FlexGroup Volumes: A Distributed WAFL File System

Ram Kesavan, *Google;* Jason Hennessey, Richard Jernigan, Peter Macko, Keith A. Smith, Daniel Tennant, and Bharadwaj V. R., *NetApp*

https://www.usenix.org/conference/atc19/presentation/kesavan

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the 2019 USENIX Annual Technical Conference is sponsored by USENIX.

FlexGroup Volumes: A Distributed WAFL File System

Ram Kesavan,* Jason Hennessey, Richard Jernigan, Peter Macko,

Keith A. Smith, Daniel Tennant, and Bharadwaj V. R., *NetApp, Inc.*

Abstract

The rapid growth of customer applications and datasets has led to demand for storage that can scale with the needs of modern workloads. We have developed FlexGroup volumes to meet this need. FlexGroups combine local WAFL® file systems in a distributed storage cluster to provide a single namespace that seamlessly scales across the aggregate resources of the cluster (CPU, storage, etc.) while preserving the features and robustness of the WAFL file system.

In this paper we present the FlexGroup design, which includes a new *remote access layer* that supports distributed transactions and the novel heuristics used to balance load and capacity across a storage cluster. We evaluate FlexGroup performance and efficacy through lab tests and field data from over 1,000 customer FlexGroups.

1 Introduction

With each new generation of hardware, computers become faster and more powerful. CPUs have more cores, memory is cheaper, networks are faster, and storage devices hold more data. Despite these advances, however, many modern applications require far more resources than a single machine can provide, leading to an explosion in the use of distributed applications and systems.

Network-attached storage solutions have evolved similarly from single-node to distributed systems. NetApp® Write Anywhere File Layout (WAFL) [11] was launched more than 20 years ago as a single-node, single-volume file system. Over time, we increased WAFL flexibility and scale by allowing many file systems per node [7], and by scaling performance with increasing core counts [5]. Today we have hundreds of thousands of storage controllers at customers' sites. But like other file systems, our single-node scale and performance have been limited by the resources (storage, memory, CPU, network) of a single machine.

This paper describes *FlexGroups*, a new feature that *automatically* balances capacity and load across the nodes of our storage cluster. A FlexGroup combines volumes from multiple nodes of a cluster into a single client-visible namespace and allows any directory entry to point to any inode on any of the member volumes. The FlexGroup selects a location for each new file or directory using heuristics that attempt to balance capacity and load. The result is a single file system that scales across the resources of an entire storage cluster.

Within a storage cluster, FlexGroups use a new *Remote Access Layer* (RAL) to perform transactional updates across multiple member volumes (for example, a directory entry on one node and the target inode on a different node). Because operations that span multiple FlexVols® necessarily introduce overhead, the FlexGroup heuristics dynamically adjust the levels of remote inode placement to minimize that overhead while still achieving a balanced system.

FlexGroups impose performance overhead on metadata operations that create or traverse cross-node links in the namespace. But the more costly of these operations, such as `MKDIR`, are relatively rare, and the overheads are modest compared to the larger latencies of I/O to SSDs or HDDs. On large mixed workloads, a FlexGroup performs comparably to a similarly sized group of individual volumes while providing the benefits of automatic scaling and balancing of the data and load across nodes.

This paper makes several contributions. We present a scalable distributed file system architecture that builds on an underlying single-node file system while preserving all of its features. We describe low-cost heuristics that dynamically balance load and capacity with little performance overhead. Finally, we analyze more than 1,000 customer FlexGroup deployments to evaluate our load balancing heuristics, understand challenging use cases, and identify improvements to FlexGroup heuristics.

2 The Building Blocks

A FlexGroup is a distributed file system built from a cluster of nodes running our storage operating system, NetApp Data ONTAP®, which includes WAFL [11], our proprietary copy-on-write (COW) file system. A FlexGroup is com-

*Ram Kesavan is currently at Google.

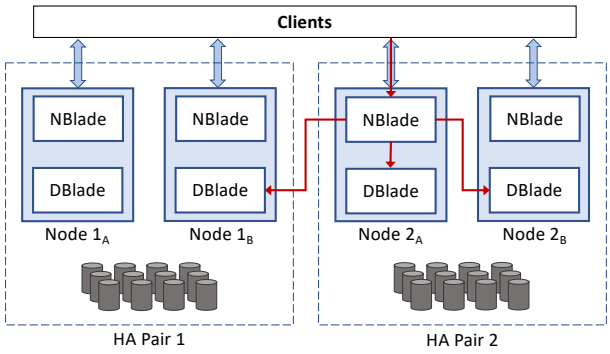

Figure 1: Arrows indicate some possible routing paths for operations. The NBlade in node 2_A may route a request to the DBlade on the same node, on the HA partner node (2_B), or to another node in the cluster (1_B).

posed of multiple single-node file systems called FlexVols; a FlexVol [7] is an exportable WAFL file system. As in other COW file systems [24, 30], every modified block of data or metadata in WAFL is written to a new location; only the superblock is ever written in place. The file system supports many enterprise features, such as snapshots [18, 19], replication [29], storage efficiency (compression, deduplication, etc.), and encryption.

A cluster of ONTAP [25] nodes is organized as multiple high-availability (HA) pairs of nodes. Each pool of storage is accessible by a different HA pair. A node is composed primarily of two software modules [8]: an *NBlade* and a *DBlade*. The NBlade is composed of the networking and protocol stacks (NFS, SMB, iSCSI, NVMe, etc.) that communicate with clients. The DBlade is composed of the WAFL file system, RAID [2, 9] module, and storage drivers to interact with the storage media. The storage pool dedicated to an HA pair is partitioned into *aggregates*, each of which can house hundreds of FlexVols. The storage devices in an aggregate are collected into RAID groups to protect against device failures [28]. Each aggregate is managed in an active-passive manner by the nodes of its HA pair. Thus the DBlade on each node serves requests for one or more aggregates while also acting as a standby to take over the HA partner's aggregates if the partner fails.

ONTAP exports file system namespaces over virtual interfaces (VIFs) that are mapped to nodes in the cluster. A client accesses the namespace within a FlexVol over NFS or SMB by connecting to a node via the corresponding VIF [8]. As illustrated in Fig. 1, each request is routed from the receiving node's NBlade to the DBlade of the node owning the FlexVol that contains the requested data. The NBlades consult a consistent, replicated database of volume information, which maps volume IDs to DBlades. Responses are routed back from a DBlade to the client via the NBlade that is connected to the client. Any operation that is routed to a different node pays a small latency penalty compared to one that is routed to the same node [8].

WAFL accelerates performance by placing a write-ahead log in NVRAM [11, 33] and provides a fast failover capability by mirroring this log to its HA partner's NVRAM. If a node (1_A) fails, the HA partner (1_B) takes ownership of its network interfaces and storage, replays its NVRAM log (aka NVLog), and starts servicing operations to those FlexVols almost instantly, thereby minimizing client outage. Even in the rare event that NVLog is lost—for example, if one node "panics" and the NVRAM devices on both nodes of the HA-pair are damaged—the COW nature of updates guarantees the consistency of the persistent file systems on the failed node [11, 24, 30]; only state from operations logged in the last few seconds is lost. In other words, no (fsck-style) repair of the file system is needed in such cases.

3 FlexGroup Volumes

ONTAP with FlexVols had been commercially successful with a wide customer deployment for several years before we tackled the problem of building a distributed namespace that seamlessly scales across the storage pools and compute resources of all nodes in a cluster. ONTAP already allowed the administrator to manually *junction* together multiple FlexVols in a cluster to export a namespace that spanned multiple nodes, but it was impossible to build an efficient junctioned namespace because avoiding conditions of imbalance—storage capacity or load—required a priori knowledge of future behavior of the applications. Customer deployments need directory namespaces to autonomously consume capacity and compute resources from nodes in the cluster with minimal administrator involvement.

Customers have high requirements of an enterprise-grade distributed file system: resiliency, availability, ease of administration, and other "standard" features, such as snapshots, replication, cloning, compression, deduplication, and encryption. Adding this comprehensive list of features into an existing open-source file system or creating a new feature-rich distributed file system would have been prohibitively expensive in terms of engineering resources and time.

Because these features had been baked into FlexVols over decades, building FlexGroups out of a collection of FlexVols was the obvious and prudent approach. It greatly simplified several aspects of our design. For example, the ability to nondisruptively move FlexVols between nodes is particularly useful for coarse-grained rebalancing without embedding any forward pointers in the file system; for more on this topic, see Sec. 3.3. Moreover, the NVLog, together with the transactional semantics of FlexVols [2, 7, 9, 17, 20, 29] provides the atomicity and reliability for maintaining metadata necessary for the distributed namespace.

More importantly, this choice simplified existing customers' workflows—a FlexGroup looks much like a FlexVol to the administrator and NFS or SMB clients—which was

key to its rapid adoption. FlexGroups also allow "upgrading" an existing FlexVol to a FlexGroup, on-the-fly addition of member FlexVols, presentation of a FlexGroup as a single volume for all administrative tasks, and other management features. Such details are outside of the scope of this paper.

3.1 Design Considerations

Data distribution comes at a cost. Typically, spreading data across nodes requires either a metadata server or internal pointers to redirect from one location to another. Maintaining this metadata costs additional CPU, network, and storage. Coarse-grained distribution of large directory subtrees minimizes metadata costs, but makes it harder to achieve balance between nodes. A distributed system can achieve better balance with a finer-grained distribution, but that comes with the additional cost of more pointers.

Because both fine- and coarse-grained distribution have drawbacks, a promising approach would be to offer a mechanism that can adaptively change distribution granularities based on need—using fine-grained distribution when necessary to ensure the use of all resources, while reverting to coarse-grained distribution when possible for higher overall performance.

An alternative approach to achieve fine-grained distribution without metadata overhead is to use hash-based algorithms to place each file on a pseudo-random node. We ruled this approach out for two reasons. First, hash-based placement causes a large fraction, $(n-1)/n$, of requests to pay the cost of a network hop (NBlade to DBlade) in an n-node system. More intelligent placement of files can reduce hops; for example, files within a directory could be colocated in the common case. Hash-based placement also prevents dynamic placement decisions based on current conditions; for example, avoiding new file creation on a node that is experiencing high load.

Ideally, a system should be capable of reacting to imbalance in load and storage space availability. Retroactive data movement has two serious challenges—picking the content to be moved and doing it in a *nondisruptive* fashion to the clients (i.e., no remounts). The latter requires inserting remote pointers in the file system to avoid invalidating file handles that were previously issued to clients.[1] However, the proliferation of these pointers over time creates an ever-increasing drag on overall system performance. Additionally, although centralizing metadata in one server simplifies some aspects, it leads to obvious performance bottlenecks.

We explored two less-successful designs before FlexGroups. In our first design, all data files were striped (based on file offset and range) across member FlexVols. Subsequently, metadata (directories and inode tables) were also striped and decentralized. A distributed ticketing mechanism was used to maintain cross-FlexVol consistency. However, the resultant fine-grained synchronization generated a high performance tax. In addition, increase in CPU core count in nodes and scaling improvements to the WAFL parallelism model [5] enabled concurrent execution of dozens of read and write operations to a single file, which obviated the need to stripe "hot" files.

The second design stored an inode indirection layer in a master FlexVol, which pointed to data files that were placed in other FlexVols based on simple round-robin policies. The design traded the extra FlexVol hop paid by each operation to consult and follow the indirection for the ease of moving data and metadata across the member FlexVols in response to imbalance in load or storage capacity. However, the design was unable to prevent the master FlexVol (hosting the indirection layer) from becoming a performance bottleneck.

To demonstrate the overhead of this indirection-based scheme, we benchmarked it against FlexGroups, which, as explained later in this section, have several important differences. FlexGroups use pointers rather than indirection to locate remote files and try to place each file in the same node as its parent directory so that most operations avoid an extra NBlade-to-DBlade hop. Finally, FlexGroup members are symmetric; no node or FlexVol necessarily handles more (or less) data, metadata, or traffic than its peers.

We ran SPEC SFS 2014 SWBUILD [35] on both a FlexGroup and the indirection-based approach using a mid-range HA pair.[2] The operational throughput (ops/s) of FlexGroups was more than 4 times that of the indirection approach. At low load points, both were easily able to handle the workload, but the average request latency of the indirection approach was 1.9 times higher than that of FlexGroups, reflecting the extra network hop required by every request.

This pattern continued at higher load points until the master FlexVol became a bottleneck in the indirection-based system. Queue lengths grew rapidly with increasing load, until a 20-fold increase in time spent to resolve indirection which caused the operational throughput to collapse in spite of the fact that almost half of the available CPU cores remained idle. In contrast, FlexGroup performance continued to scale until the cores on both nodes were almost fully utilized. Note that this test was performed on a 2-node cluster. The indirection bottleneck is more acute in a larger cluster.

3.2 Fusing FlexVols via Remote Hardlinks

FlexGroups distribute both data and metadata across multiple FlexVols by allowing directory entries to be *remote hardlinks* to inodes on other member FlexVols, as illustrated in Fig. 2. A client can connect to any node in the cluster, and the NBlade routes its requests to the appropriate FlexVol and DBlade.

FlexGroups perform most data distribution during *ingest*: an intelligent, immediate, and permanent placement decision

[1] NFS requires long-lived file handles.

[2] Each node had 16 cores and 96 GiB of DRAM.

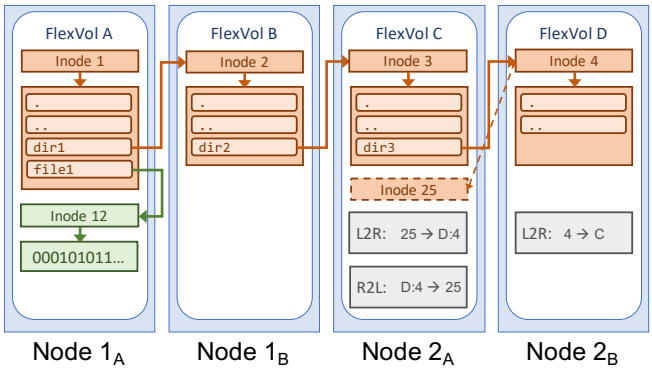

Figure 2: An example of a FlexGroup (DBlades only) that consists of member FlexVols in four nodes, along with hardlinks following `/dir1/dir2/dir3`. L2R and R2L are local-to-remote and remote-to-local databases. FlexVol C contains inode 25, which is a cached version of inode 4 from FlexVol D.

is taken when creating a new file or directory, thereby assigning the new object to one of the FlexVol members of the FlexGroup. FlexGroups adjust the granularity of distribution from the directory level down to individual files as necessary.

As explained in Sec. 3.1, striping individual files was not necessary to achieve our performance goals. Therefore, FlexGroups do not stripe individual files or directories across member FlexVols; each object lives entirely in one FlexVol. This greatly simplifies the design and allows all *data* operations (reads and writes) to be executed with the *same latency and throughput*. In other words, remote hardlinks are not traversed when servicing such operations; an additional network hop is incurred only if the NBlade connected to the client and DBlade in which the data resides are on different nodes. Only *metadata* operations that resolve or modify remote hardlinks incur additional latency for the extra internode communication.

The FlexVol identifier and the inode identifier are encoded within the opaque file handle that WAFL returns to operations such as `LOOKUP` and `OPEN`. An NFS client resolves `/dir1/dir2/foo` by starting at the root located in FlexVol A. Suppose that the client connects to Node 1_A and sends a `LOOKUP` operation for `dir1` to that NBlade. This operation is directed to the same node's DBlade, which contains the root directory, and the remote hardlink for `dir1` is located. The NFS handle returned to the client has B:2 encoded within it. The client then sends a `LOOKUP` for `dir2` using this opaque handle, which the NBlade forwards to FlexVol B (Node 1_B). The NFS handle returned to the client now embeds C:3. A `LOOKUP` for file `foo` returns a file handle that encodes the FlexVol (which could even be A) and the inode for that file. All subsequent reads and writes on that file get routed by the NBlade of Node 1_A directly to the correct DBlade with no further access of the remote hardlinks. Thus, as mentioned earlier, every data operation costs the same and is independent of the number of remote hardlinks that led to it.

Unlike NFS, the SMB client sends the entire pathname, starting from the root. For example, an SMB client may `OPEN /dir1/dir2/foo`, whereas an NFS client sends a series of `LOOKUP` operations that walk down to `foo`. NBlades accelerate these SMB operations by maintaining in-memory pathname-to-FlexVol tables that can be used to resolve an entire pathname to a FlexVol, similar to Sprite prefix tables [41]. Depending on the state of the table, the `OPEN` operation may avoid resolving some or all remote hardlinks in the pathname when it gets routed to a FlexVol. The resolved hardlinks are added to this table. Much as with NFS, once `OPEN`ed all subsequent reads and writes to that file are routed directly to the correct DBlade.

3.3 Load Balancing

Based on experience with prior designs and customer workloads we realized that remote hardlinks with ingest-based placement yield a close-to-optimal trade-off by reducing the necessary synchronization overhead even while reducing the likelihood of imbalances in storage capacity and load across member FlexVols. Sec. 3.5 explains how placement heuristics are based on both the recent IOPS load imbalance and the capacity imbalance across member FlexVols. Obviously, these heuristics cannot predictably avoid load imbalances in the future. For example, it is possible that data written aptly at ingest time to one (or a few) member FlexVols becomes "hot" several hours or days later due to some application workflow. As explained earlier, nondisruptive, retroactive, and fine-grained data redistribution requires creating permanent remote markers that will regress the overall file system performance over time. Instead, such a pathologically imbalanced FlexGroup is fixed efficiently and expeditiously by coarse-grained movement of "hot" member FlexVols to nodes that are sustaining less IOPS load. This is done by leveraging the *Volume Move* feature [26], which leaves no residual remote markers in the file system namespace of the FlexGroup.[3]

3.4 The Remote Access Layer

FlexGroup volumes work by having one member FlexVol take responsibility for executing an entire operation, updating its own state, and coordinating the state changes with the other members. Each time an operation needs to create, delete, or access a remote hardlink, control passes to the *Remote Access Layer (RAL)*. The RAL is responsible for managing and updating remote hardlinks in a transactional

[3] Movement is accomplished by taking periodic snapshots of the FlexVol and incrementally transferring all changes to the destination node. The frequency of these transfers increases as the movement catches up with the most recent version of the FlexVol at which point the volume information database consulted by the NBlades is atomically updated.

manner and allowing the existing file system code to operate on remote metadata as if it were local. The RAL is also responsible for recovering hardlink state after crashes and for dealing with network slowness and outages between nodes.

Conceptually, the RAL is a delegation service. When an operation accesses a remote hardlink, the RAL requests the corresponding metadata from the remote FlexVol and places it in a metadata cache where it can be accessed by the local operation. This caching represents a delegation from an *origin* FlexVol to the *caching* FlexVol. A delegation may be released proactively by the caching FlexVol or revoked by the origin FlexVol. Because the objects being cached are files or directories, each cache entry is conveniently stored in an inode at the caching FlexVol; the inode is freed when the caching relationship (and delegation) cease to exist.

The RAL persists its cached metadata to ensure that it is not lost in the event of a node failure. As described in Sec. 2, when a node fails, its HA partner takes over all of its FlexVols, replays logged operations, and resumes service of those FlexVols almost immediately. The RAL metadata caches benefit from this same high availability by persisting their state to metadata files in the FlexVols. This makes delegations fault tolerant and therefore simplifies the RAL design.

There are other approaches for tracking and maintaining distributed state, such as remote hardlinks. Porting and leveraging well-known services, such as ZooKeeper [15] or etcd [4], to maintain this state was tempting for reasons such as speed of implementation. However, building the RAL transactional mechanisms within our file system provided two major advantages: (1) low transactional overhead, because RAL bookkeeping occurs in the context of the client operation; and (2) resilient transaction tracking, because RAL bookkeeping leverages the enterprise-quality transactional semantics provided by WAFL.

The next three subsections explain the transactional infrastructure for creating and managing remote hardlinks.

3.4.1 RAL Caches

As summarized above, the RAL is a write-back metadata cache of remote inodes. The caching FlexVol creates (if necessary) and uses delegations cached in local inodes from one or more origin FlexVols to execute file system operations. These inodes may contain read-only (RO) or read-write (RW) caches. In other words, for any given object, its origin FlexVol can grant at most one exclusive RW cache to another FlexVol or multiple RO caches to multiple FlexVols.

The cached inodes are stored persistently in the regular inode table and are used by local file system operations much like regular inodes. Because they are stored in the local file system, updates to cached inodes are protected by the NVLog. This ensures that the delegations represented by cache entries are not lost in the event of node failures.

The combination of HA pairs with the mirrored NVLog, as described in Sec. 2, ensures that node failures appear, at worst, as transient delays to the RAL. When a node fails, its partner quickly takes over, replaying operations from the NVLog to recreate the RAL metadata state that had not yet been persisted to storage.

The caches are managed using *local-to-remote* (L2R) and *remote-to-local* (R2L) maps, which are implemented as B+ trees and stored in each FlexVol as hidden files. The L2R maps store two kinds of information: (1) the map from locally cached inodes to the corresponding remote inode number and origin FlexVol ID; and (2) the map of local inodes that are cached by other FlexVols. The R2L maps store the reverse mapping of the first type of L2R map data. For example, in Fig. 2, inode 25 on FlexVol C is a cached version of inode 4 on FlexVol D.

RAL caches are not intended as long-term storage, and are primarily used as (1) a temporary measure to complete metadata operations that choose to locate newly created content in a remote member; and (2) a mechanism for providing crash-consistent transactional semantics.

Caches can be evicted proactively by the caching FlexVol or revoked by the origin FlexVol. A background *scrub* process periodically walks and reclaims all caches, because caches are only needed temporarily. In fact, to improve performance, most read-write caches are aggressively evicted after their use, so that they don't have to be evicted when a later operation needs to create a cache of the inode in a different FlexVol. The scrub also reclaims all wasteful or stale outstanding references. The origin FlexVol can evict read-write caches of its inodes by examining its L2R map to find the set of caching FlexVols, sending them `REMOTE_WRITEBACK` messages to write back the dirty data, and then evicting the entries.

3.4.2 Example

We use an NFS `MKDIR` request—a sufficiently complex operation—to illustrate how RAL is used. Suppose that subdirectory `dir3` is created under `dir2`, that `dir2` is stored on FlexVol C, and that the placement logic decides to store the contents of `dir3` on FlexVol D. Fig. 2 illustrates this example.

1. FlexVol C suspends local processing of the `MKDIR` operation when it determines that it needs a remote hardlink.
2. FlexVol C sends a `RAL_RETRIEVE` message to FlexVol D, asking for the creation of a new inode and a copy of it for read-write caching.
3. FlexVol D allocates the new inode (4 in the figure), tags it as being cached elsewhere in the RW manner, and creates the corresponding L2R entry.
4. FlexVol D responds to FlexVol C by sending a `RAL_STORE` message to store the cached copy of this inode in FlexVol C.

5. FlexVol C processes the RAL_STORE message by allocating a local inode (25 in the figure), tags it as a RW cache, creates the appropriate L2R and R2L entries, and adds it to the pool of cached inodes for FlexVol D.
6. FlexVol C restarts the local MKDIR operation.

The MKDIR operation finds the cached inode in the pool and proceeds to completion using it as the proxy for dir3[4]. The MKDIR operation now creates the remote hardlink directory entry in dir2 pointing to dir3, converts the inode C:25 into a directory type, populates the . and .. entries in it, and marks the RW cache entry as "dirty."

If the client now sends a CREATE command to create a new file in dir3, that request gets routed to FlexVol D. When the operation executes, it notices the tag on dir3's inode that marks it as being in an RW cache elsewhere. The FlexVol suspends the operation, consults the L2R map to determine that it is cached in FlexVol C, and starts the eviction process. FlexVol C writes back the dirty RW cache to FlexVol D, removes it from the cache, and FlexVol D then marks the inode as a valid subdirectory with no RW caches. The CREATE operation is then resumed in FlexVol D.

As the MKDIR example illustrates, the DBlade of the caching FlexVol takes ownership of executing the client operation atomically, but only after it has created local RW caches from an origin FlexVol. We now briefly describe an example RENAME operation, which is more complex and may involve two origin FlexVols. A RENAME mv dirA/foo dirB/bar operation is routed to the DBlade that hosts dirB. Suppose that dirA is in FlexVol A, dirB is in FlexVol B, and foo is a hardlink from dirA to inodeC in FlexVol C. FlexVol B first acquires two RW caches[5]—one from FlexVol A for dirA and one from FlexVol C for inodeC—and then executes the RENAME as a regular WAFL operation that uses the two RW caches. The execution of the RENAME deletes bar if one already exists, creates a new local bar that hardlinks to inodeC, and deletes foo in the RW cache for dirA. The persistent RW cache entries are left in a "dirty" state; the origin FlexVols are modified when these cache entries are eventually flushed back.

Some operations require read-only (RO) caches; for example, an SMB OPEN operation creates RO caches for any traversed remote inodes while resolving a file pathname.

3.4.3 Consistency and Recovery

The FlexGroup consistency model builds on the transactional semantics of the underlying COW WAFL file system with its NVLog. A client operation is acknowledged only after it has been recorded in the NVLog of both the node and its HA partner. All RAL operations are similarly recorded to NVLog before they are acknowledged. After a node failure, NVLog replay returns the affected member FlexVols to a state representing a valid phase of any ongoing RAL transaction. It must be noted that the ONTAP HA-pair model (together with our RAID and WAFL software) is specifically engineered to be highly reliable[6], which informs the rest of this section.

For example, in the case of the MKDIR example, the node containing FlexVol D processes the RAL_RETRIEVE message, records the message and result in its NVLog, and responds to FlexVol C. If FlexVol D's node fails before the corresponding updates are persisted to the local file system, its HA partner replays the RAL_RETRIEVE message, recreating the inode.

If the node that hosts FlexVol C fails during the RAL_STORE phase, its HA partner may choose a different location for creating dir3 when it replays MKDIR. FlexVols D and C may be left with stale or unused cache entries or temporary inodes. This is the only allowed form of inconsistency after a node failure; it is eventually resolved by the background scrub process, which periodically walks and reclaims all caches, including all wasteful or stale outstanding references[7]. The continued existence of these stale references is safe and does not compromise the consistency of the file system.

Any dirty state left in the cache inode (25 in the MKDIR example) is recreated by replaying MKDIR if C fails after MKDIR is logged but before the subsequent file system transaction completes. Although a formal proof is not provided for lack of space, we conclude that *RAL usage for FlexGroup is crash consistent*.

Sec. 2 explains that no fsck-style repair is required if a node fails or even in the rare case that NVLog in both nodes of an HA pair is lost. When a member node in a FlexGroup fails, its HA partner replays its NVLog and recreates the file system state, including the RAL state. However, the loss of NVLog of one member FlexVol may result in inconsistencies between it and other members. Additional mechanisms were built to accomplish automatic on-the-fly repair when such an inconsistency is detected by an operation. In brief, the operation is suspended while a high-priority WAFL message investigates the inconsistency, fixes it, syslogs it, and restarts the original operation. For example, suppose that a LOOKUP consults a directory entry foo, which is a remote hardlink to an inode in FlexVol B that does not exist because FlexVol B suffered a failure followed by a loss of its NVLog. On-the-fly repair would be kicked off once the RAL_RETRIEVE fails,

[4] In theory, the MKDIR operation could find a RW cache entry in that pool on its first execution. That RW cache entry might have been created due to a RAL_RETRIEVE operation to member D initiated by a different operation. In that case, on resumption that operation kicks off yet another RAL_RETRIEVE operation to populate the pool.

[5] Each cache is acquired using the RAL_RETRIEVE and RAL_STORE handshake.

[6] The availability of our customers' systems is routinely measured at 5 to 6 nines; that is, annual system downtime between 3 and 30 seconds.

[7] The WAFL file system includes a time-tested background *scanner* infrastructure used to walk and operate on various file system metadata. The infrastructure paces itself based on current system load, thereby ensuring negligible (less than 2%) impact to client operations. There are about 20 different *scan* types, and the FlexGroup scrub is one of them.

which would eventually delete the entry `foo`. The restarted `LOOKUP` now finds no corresponding entry. A detailed discussion of on-the-fly repair techniques is beyond the scope of this paper.

3.5 Ingest Heuristics

As mentioned earlier, all placement decisions are made during ingest before the new inode is allocated. The heuristics balance two competing goals: distributing load (IOPS) and capacity among member FlexVols versus reducing the operational overhead associated with remote hardlinks.

Creating a remote hardlink to an idle member almost immediately brings traffic to it, which shifts some traffic to idle members and increases overall performance. Additionally, creating a remote hardlink to an underutilized FlexVol almost certainly causes it to fill up a little more, which helps bring its usage in line with that of its peers. This is more important when the member volumes are closer to full, and remote hardlinks can help get to every last byte of storage. Therefore, as the FlexVols become more and more full, the FlexGroup should employ more remote hardlinks.

On the other hand, every time the FlexGroup creates a new remote hardlink, there is a small performance penalty, both at the time of creation and in the future when accessing the remote hardlink. The penalties accumulated across too many remote hardlinks increase average request latency and reduce overall performance. However, too few remote hardlinks may mean a failure to use all available resources. Thus, the primary goal of ingest heuristics is to achieve the right balance while using as few remote hardlinks as possible.

3.5.1 Input to Heuristics

To minimize the overhead of heuristics, each node makes independent allocation decisions based on the following information about each member FlexVol:

- *Block usage:* The ratio of consumed to total storage space of the FlexVol.
- *Inode usage:* The ratio of the number of locally allocated inodes to the maximum number of inodes allowed for the FlexVol.
- *Recent ingest load:* The recent frequency with which it has created new files and directories (maintained by using a sliding window average across several seconds)[8].

The member FlexVols periodically exchange this information by using an asynchronous and lightweight viral *refresh process* provided by ONTAP. Each node propagates information about its local member FlexVols as well as information received from other nodes. Each node sends out a message to each peer every second and uses timestamps to decide the staleness of the received information. An asynchronous model suffices because the heuristics are reacting to trends in load and storage consumption rather than making instantaneously optimal decisions.

[8] Sec. 3.3 explains the handling of unpredictable future imbalance in load across the member FlexVols.

3.5.2 Heuristic Probability Tables

Each node consults a set of probability tables during inode allocation. The tables are recomputed approximately every second, using any new information that is received from the refresh process. The table in each FlexVol consists of two arrays:

- $RP[c]$: An array of *remote preference*—a value between 0 and 1 indicating the probability of remote allocation for objects of various categories, such as files and subdirectories at various depths from the root directory.
- $AT[m]$: An array of remote *allocation target* probability values for member FlexVol m; the sum of values across the array is 1.

When making an inode allocation decision, the FlexVol generates a random number between 0 and 1 and compares it to the corresponding value in *RP* for that category. If the randomly generated number is larger, it processes the allocation locally; otherwise it uses a weighted round-robin scheme to determine the FlexVol for remote allocation: Each remote member is assigned a target percentage based on its *AT* value, and each allocation selects the remote FlexVol that is furthest behind its target percentage compared to its peers.

3.5.3 Computing Heuristic Probability Tables

The probability tables are recomputed by comparing the properties of all member FlexVols and looking for trends and problem conditions. As mentioned earlier, each node computes these tables independently based on the information from the most recent refresh process. First, two intermediate values—*Urgency* and *Tolerance*—are computed.

Urgency biases the heuristics to react to imbalance in the FlexGroup. It is computed as a linear interpolation of each member's usage and each node's ingest load within their respective high and low thresholds, which are precomputed based on the FlexGroup configuration. An *Urgency* of 1 indicates that at least one member volume is critically low on one of these resources. Values between 0 and 1 indicate escalating degrees of concern.

Tolerance indicates how much disparity in load or usage among member volumes is acceptable. A low value tells the heuristics to react more strongly to disparities between member FlexVols. A FlexGroup that is empty with no load will have maximum *Tolerance*. As a FlexGroup gets closer to full capacity, *Tolerance* goes down, and the heuristics allow less disparity among the members.

Computing Remote Allocation Target (AT) Probabilities: First a hypothetical usage goal is computed—somewhere between the highest current capacity usage (combining both blocks and inodes) on any member and the maximum capacity of any member. The heuristics assign each member an allocation target based on the difference between that member's usage goal and its current usage. In essence, the heuristics select targets such that if all allocations were remote and all new files were exactly the same size, the remote members would then fill up at exactly the rates needed to reach their usage goals at the same time.

A non-zero *Urgency* affects this calculation significantly. A member that contributes to the non-zero *Urgency* is given a much lower target. For example, a member with the maximum *Urgency* value of 1 is assigned only 1% of its target. Once target values are assigned to each member, the values are normalized into $AT[m]$ as probabilities summing to 1.

Computing Remote Preference (RP) Probabilities: The heuristics iterate over each allocation category for a member FlexVol. Some categories are easily computed. For example, allocating a new subdirectory in the root directory is always a remote allocation to ensure that this new branch of content lands on the member with the least capacity usage or load. But for most categories, the calculation is more complex and uses the recent ingest load data from all members.

First, the heuristics compare the recent request load for an allocation category to the target load specified by the volume's allocation target (AT). If the recent load is below the target, then $RP[c]$ is set to 0, indicating a desire for the FlexVol to satisfy new allocations locally. However, if the recent load is above the target then $RP[c]$ is computed as the proportion of the load that is in excess of the target. For example, if a member with a target of 8% has recently received 10% of the overall allocations in a category, then that category is assigned a *RP* of 0.2 so it can attain its target. As an optimization, *RP* is reduced for members that have exceeded their target by less than the current *Tolerance* value, optimistically allowing them to keep a higher percentage of local traffic for local placement.

Again, a non-zero *Urgency* value for a member increases its *RP* values. As a member FlexVol or node runs low on resources, the allocations are more likely to happen on remote peers.

4 Topics in Practice

Building a scale-out file system required meeting customer expectations of the features, performance, and robustness that they were accustomed to with FlexVols. This section touches on a few selected topics related to satisfying those expectations.

Snapshots: Several features of ONTAP depend on the ability to efficiently create consistent snapshots. A FlexGroup snapshot is a collection of snapshots, one per member FlexVol, that are created in a coordinated fashion. First, each member FlexVol fences all new client operations and evicts all RW caches. Then each member FlexVol independently creates a snapshot and drops its fence. Because very few RW caches are outstanding at any point in time, this fencing creates no noticeable disruption to client performance (both latency and throughput of client operations). The design choice to evict all RW caches was made to avoid extra implementation work in various internal file system operations to understand, handle, and traverse RAL information when accessing snapshots. Eviction is not really necessary because the L2R and R2L metadata is consistently captured in each member FlexVol snapshot, and could be used to service reads of the FlexGroup snapshot image. The metadata can also be reactivated in the case of a restore of the entire FlexGroup to that coordinated image.

Quotas: Tracking and enforcement of user, directory, and group quotas must treat the entire FlexGroup as a single entity. Any incoming operation must fail when a quota rule is violated. Caches created by the RAL infrastructure count toward quota consumption. Quota credits are pro-actively distributed across member FlexVols to allow efficient, independent, and per-operation granular enforcement of the rules. In the worst case, an operation may be suspended while the FlexVol communicates with other members to borrow credits; the design makes such scenarios extremely rare.

Unreachable Member FlexVols: One or more members may become temporarily unreachable; for example, due to network problems. All client and RAL operations directed to those FlexVols will time out and get retried. Meanwhile, access to data in other FlexVols continues as usual. ONTAP clustering services indicate whether the FlexVols still exist, whether the outage is temporary, and whether retries will eventually succeed. If the problem is not transient or if clustering services indicate that FlexVols have been destroyed, either the FlexGroup can be restored to its most recent coordinated snapshot[9] or on-the-fly repair will eventually fix RAL metadata that point to the lost FlexVols. Both approaches recover file system consistency but result in data loss. The former is typically preferred because the loss is recent and predictable; all mutations after the snapshot are lost.

Testing: Enterprise-grade quality implies continuous validation; 102 different test suites are executed, totaling 160 hours of runtime daily. These tests use both NFS and SMB clients to specifically stress cross-member code paths that use RAL. Many of the suites also inject errors, such as dropping RAL operations, forcing node panics to trigger HA events, discarding NVLog during HA-events, and artificially creating memory pressure. There are also suites that explicitly create inconsistencies in the persisted RAL metadata to test on-the-fly repair mechanisms that correct them.

[9] Snapshots can be replicated to and restored from remote nodes by using NetApp SnapMirror® [29].

5 Evaluation

This section shows that the load-balancing automation of FlexGroups compares well to an ideal FlexVol in three areas: overhead, scale, and balance. It is not practical to formulate an apples-to-apples comparison of FlexGroup to other well-known distributed file systems, due to the difference in configurations, sizes, and associated feature sets. Instead, this section compares FlexGroup performance to ideal and worst-case scenarios that are manually configured (as explained below). Experiments to measure FlexGroup overhead (Sec. 5.1) and scale (Sec. 5.2) were completed using a cluster of up to 8 nodes, each with two 6-core Intel Broadwell DE processors, 64 GiB of DRAM, and a shelf of 24 SSDs. Capacity balancing was validated with data collected from customer deployments (Sec. 5.3).

5.1 Overhead

Automatic redirection of files and directories between FlexVols in a manner that is consistent in the face of faults adds two major sources of overhead: RAL (Sec. 3.4) introduces additional overhead for metadata operations, and some operations incur an additional network communication cost when the client sends a request to an NBlade that cannot be satisfied by the local DBlade. We measure these overheads by comparing the performance of FlexGroups to two manually created configurations with FlexVols, neither of which incur the overhead of RAL: (1) **FlexVol-Local** is an ideal configuration in which operations are routed by each NBlade to the DBlade on the same node. (2) **FlexVol-Remote** is a configuration in which operations are always routed by each NBlade to a DBlade on a different node.

Even though we used a single HA pair in these experiments, the results in these two cases are independent of the number of nodes: Either none (FlexVol-Local) or all (FlexVol-Remote) operations involve inter-node processing, and this does not change with the number of nodes. In the case of FlexGroups, the remote-to-local ratio might increase with the number of nodes, but no other are latencies added.

5.1.1 Overhead of NFS Operations

We measured overhead by generating a metadata load using `mdtest` [14] to a single HA pair. We used a single mdtest client connected by using NFSv3 to one of the nodes, with no other load on the system. We report the latencies of the individual metadata operations, measured on the storage systems. We configured mdtest to create approximately 2 million directories and 2 million 256KB files; the maximum NFS transfer size was set to 64KB.

Fig. 3 shows the normalized results. We observe that in the FlexGroup case, data is spread approximately evenly across the two DBlades. Read-only metadata operations that use NFS file handles, namely `ACCESS` and `GETATTR`, exhibit performance that is almost exactly halfway between the local and remote cases because there is no RAL overhead. Satisfying the request incurs additional communication across the cluster interconnect approximately 50% of the time. The `LOOKUP` operation incurs additional latency when resolving a name in a directory that happens to be a remote hardlink, because it creates a RO cache of the looked-up inode on the same node as its parent directory[10]. Metadata update operations, such as `CREATE`, `MKDIR`, `REMOVE`, and `RMDIR`, show the overhead of RAL and of communicating over the cluster interconnect, each occurring roughly 50% of the time.

Even though many of these metadata operations incur an overhead, they are relatively infrequent compared to data operations. Read operations exhibit performance that is approximately halfway between the local and remote cases. Write operations perform comparably worse than read operations because they require updating file inodes; for example, to extend the file lengths and to update `mtime`. As shown later in this section, the overall impact on performance is minimal when looking at the operations in aggregate.

5.1.2 Application and Data Benchmarks

Fig. 4 shows the performance of our application benchmarks, expressed as normalized operations per second. (Higher is better.) The figure presents two sets of results: random and sequential read and write benchmarks, and selected SPEC SFS 2014 benchmarks [35], which evaluate a realistic mixture of file system operations.

We generated the random and sequential results for reads and writes by using an internal benchmark that increases the load to find the maximum throughput possible while placing data in accordance with the configuration (FlexVol-Local, FlexVol-Remote, or FlexGroup). Unlike in the mdtest experiment, in which only one NBlade and one DBlade were active at any given time, in the FlexVol-Local case, we used several clients to saturate both nodes with requests. The results indicate that throughput for FlexGroups achieves a balance somewhere between the best (FlexVol-Local) and worst (FlexVol-Remote) cases.

SFS [35] is a standard file system workload generation tool that comes with profiles generated from real-world examples. We use three SFS profiles representing different mixes of metadata requests and data throughput [37]: **SWBUILD,** heavy metadata similar to Linux kernel builds; **EDA,** a balance of metadata and data throughput representative of electronic design automation applications; and **VDA,** streaming writes and few metadata operations, similar to a video recording system. Our goal in these benchmarks is to determine peak operations/second, not to produce compliant SFS numbers as defined by SPEC [36].

[10] The performance of `LOOKUP` is independent of the path length because NFSv3 resolves each pathname component separately.

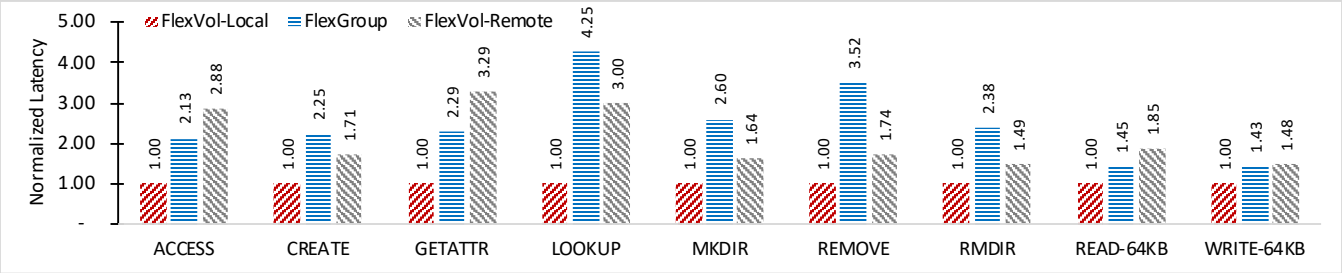

Figure 3: Server-side latency for select NFS operations generated by a single mdtest client, reported relative to FlexVol-Local latencies. Lower scores indicate better performance.

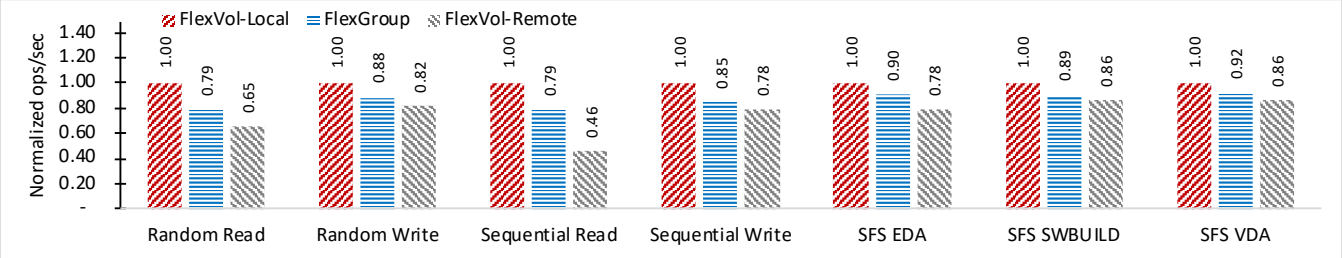

Figure 4: Operations per second for application and data benchmarksrelative to FlexVol-Local. Higher scores indicate better performance.

For the EDA and VDA workloads, FlexGroup performance fell almost exactly between FlexVol-Local and FlexVol-Remote, showing that the load-balancing heuristics achieved a balanced distribution of files across the two test nodes and that there was negligible overhead from RAL. In the SWBUILD test, FlexGroup performance was again between the extremes but closer to FlexVol-Remote. In this test, 87% of requests are metadata operations, increasing overhead due to more frequent RAL processing.

We also determined via profiling that the recomputation of the heuristic probability tables every second by each node adds a negligible amount of CPU cycles (less than 0.001%), even for large customer deployments.

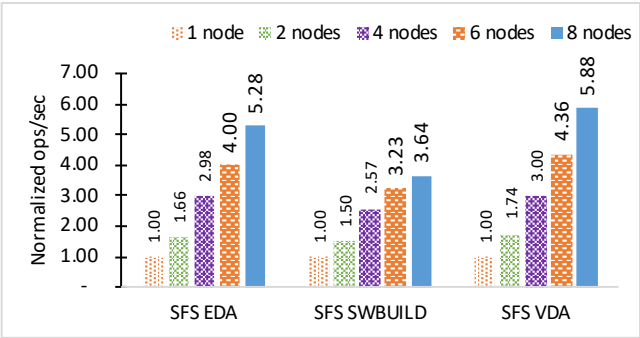

Figure 5: Scalability of FlexGroups. The max achieved operations/second for three SFS profiles, run against 1, 2, 4, 6, and 8 nodes. Each metric is reported relative to a 1-node cluster. Higher scores indicate better performance.

5.2 Workload Scaling

To examine the FlexGroup scalability under different workloads, we ran the three SFS profiles against 1-, 2-, 4-, 6-, and 8-node clusters (as described at the beginning of Sec. 5). Fig. 5 shows the scaled results. Workloads with lower metadata intensity (VDA) scale better than workloads with more metadata operations, as expected due to the added transaction overhead associated with RAL, discussed in Sec. 3.4.

5.3 Customer Experience

FlexGroups became available to customers in early 2017. In this section, we share information about how customers have used FlexGroups and about improvements made to our heuristics based on that experience. The storage systems can "phone home" to report customer configuration and event data [23]. This functionality is optional, but a large fraction of our customers enable it. Because our customers typically configure multiple FlexGroups and other FlexVols on the same nodes of a cluster, run dozens of applications on any given node of the cluster at various times, and ONTAP does not fully gather IOPS statistics on a per-FlexVol basis, it is not possible to compute or to clearly show whether the IOPS load stays balanced across member FlexVols over a long period of time. On the other hand, per-FlexVol capacity consumption is tracked for imbalance, and it also serves as a good proxy for imbalance in load.

This data shows a steady rise in FlexGroup adoption in

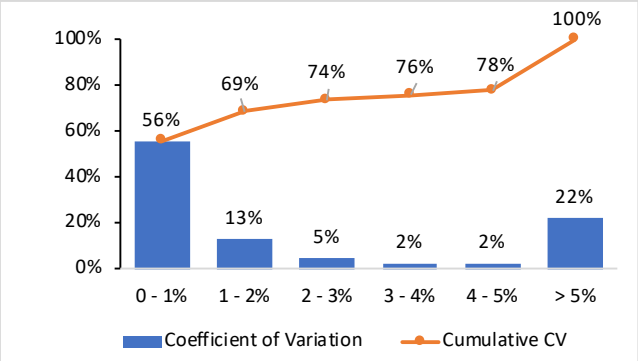

Figure 6: Do the placement algorithms balance data in the real world? This histogram shows the dispersion of FlexVol usage in customer-deployed FlexGroups using the coefficient of variation (CV). CV is the standard deviation normalized by dividing by the mean (σ/μ).A smaller CV indicates lower dispersion. Only FlexGroups greater than 10TB in size and more than 5% utilized were included.

the 2 years since its release. As of August 2018, hundreds of customers have deployed thousands of FlexGroups to manage hundreds of petabytes of storage. Roughly half of these FlexGroups are small ($<$ 10TB), and we surmise that they are being used for testing and evaluation. 25% of these FlexGroups are larger than 100TB, and 5% are larger than 1PB. A handful of FlexGroups are larger than 5PB. Of the FlexGroups that are 10TB or larger, most (70%) have between 8 and 32 member FlexVols. 5% are larger, with the largest containing over 150 members.

Customer data also provides insight into the effectiveness of our ingest heuristics at balancing capacity across the members of a FlexGroup. Standard deviation (stdev or σ) measures dispersion. However, because different FlexGroups contain different amounts of data, the standard deviations are not directly comparable. For example, a stdev of 10^5 bytes would be interpreted differently for a 1TB FlexGroup than for a 100TB one. To normalize these numbers for comparison, we divide the stdev by the mean (μ), producing a coefficient of variation (CV), or σ/μ, that gives the stdev as a percentage of the mean.[11] A FlexGroup with a CV of 1% suggests that each FlexVol has a 95% probability of being $\pm 0.02\mu$; a CV of 3% indicates a 95% probability of being $\pm 0.06\mu$.

The data indicate that the FlexGroup placement algorithms are working in most customer use cases, as shown in Fig. 6. Over half (56%) of FlexGroups had a CV less than 1%, 78% of FlexGroups had a CV less than 5%, and 85% were below 10%. Only FlexGroups larger than 10TB and more than 5% utilized were included in this analysis. For the 15% of FlexGroups that had a CV greater than 10%, we found three patterns that account for most of the cases.

[11] CV is also known as relative standard deviation.

First there is a group of FlexGroups that have a bi-modal usage pattern—one set of members with similar high usage and another set with similar, but lower, usage. These appear to be FlexGroups that customers have expanded by adding a set of new member FlexVols, and the newer members show lower usage than the older ones.

The second pattern includes FlexGroups that hold a small number of very large files. An example would be a 200 TB FlexGroup with a small number of backup archives averaging 100 GB each. In these cases there aren't enough files to average out the effects of our ingest decisions, and the impact of allocating (extremely large) outlier file sizes can increase the CV of the FlexGroup.

Finally, there is a small number of FlexGroups that are well balanced except for a single volume with much higher usage. We believe that these cases are caused by oddities in customer workloads: an output archive file that encompasses an enormous amount of workload data, or a directory of log files that were co-located when they were created, but have grown very large over time.

It should be noted that such examples of capacity imbalance do not necessarily imply imbalance in IOPS load across the member FlexVols. In the second pattern, backup files experience sequential appends and infrequent sequential reads. In the third pattern, output archive files experience infrequent bursts of append operations. The WAFL file system is well tuned to absorb a spike in reads and writes to a single FlexVol. And as mentioned earlier, in the worst case a Volume Move operation can relocate such a FlexVol from a node that happens to be overloaded.

Based on specific customer experiences, we have improved the ingest heuristics over the four releases since FlexGroups were introduced. Early on, an interesting customer case motivated the addition of the inode usage property to the refresh process. Without accounting for inode usage, the heuristics had kept several million small files in a few members to balance out some extremely large files in the others. The per-FlexVol limit on the number of inodes was hit, causing out-of-space errors. Another customer experience resulted in converting the ingest load into a sliding window average to accommodate spikes. Other tweaks were made to various constants used while recomputing the probability tables to prevent the heuristics from over-reacting to changes.

5.4 Applicability Beyond WAFL

The FlexGroup design builds on years of engineering investment in our WAFL file system, but could these concepts be applied to other file systems? FlexGroups required modest changes to WAFL, and we believe that similar enhancements are possible to other file systems: remote hardlinks and the ability to traverse, create, and delete them. The inputs to ingest heuristics are simple and should be easy to implement.

The biggest challenges may be availability and fault tol-

erance. A robust file system like WAFL can persist RAL metadata locally with its reliable native consistency semantics. Other file systems may require more expensive distributed consensus techniques, like two-phase commit [22] or Paxos [21], to ensure fault tolerant updates to remote hardlinks.

6 Related Work

Many distributed file systems have been developed by the research, commercial, and open source communities. To discuss FlexGroups in the context of this large body of prior work, we focus on file systems that share our design goals and implementation choices. Thus we emphasize systems in which storage devices are controlled by a single node (or two nodes for fault tolerance) and in which nodes manage data at the granularity of files or objects. We will not discuss shared disk file systems such as GPFS [32] or Frangipani [38].

Distributed file systems use a variety of strategies for assigning files and directories to nodes. The simplest approach is a static partitioning, whereby an administrator assigns namespace subtrees to different servers. This strategy is exemplified by systems like NFS [31], AFS [13], and Sprite [27]. It was also the approach that ONTAP supported prior to the introduction of FlexGroups [8].

The disadvantage of static namespace partitioning is uneven load and capacity balancing. Dynamic distribution addresses this challenge by selecting or updating file locations on the fly. Slice [1] compared two heuristics for dynamic partitioning. *Name hashing* maximizes balancing by assigning every new file and directory to a random node chosen by hashing its name. *Mkdir switching* maintains namespace locality by assigning a configurable fraction of new directories to a different node than their parent and allocating all other files and directories on the same node as their parent.

The name hashing strategy has also been used in other distributed file systems including Vesta [3] and GlusterFS [12]. Like mkdir switching, FlexGroups aim to maintain namespace locality by rarely using remote links. But FlexGroups also uses current load and capacity in deciding when to split the namespace.

Unlike FlexGroups, many distributed file systems separate namespace management from data storage. These designs have nodes that store objects and a metadata service that maps filenames to objects. For scalability, some systems have multiple metadata servers, introducing the same tradeoff between load balancing and namespace locality that we address in FlexGroups. Ceph [39] is a widely-used system of this type. Unlike FlexGroups, Ceph repartitions the namespace in response to observed load. This is facilitated by Ceph's use of separate metadata servers; migrating a namespace subtree does not require moving the corresponding data objects. Policies for migrating subtrees is an area of ongoing research [34]. Ceph manages data placement using a hash-based algorithm to select object storage devices [40].

PanFS [42] represents another point in the design space. It statically partitions its namespace across metadata managers and randomly places files on different object servers (blades). Like FlexGroups it adjusts its allocation probabilities to reflect disparities in free space across the object servers. PanFS can also actively balance capacity by relocating data objects. Unlike FlexGroups, PanFS does not take load into account during data placement.

Farsite [6] takes a unique approach to metadata partitioning. It spreads files across servers based on file identifiers. But instead of using a hash, it uses a tree-structured system of file identifiers. This supports the colocation of related files, while avoiding the problem of directory renames forcing data migration between servers.

Chunkfs [10] is a single-node file system with dynamic namespace partitioning, but with different goals and implementation. Chunkfs improves fault isolation and recovery by dividing the file system into multiple *chunks*, each containing one or more namespace subtrees that can be checked independently. Chunkfs uses *continuation inodes* to connect subtrees across chunks, similar to remote hardlinks in FlexGroups, except that they are restricted to a single node.

Like Chunkfs, SpanFS [16] stitches together multiple local file systems, called *domains*, into a single volume. SpanFS provides better MP scaling because locks and other resources are local to a single domain, allowing threads in different domains to execute without contention.

7 Conclusion

In this paper, we presented FlexGroup volumes, a distributed version of NetApp FlexVol volumes. FlexGroups achieve seamless scaling across the storage cluster even while simplifying the job of the storage administrator. FlexGroups leverage the maturity, stability, and feature richness of FlexVols. We described the core elements of the design: the infrastructure for remote hardlinks and the ingest heuristics that distribute newly created content. We evaluated FlexGroup performance using both benchmarks and archived customer usage data. The success of FlexGroups has been further validated by rapid customer adoption.

Acknowledgements: We thank the many WAFL engineers who contributed to these designs over the years; they are too many to list. We thank the anonymous reviewers and our the shepherd, Michael Factor, for their helpful comments and advice. We also thank Mike Montour and Robert Franz for their help with performance experiments, and we thank Jessie Wood for copy editing this paper.

References

[1] ANDERSON D.C., CHASE J.C., and VAHDAT A.M. Interposed request routing for scalable network storage. *ACM Transactions on Computer Systems*, *20*(1), pp. 25–48, February 2002.

[2] CORBETT P., ENGLISH B., GOEL A., GRCANAC T., KLEIMAN S., LEONG J., and SANKAR S. Row-diagonal parity for double disk failure correction. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST)*, pp. 1–14. March 2004.

[3] CORBETT P.F. and FEITELSON D.G. The Vesta parallel file system. *ACM Transactions on Computer Systems*, *14*(3), pp. 225–264, August 1996.

[4] COREOS. etcd: A distributed, reliable key-value store for the most critical data of a distributed system. https://coreos.com/etcd/.

[5] CURTIS-MAURY M., DEVADAS V., FANG V., and KULKARNI A. To Waffinity and beyond: A scalable architecture for incremental parallelization of file system code. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 419–434. November 2016.

[6] DOUCEUR J.R. and HOWELL J. Distributed directory service in the Farsite file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 321–334. November 2006.

[7] 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 *Proceedings of the 2008 USENIX Annual Technical Conference*, pp. 129–142. June 2008.

[8] EISLER M., CORBETT P., KAZAR M., and NYDICK D.S. Data ONTAP GX: A scalable storage cluster. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, pp. 139–152. February 2007.

[9] GOEL A. and CORBETT P. RAID triple parity. *ACM SIGOPS Operating Systems Review*, *46*(3), pp. 41–49, 2012.

[10] HENSON V., VAN DE VEN A., GUD A., and BROWN Z. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *Proceedings of the 2nd Conference on Hot Topics in System Dependency (HotDep)*. November 2006.

[11] HITZ D., LAU J., and MALCOLM M. File system design for an NFS file server appliance. In *Proceedings of USENIX Winter 1994 Technical Conference*, pp. 235–246. January 1994.

[12] How GlusterFS distribution works. https://staged-gluster-docs.readthedocs.io/en/release3.7.0beta1/Features/dht/.

[13] HOWARD J.H., KAZAR M.L., MENEES S.G., NICHOLS D.A., SATYANARAYANAN M., SIDEBOTHAM R.N., and WEST M.J. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, *6*(1), pp. 51–81, February 1988.

[14] HPC IO Benchmark Repository. https://github.com/hpc/ior.

[15] HUNT P., KONAR M., JUNQUEIRA F.P., and REED B. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*, pp. 145–158. June 2010.

[16] KANG J., BENLONG, WO T., YU W., DU L., MA S., and HUAE J. SpanFS: A scalable file system on fast storage devices. In *Proceedings of the 2015 USENIX Annual Technical Conference*, pp. 249–261. July 2015.

[17] KESAVAN R., KUMAR H., and BHOWMIK S. WAFL Iron: Repairing live enterprise file systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, pp. 33–47. February 2018.

[18] KESAVAN R., SINGH R., GRUSECKI T., and PATEL Y. Algorithms and data structures for efficient free space reclamation in WAFL. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, pp. 1–13. February 2017.

[19] KESAVAN R., SINGH R., GRUSECKI T., and PATEL Y. Efficient free space reclamation in WAFL. *ACM Transactions on Storage*, *13*(3), September 2017.

[20] KUMAR H., PATEL Y., KESAVAN R., and MAKAM S. High performance metadata integrity protection in the WAFL copy-on-write file system. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, pp. 197–211. February 2017.

[21] LAMPORT L. The part-time parliament. *ACM Transactions on Computer Systems*, *16*(2), pp. 133–169.

[22] LAMPSON B.W. and LOMET D. A new presumed commit optimization for two phase commit. In *Proceedings of the 8th International Conference on Very Large Data Bases (VLDB)*, pp. 630–640. August 1993.

[23] LANCASTER L. and ROWE A. Measuring real world data availability. In *Proceedings of the 15th Systems Administration Conference (LISA)*, pp. 93–100. December 2001.

[24] MCKUSICK M.K., NEVILLE-NEIL G.V., and WATSON R.N.M. *The Design and Implementation of the FreeBSD Operating System*, chapter 10. Addison Wesley, 2nd edition, 2015. ISBN 9780321968975.

[25] NETAPP, INC. Data ONTAP 8. http://www.netapp.com/us/products/platform-os/data-ontap-8/, 2010.

[26] NETAPP, INC. Volume Move Express Guide. https://library.netapp.com/ecm/ecm_download_file/ECMLP2496251, May 2019.

[27] OUSTERHOUT J.K., CHERENSON A.R., DOUGLIS F., NELSON M.N., and WELCH B.B. The Sprite network operating system. *IEEE Computer*, *21*(2), pp. 23–36, February 1988.

[28] PATTERSON D.A., GIBSON G., and KATZ R.H. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 SIGMOD International Conference on Management of Data*, pp. 109–116. June 1988.

[29] PATTERSON H., MANLEY S., FEDERWISCH M., HITZ D., KLEIMAN S., and OWARA S. SnapMirror: File system based asychronous mirroring for disaster recovery. *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, pp. 117–129, January 2002.

[30] RODEH O., BACIK J., and MASON C. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage*, *9*(3), 2013. ISSN 1553-3077. doi:10.1145/2501620.2501623.

[31] SANDBERG R., GOLDBERG D., KLEIMAN S., WALSH D., and LYON B. Design and implementation of the Sun network filesystem. In *Proceedings of the USENIX Summer 1985 Technical Conference*, pp. 119–130. June 1985.

[32] SCHMUCK F. and HASKIN R. GPFS: A shared-disk file system for large computing clusters. In *1st USENIX Conference on File and Storage Technologies (FAST)*, pp. 213–244. January 2002.

[33] SELTZER M.I., GANGER G.R., MCKUSICK M.K., SMITH K.A., SOULES C.A.N., and STEIN C.A. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pp. 71–84. June 2000.

[34] SEVILLA M.A., WATKINS N., MALTZAHN C., NASSI I., BRANDT S.A., WEIL S.A., FARNUM G., and FINEBERG S. Mantle: A programmable metadata load balancer for the Ceph file system. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. November 2015.

[35] STANDARD PERFORMANCE EVALUATION CORPORATION. SPEC SFS 2014. https://www.spec.org/sfs2014/, 2017.

[36] STANDARD PERFORMANCE EVALUATION CORPORATION. SPEC SFS 2014 SP2 Run and Reporting Guide. https://www.spec.org/sfs2014/docs/runrules.pdf, 2017.

[37] STANDARD PERFORMANCE EVALUATION CORPORATION. SPEC SFS 2014 SP2 Users Guide. https://www.spec.org/sfs2014/docs/usersguide.pdf, 2017.

[38] THEKKATH C.A., MANN T., and LEE E.K. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 224–237. October 1997.

[39] WEIL S.A., BRANDT S.A., MILLER E.L., LONG D.D.E., and MALTZAHN C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 307–320. November 2006.

[40] WEIL S.A., BRANDT S.A., MILLER E.L., and MALTZAHN C. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *SC '06: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. November 2006.

[41] WELCH B. and OUSTERHOUT J. Prefix tables: A simple mechanism for locating files in a distributed system. In *Proceedings of the 6th International Conference on Distributed Computing Systems (ICDCS)*, pp. 184–189. May 1986.

[42] WELCH B.B., UNANGST M., ABBASI Z., GIBSON G.A., MUELLER B., SMALL J., ZELENKA J., and ZHOU B. Scalable performance of the Panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, pp. 17–33. February 2008.