentrations of delayed free blocks. Since an indirect block in the container file buftree has 1,024 entries, an average indirect block of the container will hold at least ten (1% of 1,024) delayed frees, and often significantly more. This reduces the container file overhead of freeing blocks to less than one update per ten frees.

4 New Features

Adding a level of indirection between the block addressing used by a FlexVol volume and the physical block addressing used by the underlying RAID system creates a leverage point that we have used to introduce new functionality and optimizations. In this section, we describe three such improvements—volume mirroring for remote replication, volume cloning, and thin provisioning.

4.1 Volume Mirroring

Volume SnapMirror[20] is a replication technology that mirrors a volume from one system to another. SnapMirror examines and compares block allocation bitmaps from different Snapshot copies on the source volume to determine the set of blocks that it must transfer to transmit the corresponding Snapshot copy to the remote volume. This allows SnapMirror to efficiently update the remote volume by advancing its state from Snapshot copy to Snapshot copy.

Since the decisions of which blocks to transfer and where to place them at the destination are based on the allocation maps, they are based on the type of VBN that serves as the index to those files. In a traditional vol-

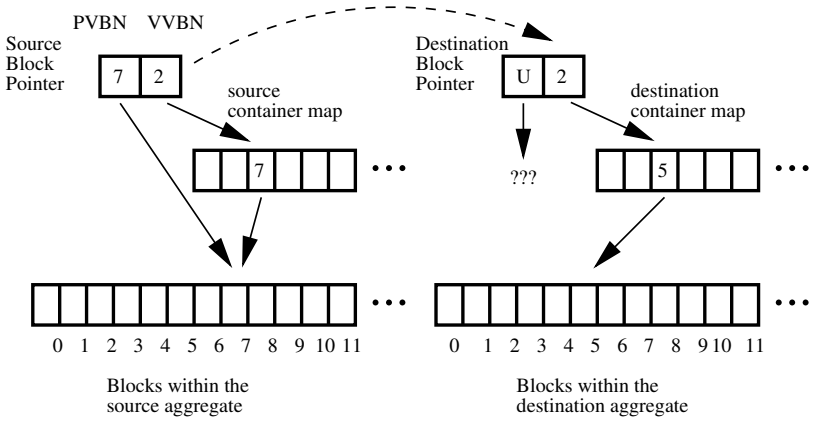

Figure 4: SnapMirror Transfer. *SnapMirror transfers blocks from a source volume to a destination volume based on VVBNs. The transfers are independent of the physical block numbers involved. Here we show how SnapMirror updates a block pointer as a block is transferred. On the source, the block has a VVBN of 2 and a PVBN of 7. When the block is transferred, the destination system assigns it a new PVBN (5) and enters that it in the destination FlexVol volume's container map. When it copies the block pointer, SnapMirror preserves the VVBN; the block has the same logical address within both FlexVols. The PVBN of the destination block pointer is set to PVBN-UNKNOWN (indicated by a 'U' above). A background process will eventually replace this with the correct PVBN. If WAFL needs to resolve the block pointer before the PVBN is filled in, it can look up the PVBN using the container map.*

ume, file block pointers are PVBNs. Thus, to transfer a block, WAFL reads it from RAID using the PVBN and transfers it to the destination where it is written to the same PVBN. If the physical geometries of the source and destination differ, WAFL cannot optimize the I/O for the transfer on both the source and the destination. This is particularly problematic when transferring blocks between systems with drives that have different sizes and geometries, such as when transferring between smaller, faster primary storage disks and larger, slower secondary storage disks.

In contrast, flexible volume transfers are VVBN-based. WAFL uses the FlexVol volume's block allocation files to determine which VVBNs to transfer, and it transfers those blocks from the container file on the source system to the container file at the destination. The destination system assigns a new PVBN to each block while maintaining the same VVBN as on the source system. This removes geometry restrictions from Volume SnapMirror because the source and destination make physical allocation decisions independently. As a result, volumes can be mirrored between aggregates with different sizes and/or disk configurations.

Changing the PVBNs of the volume across a transfer introduces a difficulty. Among the blocks transferred are metadata blocks that contain block pointers, particularly inode file blocks and indirect blocks. Since the VVBN-to-PVBN mapping at the destination is different from that at the source, all of the PVBNs in block pointers must be changed as part of the transfer. To allow SnapMirror to locate the block pointers within blocks, WAFL maintains a block type file that identifies the general function of each block within the volume, indexed by VVBN. As part of the transfer itself, PVBNs are replaced with a special reserved value, *PVBN-UNKNOWN*. The destination must then replace the PVBN-UNKNOWNs with actual PVBNs. Figure 4 illustrates the process of mirroring a single block from one FlexVol volume to another.

Even in the presence of PVBN-UNKNOWN, access and use of the destination volume are possible. Any code encountering an unknown PVBN while attempting to read data can instead use the VVBN and the container map to find the PVBN for the required block. This allows access to transferred data while the PVBNs are still being repaired via a background process.

4.2 Volume Cloning

WAFL Snapshot copies provide consistent point-in-time copies of a volume. This has many uses, but sometimes the read-only nature of a Snapshot copy is a limitation. In database environments, for example, it is often desirable to make writable copies of a production database for development or test purposes. Other uses for writable Snapshots copies include upgrades or modifications to large applications and provisioning many systems in a grid or virtual machine environment from a single master disk image.

Volume cloning creates a FlexVol volume in which the active file system is a logical replica of a Snapshot copy in a different FlexVol volume within the same aggregate. The parent volume can be any FlexVol volume, including a read-only mirror. Like the creation of a snap-

shot, creating a flexible volume clone, or *FlexClone volume*, requires the writing of a fixed, small number of blocks and is nearly instantaneous. The FlexClone volume is a full-fledged FlexVol volume with all the features and capabilities of a normal WAFL volume.

Creating a clone volume is a simple process. WAFL creates the files required for a new FlexVol volume. But, rather than creating and writing a new file system inside the volume, WAFL seeds the container file of the clone with a vol_info block that is a copy of the vol_info block of the Snapshot copy on which the clone is based. Since that vol_info is the top-level block in the tree of blocks that form the Snapshot copy, the clone inherits pointers to the complete file system image stored in the original Snapshot copy. Since the clone does not actually own the blocks it has inherited from the parent, it does not have those blocks in its container file. Rather the clone's container has holes, represented by zeros in the container map, indicating that the block at that VVBN is inherited from the parent volume (or some more distant ancestor, if the parent volume is also a clone).

To protect the cloned blocks from being freed and overwritten, the system needs to ensure that the original volume will not free the blocks used by the clone. The system records that the clone volume is relying on the Snapshot copy in the parent volume and prevents the deletion of the Snapshot copy in the parent volume or the destruction of the parent volume. Similarly, WAFL ensures that the clone will not free blocks in the aggregate that are owned by the parent volume. Inherited blocks are easily identified at the time they are freed because the container file of the clone has a hole at that VVBN.

A clone volume can be split from the parent volume. To do this a background thread creates new copies of any blocks that are shared with the parent. As a result of this process, the clone and parent will no longer share any blocks, severing the connection between them.

4.3 Thin Provisioning

As described earlier, the free space within a WAFL aggregate is held by the aggregate. Since FlexVol volumes do not consume physical space for unallocated blocks in their address space, it is natural to consider thin provisioning of volumes. For example, several 1TB volumes can be contained in a 1TB aggregate if the total physical space used by the volumes is less than 1TB. The ability to present sparsely filled volumes of requested sizes without committing underlying physical storage is a powerful planning tool for administrators. Volume clones also present a natural case for thin provisioning, since users will often want many clones of a given volume, but the administrator knows that the clones will all share most of their blocks with the base snapshot.

While thin provisioning of volumes presents many opportunities to administrators, it also provides challenges. An aggregate can contain many volumes, and no single provisioning policy will suit all of them, so WAFL allows different volumes in the same aggregate to use different policies. The policies are *volume*, *none*, and *file*, which provide a range of options for managing thin provisioning. The three policies differ in their treatment of volume free space and their treatment of space-reserved objects within volumes.

The *volume* policy is equivalent to a space reservation at the aggregate level. It ensures that no other volumes can encroach on the free space of the volume. The *none* policy implements thin provisioning for the entire volume. No space is reserved for the aggregate beyond that currently consumed by its allocated blocks.

The third policy, *file*, exists for cases where writes to specific files should not fail due to lack of space. This typically occurs when clients access a file using a block protocol such as iSCSI. In such a case, the file appears to the client as logical disk device. For such objects, WAFL provides the ability to reserve space for the underlying files. On a volume with a *file* policy, the reservations on individual objects within the volume are honored at the aggregate level. Thus, if a 400GB database table is created on a 1TB FlexVol volume with a *file* policy, the aggregate would reserve 400GB of space to back the database file, but the remaining 600GB would be thinly provisioned, with no storage reservation.

The default policy for a FlexVol volume is *volume*, since this means that volumes behave precisely the same as a fully provisioned TradVol. By default, a FlexClone volume inherits the storage policy of its parent volume.

4.4 Other Enhancements

Over time we have continued to find this new level of virtualization valuable for cleanly implementing new features in WAFL. In addition to the features described above, we are also using this technique to introduce block-level storage deduplication and background defragmentation of files and free space.

5 Evaluation

In this section, we evaluate the performance overhead imposed by the FlexVol architecture. While FlexVol virtualization is not free, we find that the extra overhead it imposes is quite modest. We compare the performance of FlexVol volumes and TradVol volumes using both microbenchmarks and a large-scale workload. We also discuss how customers use FlexVol volumes in production

environments, presenting data drawn from NetApp's installed base. Finally, we comment on our practical experiences with the engineering challenges of introducing FlexVol volumes into Data ONTAP.

5.1 FlexVol Overhead

We compare the performance of FlexVol volumes and TradVol volumes for basic file serving operations and for larger workloads. Our goal is to quantify and understand the overheads imposed by the extra level of indirection that allows us to implement the features of FlexVol volumes.

Intuitively, we expect FlexVol volumes to impose overhead due to the increased metadata footprint required to maintain two levels of buftree. Larger metadata working sets will increase cache pressure, I/O load, and disk utilization. In addition, generating and traversing this increased metadata will add CPU overhead.

To determine where these overheads occur and quantify them, we ran a set of simple micro-benchmarks on both FlexVol volumes and TradVol volumes. By examining the performance differences between the two volume types in conjunction with performance counter data from Data ONTAP, we can measure the specific performance cost of FlexVol volumes.

Finally, to understand how these detailed performance differences affect large-scale benchmarks, we used the industry-standard SPEC SFS benchmark [24] to compare the macro-level performance of both volume types.

5.1.1 Microbenchmarks

For our microbenchmarks, we used an FAS980 server running Data ONTAP version 7.2.2. This system has two 2.8GHz Intel® Xeon processors, 8GB of RAM, and 512MB of NVRAM. The storage consists of 28 disk drives (72GB Seagate Cheetah 10K RPM) configured as two 11 disk RAID-DP® [7] aggregates. One aggregate was used for FlexVol volumes and the other as a TradVol volume. The remaining disks were not used in these tests; they held the root volume or served as spares.

The benchmark we used is called *filersio*. It is a simple tool that runs on a file server as part of Data ONTAP. We used it to generate read and write workloads with either sequential or random access patterns. Because filersio runs on the file server, it issues requests directly to WAFL, bypassing the network and protocol stacks. This allows us to focus our investigation on the file system, ignoring the interactions of client caches and protocol implementations.

We used filersio to examine four workloads—random reads, sequential reads, random writes, and sequential writes. To examine the impact of FlexVols volumes' larger metadata foot print, we ran each test on increasing dataset sizes, ranging from 512MB to 32GB. At 512MB, all of the test data and metadata fits in the buffer cache or our test system. With a 32GB data set, the buffer cache holds less than 20% of the test data. In each test, our data set was a single large file of the appropriate size.

For each test, we ran the workload for 5 minutes. We configured filersio to maintain ten outstanding I/O requests. We warm the WAFL buffer cache for these tests by first creating the test file and then performing five minutes of the test workload before we execute the measured test. The I/O request size and alignment for all tests is 4KB, matching the underlying WAFL block size. All of the results are averages of multiple runs, each using a different test file.

Figures 5, 6, 7, and 8 show the results of our microbenchmarks. We note that across most of these benchmarks the FlexVol volume has performance nearly identical to the TradVol. There are three significant areas where FlexVol performance lags—for file sizes less than about 6GB in the random read tests, for file sizes greater than 8GB in the sequential write test, and for all file sizes in the random write test. The largest performance differences were for 32GB random writes, where the FlexVol performed 14% worse than the TradVol. In the remainder of this section, we discuss these differences and also provide some intuition about the general shapes of the performance curves we see in these microbenchmarks.

Figure 5 shows the results of the random read benchmark. This graph can be divided into two regions. For file sizes larger than about 5.8GB, the working set size exceeds the available cache. Performance drops off rapidly in this region, as more and more requests require disk I/O. It is in this region that we might expect to see the performance overhead of the FlexVol volume, as it needs to access additional metadata, both on disk and in memory, to satisfy cache misses. The FlexVol requires twice as many indirect blocks to describe a file of a given size. These extra blocks compete with data blocks for cache space and reduce our hit rate. In fact, the performance of the two configurations is nearly identical, as the increase in cache pressure is negligible. Roughly 1 in 500 cache blocks is used for indirect metadata on a FlexVol volume, compared to 1 in 1000 on a TradVol.

On the left-hand side of Figure 5 our working set fits entirely in the buffer cache. Cache hits in WAFL follow the same code path, regardless of the volume type, and do not read any buftree metadata. Thus, we would expect all tests in this range to perform identically, regardless of file size or volume type. Surprisingly, this was not the case. The performance of both volume types steadily declines as the file size increases, and FlexVol volumes consistently underperform TradVols.

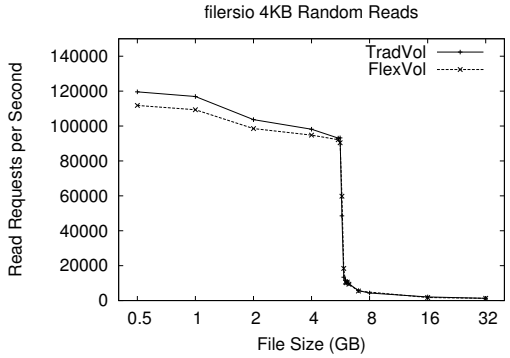

Figure 5: The performance of a random read workload on files of increasing size. *Each data point is the average of eight test runs. Standard deviations are less than 5% of the average on all data points except where the working set starts to exceed the size of the buffer cache (5.7–6.4GB). At these sizes standard deviations ranged from 5–30%, reflecting the sensitivity of these tests. With a 100x performance difference between cache hits and cache misses, small differences in cache hit rates, caused by the random number generator, result in large differences in average performance.*

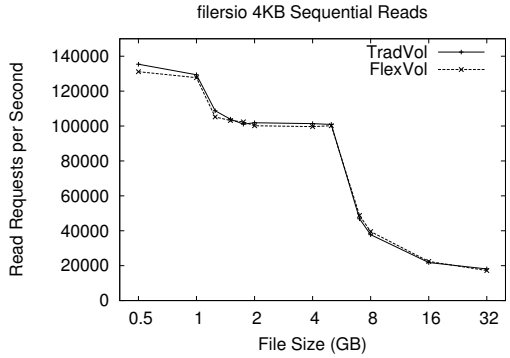

Figure 6: The performance of a sequential read workload on files of increasing size. *Each data point is the average of eight runs. Standard deviations are less than 9% of the average at all data points.*

To better understand this behavior, we examined the performance statistics Data ONTAP collects. As expected, these tests had 100% hit rates in the buffer cache. Further study of these statistics uncovered the explanations for these behaviors. First, because we have 8GB of RAM on a system with a 32-bit architecture, not all cache hits have the same cost. When WAFL finds a page in the cache that is not mapped into its address space, Data ONTAP must remap the page before returning a data pointer to the page. This accounts for the declining performance as the file size increases. With larger working set sizes, a greater percentage of blocks are cached but not in Data ONTAP's address space, and thus a greater percentage of cache hits pay the page remapping penalty. In the tests with 512MB files, the total number of page remappings is less than 0.001% of the number of cache hits. For the 5.5GB tests, this ratio is 75%.

The second anomaly—the fact that TradVol volumes appear to serve cache hits faster than FlexVol volumes—is explained by a cached read optimization. When WAFL detects a sequence of 5,000 or more successive reads that are serviced from the cache, it stops issuing read-ahead requests until there is a cache miss. Unexpectedly, this optimization is not always enabled during these filersio runs. The culprit is various *scanner* threads that WAFL runs in the background. These threads perform a variety of tasks, such as the delayed-free processing described in Section 3.4. Whenever one of these scanners has a read miss, it disables the cached read optimization on its volume. There is more of this background activity on a FlexVol volume than a TradVol. As a result, the cached read optimization is not enabled as much on the FlexVol volume, causing it to spend extra CPU cycles needlessly looking for blocks to prefetch. Since this workload is completely CPU-bound, this extra overhead has a noticeable effect on performance. In the tests with 512MB files, the cached read optimization was enabled for 82% of the reads from the TradVol, but only for 32% of the reads from the FlexVol volume. Unfortunately, it is not possible to completely disable this background processing in WAFL. It generally causes very little additional I/O, and this was the only test case where it had a noticeable effect on our test results.

Figure 6 shows sequential read performance. The performance on the FlexVol volume is, again, nearly identical to TradVol performance. The overall shape of the performance curves is similar to the random read benchmark with two noteworthy differences. First, the out-of-cache performance is substantially better than the random read case, reflecting the performance gains from prefetching and sequential I/O. The second difference is the step-like performance drop around the 1.5GB file size. This is where the file size exceeds the available mapped buffer cache. For file sizes smaller than this we see almost no page remaps, but for file sizes larger than this the number of page remaps roughly equals the total number of reads during the test.

Figure 7 shows sequential write performance. Here we see similar performance for FlexVol volumes and TradVol volumes at most file sizes. Although FlexVol volumes update more metadata during this benchmark, it has no discernible effect on performance for most file sizes. The sequential nature of the workload provides good locality of reference in the metadata the FlexVol

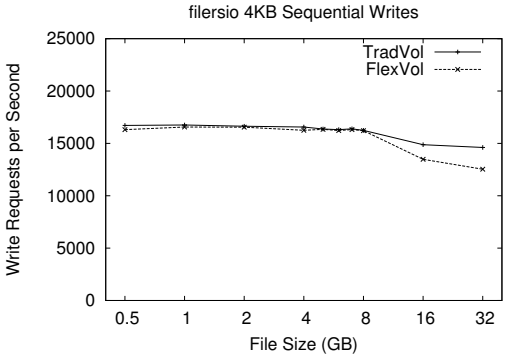

Figure 7: The performance of a sequential write workload on files of increasing size. *Each data point is the average of ten runs. Standard deviations are less than 5% of the average at all data points.*

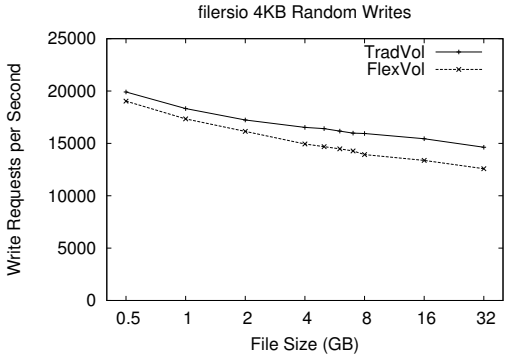

Figure 8: The performance of a random write workload on files of increasing size. *Each data point is the average of eight test runs. Standard deviations are less than 8% of the average on all data points.*

volume needs to modify. Thus, the amount of metadata of written to the FlexVol volume is quite modest compared to the file data writes, which dominate this test.

As the file sizes grow past 8GB, sequential write performance on both volumes started to drop, with a larger performance decrease on the FlexVol volume. Our performance statistics are less clear about the cause of this drop. It appears to stem from an increase in the number of cache misses for metadata pages, particularly the allocation bitmaps. This has a larger impact on the FlexVol volume because it has twice as much bitmap information—both a volume-level bitmap and an aggregate-level bitmap.

Finally, in Figure 8 we show random write performance. On both volumes, performance slowly decreased as we increased the file size. This occurs because we have to write more data to disk for larger file sizes. For smaller file sizes, a greater percentage of the random writes in each consistency point are overwrites of a previously written block, reducing the number of I/Os we have to perform in that consistency point. For example, the 512MB FlexVol test wrote an average of 45,570 distinct data blocks per CP; the 32GB FlexVol test wrote an average of 72,514 distinct data blocks per CP.

In the random write test we also see that FlexVol performance lagged TradVol performance. The performance difference ranges from a few percent at the smaller file sizes to 14% for 32GB files. This performance gap reflects several types of extra work that the FlexVol volume performs, most notably the extra I/O associated with updating both the file and container buftrees and the delayed free activity associated with all of the blocks that are overwritten during this test. These same factors occur in the sequential write test, but their performance impact is attenuated by the locality of reference we get in the metadata from doing sequential writes. With random writes, we change much more metadata in each CP, increasing the amount of data that have we to write to disk. The slightly larger performance gap for large file sizes is due to the cache misses loading bitmap files, as we saw in the sequential write test described above.

In summary, most of our microbenchmarks show nearly identical FlexVol and TradVol performance. The major exceptions are cached random reads, random writes, and sequential writes to large files. In these cases, FlexVol performance is often within a few percent of TradVol performance; in the worst cases the performance difference is as much as 14%.

5.1.2 SFS Benchmark

To understand how the behaviors observed in our microbenchmarks combine to affect the performance of a more realistic workload, we now examine the behavior of a large scale benchmark on both FlexVol volumes and TradVol volumes.

For this test, we use the SPEC Server File System (SFS) benchmark [24]. SFS is an industry standard benchmark for evaluating NFS file server performance. SFS originated 1993 when SPEC adopted the LADDIS benchmark [28] as a standard benchmark for NFS servers. We used version SFSv3.0r1, which was released in 2001.

SFS uses multiple clients to generate a stochastic mix of NFS operations from a predefined distribution. The load generating clients attempt to maintain a fixed load level, for example 5,000 operations/second. SFS records the actual performance of the server under test, which may not be the target load level. SFS scales the total data set size and the working set size with the offered load. A complete SFS run consists of multiple runs with

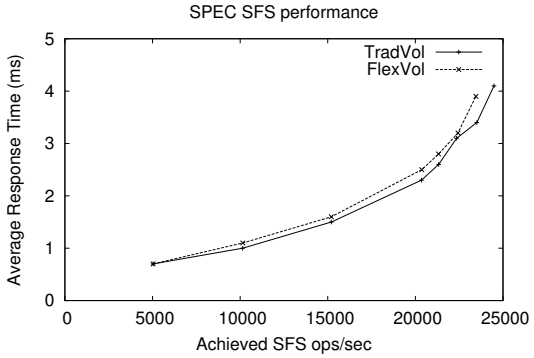

Figure 9: Performance of the SPEC SFS benchmark. *This graph plots average response time as a function of the achieved throughput (in operations per second). We performed three SFS runs in each configuration. Here we show the runs with median peak performance. There was little variance across the runs. Throughputs for all load points are within 1.5% of the values shown here. Response times are were within 10% of the values shown here, except for the two highest FlexVol load points, where they varied by as much as 18%.*

increasing offered load, until the server's performance peaks. At each load point, SFS runs a five minute load to warm the server cache, then a five minute load for measurement. The result from each load point is the achieved performance in operations per second and the average latency per request. Further details about SFS are available from the SPEC web site [24].

Our SFS tests used NFSv3 over TCP/IP. The file server was a single FAS 3050 (two 2.8GHz P4 Xeon processors, 4GB RAM, 512MB NVRAM) with 84 disk drives (72 GB Seagate Cheetah 15K RPM). The disks were configured as a single volume (either flexible or traditional) spanning five RAID-DP [7] groups, each with 14 data disks and 2 parity disks.

Figure 9 shows the results of these tests. As we would expect, given the relatively small performance differences uncovered in our microbenchmarks, the FlexVol and TradVol volumes have similar performance. The TradVol achieves a peak performance of 24,487 ops/sec, 4.4% better than the peak FlexVol performance of 23,452 ops/sec. With increasing loads, there is an increasing gap in response time, with FlexVol volumes showing higher latencies than TradVols. At the peak FlexVol throughput, its average response time of 3.9ms is 15% longer than the corresponding TradVol load point.

Our microbenchmarks exhibited several sources of FlexVol overhead that affect SFS performance, including extra CPU time and I/Os to process and update FlexVol metadata. Performance statistics collected from the file server during the SFS runs show that at comparable load points the FlexVol volume used 3–5% more CPU than the TradVol, read 4–8% more disk blocks, and wrote 5–10% more disk blocks. A large amount of the extra load came from medium size files (32–64KB) that don't need any indirect blocks on the TradVol but use indirect blocks on the FlexVol volume because of the extra space used by dual VBNs.

Overall, we are pleased with the modest overhead imposed by FlexVol virtualization. FlexVol performance is seldom more than a few of percent worse than TradVol performance. In comparison, this overhead is far less than the performance increase seen with each new generation of server hardware. In exchange for this slight performance penalty, customers have increased flexibility in how they manage, provision, and use their data.

5.2 Customer Usage of FlexVol Volumes

NetApp storage systems have a built-in, low-overhead facility for reporting important system events and configuration data back to the NetApp AutoSupport database. The use of this reporting tool is optional, but a large percentage of NetApp customers enable it. Previous studies have used this data to analyze storage subsystem failures [14], latent sector errors [1], and data corruptions [2]. In this section we examine system configuration data to understand how our customers use FlexVol volumes.

We examined customer configuration data over a span of a year from September 2006 to August 2007. At the beginning of this period, our database had information about 38,800 systems with a total of 117.8PB of storage. At the end of this period, we had data from 50,800 systems with 320.3PB of storage. Examining this data, we find evidence that customers have embraced FlexVol technology and rely on it heavily. Over this year, we found that the percentage of deployed systems that use only TradVols decreased from 46% of all systems to less than 30% of all systems. At the same time, the percentage of systems that use only FlexVol volumes increased from 42% to almost 60%. The remainder, systems that had a mixture of TradVol and FlexVol volumes, was fairly stable at 11.5% of the systems in our dataset.

Looking at numbers of volumes instead of systems we found that over the same year the number of FlexVol volumes increased from 69% of all volumes to 84%, while traditional volumes decreased from 31% to 16%.

FlexVol volumes allow customers to manage their data (volumes) at a different granularity than their storage (aggregates). As disk capacities have grown, we see increasing evidence that our customers are using FlexVol volumes in this way—allocating multiple FlexVol volumes on a single aggregate. During the year, the average size of the aggregates in our data set increased from 1.5TB to 2.3TB. At the same time, the average number

of volumes per aggregate increased from 1.36 to 1.96. Excluding traditional volumes, which occupy the whole aggregate, the average number of volumes per aggregate increased from 1.95 to 3.40.

Finally, we explored customer adoption of thinly provisioned FlexVol and FlexClone volumes. Over the past year, the number of thinly provisioned volumes (i.e., with a provisioning policy of *none*) has grown by 128%, representing 6.0% of all FlexVol volumes in the most recent data. The total size of all thin provisioned volumes currently exceeds the physical space available to them by 44%, up from 35% a year ago.

FlexClone volumes represent a smaller fraction of volumes, but their use has been growing rapidly, from 0.05% to 1.0% of all FlexVol volumes during our year of data. Customers are making heavy use of thin provisioning with their clone volumes. In the current data, 82% of all clones are thinly provisioned. We expect that this data under-represents customer adoption of FlexClone volumes, since many use cases involve short-lived clones created for testing and development purposes.

5.3 Experience

While FlexVol volumes provide good performance and have been readily adopted by NetApp customers, the experience of introducing a major piece of new functionality into an existing operating system has not been perfect. The majority of the challenges we faced have come from legacy parts of Data ONTAP, which assumed there would be a modest number of volumes.

One of the major areas where these problems have manifested is in limits on the total number of volumes allowed on a single system. At boot time and at failover, Data ONTAP serially mounts every volume on a system before accepting client requests. With many hundreds or thousands of volumes, this can have an adverse effect on system availability. Likewise, when Data ONTAP supported only a small number of volumes, there was little pressure to limit the memory footprint of per-volume data structures, and no provision was made to swap out volume-related metadata for inactive volumes.

Over time, these constraints have gradually been addressed. Data ONTAP currently restricts the total number of volumes on a single system to 500. This limits failover and reboot times and prevents volume metadata from consuming too much RAM.

6 Related Work

FlexVol is not the first system to allow multiple logical storage containers to share the same physical storage pool. In this section we survey other systems that have provided similar functionality. We first discuss other systems that provide virtual file systems, contrasting them with FlexVol volumes in terms of implementation and functionality. In the remainder of this section we contrast FlexVol volumes with various systems that provide other virtualized storage abstractions.

The earliest example of a file system supporting multiple file systems in the same storage was the Andrew File System (AFS) [12, 13, 15]. In AFS, each separate file system was called a volume [23]. AFS is a client-server system; on a file server, many volumes are housed on a single disk partition. Clients address each volume independently. Volumes grow or shrink depending on how much data is stored in them, growing by allocating free space within the disk region and shrinking by freeing space for use by others. An administrative limit (or *quota*) on this growth can be set independent of the disk region size. Thus, administrators can implement thin provisioning by overcommitting the free space in the region. AFS maintains a read-only copy-on-write point-in-time image of an active volume, called a *clone* in AFS terminology; the clone shares some of the storage with the active volume, similar to WAFL Snapshots copies. In AFS, clones are always read-only; the writable clones we describe do not exist in AFS.

Howard *et al.* [13] describe an evolution of AFS similar to that of WAFL and FlexVol volumes—moving from a prototype implementation that supported a single file volume per storage device (essentially unmodified 4.2BSD) to a revised system with the multiple volume per container architecture described above. Both the revised AFS implementation as well as FlexVol volumes implement virtualization by providing a layer of indirection: in the AFS case, per-volume inode tables, and in the FlexVols case, per-volume block maps.

Other file systems have since used an architecture similar to volumes in AFS. Coda [21] is a direct descendant of AFS that supports disconnected and mobile clients. DCE/DFS [6, 16], like WAFL, can also accommodate growth of the underlying disk media, for instance by adding a disk to a logical volume manager, allowing thin provisioning of volumes (called *filesets* in DCE/DFS) since space can be added to a storage region. Both of these systems can also maintain read-only copy-on-write clones as images of their active file systems.

DEC's AdvFS [10], available with the Tru64 operating system [5], provides a storage pooling concept atop which separate file systems (filesets) are allocated. It allows not only for media growth, and thus complete thin provisioning, but also media shrinkage as well. Disks can be added to or deleted from an AdvFS storage pool. It, too, provides for read-only copy-on-write images of filesets, also called clones.

Sun™ ZFS is the file system with the closest match

to FlexVol functionality. ZFS provides multiple directory trees (file systems) in a storage pool and supports both read-only snapshots and read-write clones [26]. ZFS does not use a two-level block addressing scheme in the manner of FlexVol volumes. Instead, all file systems, snapshots, and clones in a ZFS storage pool use storage pool address, the equivalent of aggregate-level PVBNs [25]. Thus, ZFS avoids some of the overheads in WAFL, such as the need to store dual VBN mappings and to perform block allocation in both the container file and the aggregate. As we have shown, these overheads are quite modest in WAFL, and the resulting indirection facilitates the introduction of new functionality.

IBM's Storage Tank is an example of another class of file system. It stores file data and file system metadata on separate, shared storage devices [19]. Treating the data as a whole, Storage Tank implements multiple file system images ("containers") that can grow or shrink, and active and snapshot file systems can share pointers to storage blocks. Read-write clones are not provided.

The basic FlexVol idea of virtualizing file systems by creating a file system inside a file can be implemented using standard tools on commodity operating systems. Both the BSD and Linux® operating systems support the creation of a block device backed by a file. By formatting these devices as subsidiary file systems, an administrator can achieve a result similar to FlexVol volumes. This mode of operation would support thin provisioning by using a sparse backing file. It would not, however, provide many of the other optimizations and enhancements available with FlexVols—dual VBNs, storage deallocation by hole punching, free behind, clone volumes, etc.

There are many block-oriented storage systems that provide virtualization functionality similar to FlexVol volumes. Logical volume managers (LVMs) such as Veritas™ Volume Manager [27] and LVM2 in Linux [17] allow the dynamic allocation of logical disk volumes from pools of physical disk resources. Some volume managers also include support for read-only snapshots and read-write clones. Many mid-range and high-end disk arrays [4, 8] provide similar features, essentially implementing volume management internally to the device and exporting the resulting logical disk volumes to hosts via block protocols such as FCP or iSCSI.

Since volume managers and disk arrays provide block-oriented storage virtualization, their implementation differs substantially from file-based virtualization such as FlexVol volumes. Volume managers do not support allocation at the fine grain of a file system (4KB in WAFL), so copy-on-write allocation for snapshots and clones is handled in larger units—often several megabytes. Similarly, file-system-level knowledge allows WAFL to determine when blocks are no longer in use (e.g., because they belonged to a deleted file) so it can transfer free space from a FlexVol volume back to its aggregate, enabling features such as thin provisioning.

7 Conclusion

FlexVol volumes separate the management of physical storage devices from the management of logical data sets, reducing the management burden of a file server and allowing administrators to match data set size to user and application needs. The virtualization of file volumes also provides increased flexibility for many routine management tasks, such as creating or resizing volumes or dynamically dividing physical storage among volumes. The FlexVol architecture enables many new features, including FlexClone volumes and thin provisioning of volumes. Using a few simple optimizations, we provided this expanded functionality at low cost. Many operations see no overhead when using FlexVol volumes. Even a workload with a broad functional profile, such as SFS, only shows a modest performance degradation of 4%—much less than the performance gain achieved with each new generation of hardware.

8 Acknowledgments

We thank Sudheer Miryala, the lead development manager on the FlexVol project, and Blake Lewis, who was the Technical Director for WAFL and provided many significant insights in numerous discussions. We also wish to thank the many others at NetApp—far too many to list here—whose inspiration and hard work made FlexVol volumes a reality. Finally, we thank the anonymous reviewers for their thoughtful comments.

References

[1] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of SIGMETRICS 2007: Measurement and Modeling of Computer Systems*, pages 289–300, San Diego, CA, June 2007.

[2] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 223–238, San Jose, CA, February 2008.

[3] Remy Card, Theodore Ts'o, and Stephen Tweedie. Design and implementation of the second extended filesystem. In *Proceedings of the First Dutch International Symposium on Linux*, December 1994.

[4] Gustavo. A. Castets, Daniel Leplaideur, J. Alcino Bras, and Jason Galang. *IBM Enterprise Storage Server, SG24-5465-01*. IBM Corporation, October 2001.

[5] Matthew Cheek, Scott Fafrak, Steven Hancock, Martin Moore, and Gregory Yates. *Tru64 UNIX System Administrators Guide*. Digital Press, 2001.

[6] Sailesh Chutani, Owen T. Anderson, Michael L. Kazar, Bruce W. Leverett, W. Anthony Mason, and Robert N. Sidebotham. The Episode File System. In *Proceedings of the Winter 1992 USENIX Conference*, pages 43–60, San Francisco, CA, January 1992.

[7] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, San Francisco, CA, March 2004.

[8] EMC Corporation. EMC Symmetrix DMX Architecture Product Description Guide. `http://www.emc.com/products/systems/interstitial/inter_c1011.jsp`, March 2004.

[9] Helen Custer. *Inside the Windows NT File System*. Microsoft Press, Redmond, WA, 1994.

[10] Steven Hancock. *Tru64 UNIX File System Adminstration Handbook*. Digital Press, 2000.

[11] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, San Fransisco, CA, January 1994.

[12] John Howard. An Overview of the Andrew File System. Technical Report CMU-ITC-88-062, Information Technology Center, Carnegie-Mellon University, 1988.

[13] John Howard, Michael Kazar, Sherri Menees, David Nichols, M. Satyanarayanan, Robert Sidebotham, and Michael West. Scale and Performance in a Distributed System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[14] Weihang Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady Kanevsky. Are Disks the Dominant Contributor for Storage Failures? In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, February 2008.

[15] Michael Kazar. Synchronization and Caching Issues in the Andrew File System. Technical Report CMU-ITC-88-063, Information Technology Center, Carnegie-Mellon University, 1988.

[16] Michael L. Kazar, Bruce W. Leverett, Owen T. Anderson, Vasilis Apostolides, Ben A. Bottos, Sailesh Chutani, Craig F. Everhart, W. Antony Mason, Shu-Tsui Tu, and Edward R. Zayas. DEcorum File System Architectural Overview. In *Proceedings of the Summer 1990 USENIX Conference*, pages 151–164, Anaheim, CA, June 1990.

[17] A.J. Lewis. LVM HOWTO. `http://www.linux.org/docs/ldp/howto/LVM-HOWTO`, 2006.

[18] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *Computer Systems*, 2(3):181–197, 1984.

[19] J. Menon, D. A. Pease, R. Rees, L. Duyanovich, and B. Hillsberg. IBM Storage Tank—A heterogeneous scalable SAN file system. *IBM Systems Journal*, 42(2):250–267, 2003.

[20] Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Steve Kleiman, and Shane Owara. SnapMirror: File-System Based Asynchronous Mirroring for Disaster Recovery. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*, pages 117–129, Monterey, CA, January 2002.

[21] M. Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[22] Margo Seltzer, Greg Ganger, M. Kirk McKusick, Keith Smith, Craig Soules, and Christopher Stein. Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 71–84, San Diego, CA, June 2000.

[23] Robert Sidebotham. VOLUMES: the Andrew File System data structuring primitive. In *Proceedings of the European UNIX Systems User Group*, pages 473–480, September 1986.

[24] SPEC SFS (System File Server) Benchmark. `http://www.spec.org/osg/sfs97r1`, 1997.

[25] Sun Microsystems, Inc. ZFS On-Disk Specification. `http://www.opensolaris.org/os/community/zfs/docs/ondiskformat0822.pdf`, 2006.

[26] Sun Microsystems, Inc. Solaris ZFS Administration Guide. `http://www.opensolaris.org/os/community/zfs/docs/zfsadmin.pdf`, 2008.

[27] Symantec Corporation. Veritas Volume Manager Administrator's Guide. `http://sfdoccentral.symantec.com/sf/5.0/solaris/pdf/vxvm_admin.pdf`, 2006.

[28] Mark Wittle and Bruce E. Keith. LADDIS: The Next Generation in NFS File Server Benchmarking. In *Proceedings of the Summer 1993 USENIX Technical Conference*, pages 111–128, Cincinnati, OH, June 1993.