Everyone Loves File: File Storage Service (FSS) in Oracle Cloud Infrastructure

Bradley C. Kuszmaul   Matteo Frigo   Justin Mazzola Paluska   Alexander (Sasha) Sandler
Oracle Corporation

Abstract

File Storage Service (FSS) is an elastic filesystem provided as a managed NFS service in Oracle Cloud Infrastructure. Using a pipelined Paxos implementation, we implemented a scalable block store that provides linearizable multipage limited-size transactions. On top of the block store, we built a scalable B-tree that provides linearizable multikey limited-size transactions. By using self-validating B-tree nodes and performing all B-tree housekeeping operations as separate transactions, each key in a B-tree transaction requires only one page in the underlying block transaction. The B-tree holds the filesystem metadata. The filesystem provides snapshots by using versioned key-value pairs. The entire system is programmed using a nonblocking lock-free programming style. The presentation servers maintain no persistent local state, with any state kept in the B-tree, making it easy to scale up and failover the presentation servers. We use a non-scalable Paxos-replicated hash table to store configuration information required to bootstrap the system. The system throughput can be predicted by comparing an estimate of the network bandwidth needed for replication to the network bandwidth provided by the hardware. Latency on an unloaded system is about 4 times higher than a Linux NFS server backed by NVMe, reflecting the cost of replication.

1 Introduction

This paper describes Oracle Cloud Infrastructure File Storage Service (FSS), a managed, multi-tenanted NFS service. FSS, which has been in production for over a year, provides customers with an elastic NFSv3 file service [15]. Customers create filesystems which are initially empty, without specifying how much space they need in advance, and write files on demand. The performance of a filesystem grows with the amount of data stored. We promise customers a convex combination of 100 MB/s of bandwidth and 3000 operations per second for every terabyte stored. Customers can mount a filesystem on an arbitrary number of NFS clients. The size of a file or filesystem is essentially unbounded, limited only by the practical concerns that the NFS protocol cannot cope with files bigger than 16 EiB and that we would need to deploy close to a million hosts to store multiple exabytes. FSS provides the ability to take a snapshot of a filesystem using copy-on-write techniques. Creating a filesystem or snapshot is cheap, so that customers can create thousands of filesystems, each with thousands of snapshots. The system is robust against failures since it synchronously replicates data and metadata 5-ways using Paxos [44].

We built FSS from scratch. We implemented a Paxos-replicated block store, called DASD, with a sophisticated multipage transaction scheme. On top of DASD, we built a scalable B-tree with multikey transactions programmed in a lockless nonblocking fashion. Like virtually every B-tree in the world, ours is a B+-tree. We store the contents of files directly in DASD and store file metadata (such as inodes and directories) in the B-tree.

Why not do something simpler? One could imagine setting up a fleet of ZFS appliances. Each appliance would be responsible for some filesystems, and we could use a replicated block device to achieve reliability in the face of hardware failure. Examples of replicated block devices include [2, 4, 25, 54, 61]. We have such a service in our cloud, so why not use it? It's actually more complicated to operate such a system than a system that's designed from the beginning to operate as a cloud service. Here are some of the problems you would need to solve:

- How do you grow such a filesystem if it gets too big to fit on one appliance?
- How do you partition the filesystems onto the appliance? What happens if you put several small filesystems onto one appliance and then one of the filesystems grows so that something must move?

- How do you provide scalable bandwidth? If a customer has a petabyte of data they should get 100 GB/s of bandwidth into the filesystem, but a single appliance may have only a 10 Gbit/s network interface (or perhaps two 25 Gbit/s network interfaces).
- How do you handle failures? If an appliance crashes, then some other appliance must mount the replicated block device, and you must ensure that the original appliance doesn't restart and continue to perform writes on the block device, which would corrupt the filesystem.

This paper describes our implementation. Section 2 provides an architectural overview of FSS. The paper then proceeds to explain the system from the top down. Section 3 describes the lock-free nonblocking programming style we used based on limited-size multipage transactions. Section 4 shows how we organize metadata in the B-tree. Section 5 explains how we implemented a B-tree key-value store that supports multikey transactions. Section 6 explains DASD, our scalable replicated block storage system. Section 7 describes our pipelined Paxos implementation. Section 8 discusses congestion management and transaction-conflict avoidance. Section 9 describes the performance of our system. Sections 10 and 11 conclude with a discussion of related work and a brief history of our system.

2 FSS Architecture

This section explains the overall organization of FSS. We provision many hosts, some of which act as storage hosts, and some as presentation hosts. The storage hosts, which include local NVMe solid-state-drive storage, store all filesystem data and metadata replicated 5-ways,[1] and provide an RPC service using our internal FSS protocol. The presentation hosts speak the standard NFS protocol and translate NFS into the FSS protocol.

A customer's filesystems appear as exported filesystems on one or more IP addresses, called ***mount targets***. A single mount target may export several filesystems, and a filesystem may be exported by several mount targets. A mount target appears as a private IP address in the customer's virtual cloud network (VCN), which is a customizable private network within the cloud. Most clouds provide VCNs in which hosts attached to one VCN cannot even name hosts in another VCN. Each mount target terminates on one of our presentation hosts. A single mount target's performance can be limited by the network interface of the presentation host, and so to get more performance, customers can create many mount targets that export the same filesystem.

[1]Data is erasure coded, reducing the cost to 2.5, see Section 3.

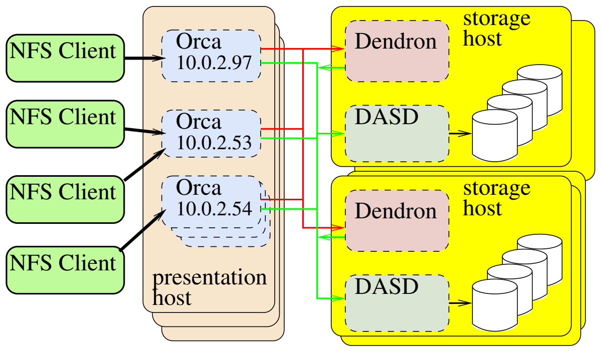

Figure 1: FSS architecture. The NFS clients are on the left, and belong to various customers. Hosts are shown as boxes with solid edges and processes are shown with dashed lines. The presentation hosts are in the middle, each running several Orca processes. The Orca processes are connected to the various customer virtual cloud networks (VCNs) on the left. The IP addresses of each Orca's mount target is shown. The Orca process are also connected to our internal VCN, where they can communicate with the storage hosts. The storage hosts contain NVMe drives and run both the Dendron and DASD processes.

Figure 1 shows how the FSS hosts and processes are organized. The customer sets up NFS clients in their VCN. Our presentation hosts terminate NFS connections from the clients in per-mount-target Orca processes. The Orca processes translate NFS requests into the FSS protocol, and send the FSS to our storage hosts. In the future, the presentation hosts might speak other client protocols, such as SMB [55] or NFSv4 [68].

To ensure isolation between filesystems we depend on a combination of process isolation on our servers, VCN isolation, and encryption. All data stored in the storage hosts or in flight in the FSS protocol is encrypted with a file-specific encryption key that derives from a filesystem master key. The NFSv3 protocol is not encrypted, however, so data arriving at an Orca is potentially vulnerable. To mitigate that vulnerabilty, we rely on VCN isolation while the data is in flight from the NFS client to the presentation host, and use the presentation host's process isolation to protect the data on the presentation host. All data and file names are encrypted as soon as they arrive at an Orca, and each Orca process serves only one mount target.

Each storage host contains NVMe drives and runs two processes, DASD and Dendron. ***DASD***, described in Section 6, provides a scalable block store. ***Dendron*** implements a B-tree (Section 5) in which it maintains the metadata (Section 4) for the filesystem.

We chose to replicate filesystems within a data center rather than across data centers within a metropolitan area or across a long distance. There is a tradeoff between latency and failure tolerance. Longer-distance replication means the ability to tolerate bigger disasters, but incurs longer network latencies. We chose local replication so that all of our operations can be synchronously replicated by Paxos without incurring the latency of long-distance replication. It turns out that most of our customers rely on having a functional disaster-recovery plan, and so they're more interested in single-data center file system performance than synchronous replication. In the future, however, we may configure some filesystems to be replicated more widely.

Within a data center, hosts are partitioned into groups called ***fault domains***. We typically employ 9 fault domains. In a small data center, a fault domain might be a single rack. In a large data center, it might be a group of racks. Hosts within a fault domain are likely to fail at the same time (because they share a power supply or network switch). Hosts in different fault domains are more likely to fail independently. We employ 5-way Paxos replicated storage that requires at least 3 out of each group of 5 Paxos instances in order to access the filesystems. We place the Paxos instances into different fault domains. When we need to upgrade our hosts, we can bring down one fault domain at a time without compromising availability. Why 5-way replication? During an upgrade, one replica at a time is down. During that time, we want to be resilient to another host crashing.

We also use the same 5-way-replicated Paxos machinery to run a non-scalable hash table that keeps track of configuration information, such a list of all the presentation hosts, needed for bootstrapping the system.

All state (including NLM locks, leases, and idempotency tokens) needed by the presentation servers is maintained in replicated storage rather than in the memory of the presentation hosts. That means that any Orca can handle any NFS request for the filesystems that it exports. The view of the filesystem presented by different Orcas is consistent.

All memory and disk space is allocated when the host starts. We never run `malloc()` after startup. By construction, the system cannot run out of memory at runtime. It would likely be difficult to retrofit this memory-allocation discipline into old code, but maintaining the discipline was relatively straightforward since the entire codebase is new.

3 Multi-Page Store Conditional

FSS is implemented on top of a distributed B-tree, which is written on top of a a distributed block store with multi-page transactions (see Figure 2). This section describes the programming interface to the distributed block store and how the block store is organized into pages, blocks, and extents.

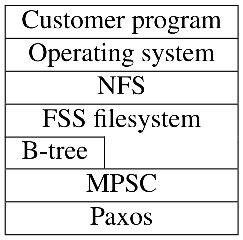

Figure 2: Each module is built on the modules below.

The filesystem is a concurrent data structure that must not be corrupted by conflicting operations. There can be many concurrent NFS calls modifying a filesystem: one might be appending to a file, while another might be deleting the file. The filesystem maintains many invariants. One important invariant is that every allocated data block is listed in the metadata for exactly one file. We need to avoid memory leaks (in which an allocated block appears in no file), dangling pointers (in which a file contains a deallocated block), and double allocations (in which a block appears in two different files). There are many other invariants for the filesystem. We also employ a B-tree which has its own invariants. We live under the further constraint that when programming these data structures, we cannot acquire a lock to protect these data structures, since if a process acquired a lock and then crashed it would be tricky to release the lock.

To solve these problems we implemented FSS using a nonblocking programming style similar to that of transactional memory [32]. We use a primitive that we call ***multi-page store-conditional (MPSC)*** for accessing pages in a distributed block store. An MPSC operation is a "mini-transaction" that performs an atomic read-and-update of up to 15 pages. All page reads and writes follow this protocol:

1. Read up to 15 pages, receiving the page data and a ***slot number*** (which is a form of a version tag [38]). A page's slot number changes whenever the page changes. You can read some pages before deciding which page to read next, or you can read pages in parallel. Each read is linearizable [34].
2. Compute a set of new values for those pages.
3. Present the new page values, along with the previously obtained slot numbers, to the MPSC function. To write a page requires needs a slot number from a previous read.
4. The update will either succeed or fail. Success means that all of the pages were modified to the new values and that none of the pages had been otherwise modified since they were read. A successful

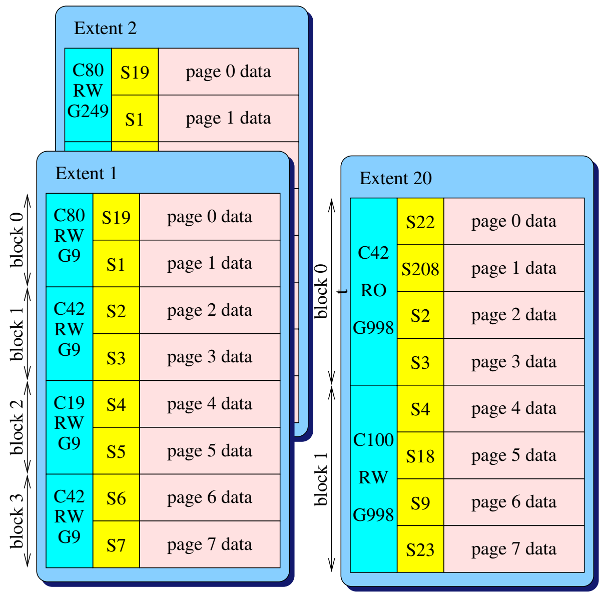

Figure 3: Pages, blocks, and extents. Three extents are shown, each with an array of pages. Each page has a slot. E.g., page 0 of extent 1 has slot 19. Each block has ownership. The first block of extent 1 is owned by customer 80 ("C80"), is read-write ("RW"), and is on its 9th allocation generation ("G9"). Extents 1 and 2 each have 2 pages per block and 4 blocks, whereas extent 20 has 4 pages per block and only 2 blocks.

update linearizes with other reads and MPSC updates. A failure results in no changes.

In addition to reading and writing pages, an MPSC can allocate or free space in the distributed block store.

An MPSC could fail for many reasons. For example, if, between reading a page and attempting an MPSC, some other transaction wrote the page, the MPSC will fail. Even if there is no conflict, an MPSC may fail due to, e.g., packet loss or Paxos leadership changes. Even if a transaction succeeds, the caller may receive an error indication, e.g., if network fails between the update's commit and the caller notification. Our implementation deliberately introduces failures (sometimes called fuzzing [56]) with a small probability rate, so that all of the error-handling code paths are exercised frequently, even in production.

Pages and Blocks. We subdivide the distributed block store into a hierarchy of pages, blocks, and extents, as shown in Figure 3. An MPSC performs an atomic update on a set of ***pages***. A ***block*** includes one or more pages, and is the unit on which we do bookkeeping for allocation. To reduce bookkeeping overheads on small pages, we allocate relatively large blocks. To keep transactions small, we update relatively small pages. An ***extent*** is an array of pages, up to 256 GiB total, and is implemented by a replicated Paxos state machine.

Geometry	Page size	Block size	Extent size	RF	EC
B-tree	8 KiB	1 MiB	16 GiB	5	1
8 KiB	8 KiB	8 KiB	32 GiB	5	5:2
32 KiB	32 KiB	32 KiB	128 GiB	5	5:2
256 KiB	32 KiB	256 KiB	256 GiB	5	5:2
2 MiB	32 KiB	2 MiB	256 GiB	5	5:2

Figure 4: Extent geometries. The B-tree extents contain metadata organized as a B-tree. The other extents contain file contents, and are identified by their block size. For each extent the page size, block size, extent size, replication factor (RF), and erasure-coding (EC) are shown.

For example, one kind of extent contains 256 GiB of disk-resident data, organized in 2 MiB blocks with 32 KiB pages, and is replicated 5 ways using 5:2 erasure coding (an erasure-coding rate of 2/5) [62]. Thus the 256 GiB of disk-resident data consumes a total of 640 GiB of disk distributed across 5 hosts.

An extent's ***geometry*** is defined by its page size, block size, extent size, replication factor, and erasure-coding rate. Once an extent is created, its geometry cannot change. Figure 4 shows the extent geometries that we use for file data and metadata. All of our extents are 5-way replicated within a single data center. The pages in extents used for file contents are erasure coded using a 5:2 erasure coding rate, so that the overhead of storing a page is 2.5 (each replica stores half a page, and there are 5 replicas). The B-tree data is mirrored, which can be thought of as 5:1 erasure coding.

We size our extents so there are hundreds of extents per storage host to ease load balancing. We use parallelism to recover the missing shards when a host crashes permanently—each extent can recover onto a different host.

Block ownership. When operating a storage system as a service, it is a great sin to lose a customer's data. It is an even greater sin to give a customer's data to someone else, however. To avoid the greater sin, blocks have ownership information that is checked on every access.

A block's ownership information includes a version tag, called its ***generation***, as well as 64-bit customer identifier, and a read-only bit. When accessing a block, the generation, customer id, and read-only bit must match exactly. This check is performed atomically with every page access. When a block is allocated or deallocated its generation changes. A ***tagged pointer*** to a page includes the block ownership information, as well as the extent number and page number. A block's pointer is simply the tagged pointer to the block's first page.

The problem that block ownership solves can be illustrated as follows. When data is being written into a new file, we allocate a block and store the block's pointer in the B-tree as a single transaction. To read data from a file, Orca first obtains the block pointer by asking Dendron to read the B-tree. Orca caches that block pointer, so that it can read the data without the overhead of checking the B-tree again on every page access. Meanwhile, another thread could truncate the file, causing the block to be deallocated. The block might then be allocated to a file belonging to a different customer. We want to invalidate Orca's cached pointers in this situation, so we change the block ownership. When Orca tries to use a cached pointer to read a deallocated page, the ownership information has become invalid, and the access fails, which is what we want.

Each of our read operations is linearizable, meaning that they are totally ordered with respect to all MPSC operations and the total ordering is consistent with real time. Although our read operations linearize, if you perform several reads they take place at different times, meaning that the reads may not be mutually consistent. It's easy to trick a transactional-memory-style program into crashing, e.g., due to a failed assertion. For example, if you have two pages in a doubly linked list, you might read one page, and then follow a pointer to the second page, but by the time you read the second page it no longer points back to the first page. Getting this right everywhere is an implementation challenge, leading some [10, 16] to argue that humans should not program transactional memory without a compiler. We have found this problem to be manageable, however, since an inconsistent read cannot lead to a successful MPSC operation, so the data structure isn't corrupted.

4 A Filesystem Schema

This section explains how we represent the filesystem metadata in our B-tree. FSS implements an inode-based write-in-place filesystem using a single B-tree to hold its metadata. What does that mean? "Inode-based" means that each file object has an identifier, called its ***handle***. The handle is used as an index to find the metadata for a file. "Write-in-place" means that updates to data and metadata usually modify an existing block of data. (As we shall see, snapshots introduce copy-on-write behavior.) "Single B-tree to hold the metadata" means there is only one B-tree per data center. Our service provides many filesystems to many customers, and they are all stored together in one B-tree.

The B-tree must support various metadata operations. For example, given an object's handle, we need to find and update the fixed-size part of the object's metadata, which includes the type of the object (e.g., regular, directory, symlink), permissions bits (`rwxrwxrwx`), owner, group, file size, link count, and timestamps. Given a file handle and an offset, we need to find the tagged pointer of the block holding data at that offset, so that reads or write can execute. Given a directory handle and a filename we need to be able perform a directory lookup, yielding a file handle. For a directory, we need to iterate through the directory entries. In NFS this is performed using a 64-bit number called a ***cookie***. Given a directory handle and a cookie we need to find the directory entry with the next largest cookie.

Key-value pairs:

leaderblock:	$0 \rightarrow$ next F.
superblock:	$F, 0 \rightarrow$ next D, next C, keys.
inode:	$F, 1, D, C, 2, S \rightarrow$ stat-data.
name map:	$F, 1, D, C = 0, 3, N, S \rightarrow F, D', C', S$.
cookie map:	$F, 1, D, C = 0, 4, c, S \rightarrow F, D', C', S, N$.
block map:	$F, 1, D, C, 5, o, S \rightarrow$ block ID and size.

Glossary:

F	filesystem number.
D	Directory unique id.
C	File unique id.
S	Snapshot number.
o	Offset in file.
N	Filename in directory.
c	Directory iteration cookie.
F, D', C', S	The handle of a file in a directory.

Figure 5: Filesystem schema showing key $\rightarrow$ value pair mappings. The small numbers (e.g., "1") are literal numbers inserted between components of a key to disambiguate key types and force proper B-tree sort ordering. For directories, $C = 0$.

Our strategy is to create B-tree key-value pairs that make those operations efficient. We also want to minimize the number of pages and extents that we access in each transaction. Every B-tree key is prefixed with a filesystem number F, so that all the keys for a given filesystem will be adjacent in the B-tree. Our handles are ordered tuples of integers $\langle F, D, C, S \rangle$, where D a unique number for every directory, C is a unique number for every file ($C = 0$ for directories), and S is a snapshot number. A file's handle depends on the directory in which it was created. The file can be moved to another directory after it is created, but the file's handle will always mention the directory in which the file was originally created.

Figure 5 shows the schema for representing our filesystems. We encode the B-tree keys in a way that disambiguates the different kinds of key value pairs and sorts the key-value pairs in a convenient order. For example, all the pairs for a given filesystem F appear together, with the superblock appearing first because of the "0" in its key. Within the filesystem, all the non-superblock pairs are sorted by D, the directory number. For a directory, the directory inode sorts first, then come the name map entries for the directory, and then the cookie map, then come all the inodes for the files that were created in that directory. In that set for each file, the file inode sorts first, followed by the block maps for that file. Finally two entries that are the same except for the snapshot number are sorted by snapshot number.

We implement snapshots using copy-on-write at the key-value pair level, rather than doing copy-on-write in the B-tree data structure or at the block level [13, 28, 35, 43, 49, 63, 65, 66, 72]. In the interest of space, we don't show all the details for snapshots, but the basic idea is that each key-value pair is valid for a range of snapshots. When looking up a pair for snapshot S, we find the pair whose key has the largest snapshot number that's no bigger than S.

Our key-value scheme achieves locality in the B-tree. When a file is created it is lexicographically near its parent directory, and the file's block maps and fixed-sized metadata are near each other. (If the file is later moved, it still appears near the original parent directory.) This means that if you create a file in a directory that has only a few files in it, it's likely that the whole transaction to update the directory inode, add directory entries, and create the file inode will all be on the same page, or at least in the same extent, since the B-tree maintains maintains block as well as page locality (see Section 5).

We use multiple block sizes (which are shown in Figure 4) to address the tension between fragmentation and metadata overhead. Small blocks keep fragmentation low for small files. Big blocks reduce the number of block map entries and other bookkeeping overhead for big files. In our scheme the first few blocks of a file are small, and as the file grows the blocks get bigger. For files larger than 16 KiB, the largest block is no bigger than $1/3$ the file size, so that even if the block is nearly empty, we have wasted no more than $1/3$ of our storage. We sometimes skip small-block allocation entirely. For example if the NFS client writes 1 MiB into a newly created file, we can use 256 KiB blocks right away.

5 The B-tree

To hold metadata we built a B-tree [7] on top of MPSC. MPSC provides transactions that can update up to 15 pages, but we want to think about key-value pairs, not pages, and we want B-tree transactions to be able include non-B-tree pages and blocks, e.g., to allocate a data block and store its tagged pointer in a B-tree key-value pair. The B-tree can perform transactions on a total of 15 values, where a value can be a key-value pair, or a non-B-tree page write or block allocation.

Consider the simple problem of executing a B-tree transaction to update a single key-value pair. How many pages must be included in that transaction? The standard B-tree algorithm starts at the root of the tree and follows pointers down to a leaf page where it can access the pair. To update the leaf we need to know that it is the proper page, and the way we know that is by having followed a pointer from the leaf's parent. Between reading the parent and reading the leaf, however, the tree might have been rebalanced, and so we might be reading the wrong leaf. So we we need to include the parent in the transaction. Similarly, we need to include the grandparent and all the ancestors up to the root. A typical B-tree might be 5 or 6 levels deep, and so a single key-value pair update transaction involves 5 or 6 pages, which would limit us to 2 or 3 key-value pairs per transaction. Furthermore, every transaction ends up including the root of the B-tree, creating a scalability bottleneck.

Our solution to this problem is to use ***self-validating pages***, which contain enough information that we can determine if we read the right page by looking at that page in isolation. We arrange every page to "own" a range of keys, for the page to contain only keys in that range, and that every possible key is owned by exactly one page. To implement this self validation, we store in every page a lower bound and upper bound for the set of keys that can appear in the page (sometimes called "fence keys" [26, 47]), and we store the height of the page (leaf pages are height 0). When we read a page to look up a key, we verify that the page we read owns the key and is the right height, in which case we know that if that page is updated in a successful transaction, that we were updating the right page. Thus, we do not need to include the intermediate pages in the MPSC operation and we can perform B-tree transactions on up to 15 keys.

We usually skip accessing the intermediate B-tree nodes altogether by maintaining a cache that maps keys to pages. If the cache steers us to a wrong page, either the page won't self validate or the transaction will fail, in which case we simply invalidate the cache and try again. If a key is missing from the cache, we can perform a separate transaction that walks the tree to populate the cache. It turns out that this cache is very effective, and for virtually all updates we can simply go directly to the proper page to access a key-value pair.

Another problem that could conceivably increase the transaction size is tree rebalancing. In a B-tree, tree nodes must generally be split when they get too full or merged when they get too empty. The usual rule is that whenever one inserts a pair into a node and it doesn't fit, one first splits the node and updates the parent (possibly triggering a parent split that updates the grandparent, and so on). Whenever one deletes a pair, if the node becomes too empty (say less than $1/4$ full), one

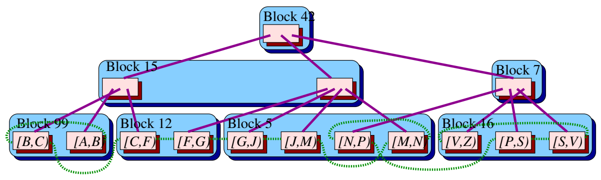

Figure 6: The B-tree comprises block (in blue) and pages (in pink). The pages form a tree. The leaf pages, where the key-value pairs are stored, form a doubly linked list (shown with dashed green lines). Each leaf page is responsible for a range of keys, e.g., $[C,F)$ means the keys from C inclusive to F exclusive. Each block holds a key range of pages for one level. For example, Block 5 has all the leaf pages in the range $[G,P)$.

merges nodes, updating the parent (which can possibly trigger a parent merge that updates the grandparent, and so on). This means that any insertion or deletion can add as many pages to a transaction as the height of the tree. Those rebalancings are infrequent so they don't introduce a scalability bottleneck, but they do make our MPSC operations too big.

Our solution to the rebalancing problem is to rebalance in a separate transaction. When inserting if we encounter a page overflow, we abort the transaction, split the page in a separate transaction, and restart the original transaction. We split the page even if it is apparently nearly empty: as long as there are two keys we can split the page. For merges, we delete keys from the page, and then do a separate transaction afterward to rebalance the tree. It's possible that a page could end up empty, or nearly empty, and that due to some crash or packet loss, we forget to rebalance the tree. That's OK because we fix it up the next time we access the page.

To improve locality we exploit both the page and block structure of MPSC. Figure 6 shows how the B-tree is organized to exploit block locality as well as page locality. Each page is responsible for a key range, and the union of the key ranges in a block is a single key range. When splitting a page, we place the new page into the same block as the old page, and if the block is full, we insert a new block. If the B-tree schema strives to keep keys that will appear in the same transaction lexicographically near each other, locality causes those keys to likely be in the same page, or at least the same block. Our MPSC implementation optimizes for the case where some pages of a transaction are in the same extent. With the schema described in Section 4, this optimization is worth about a 20% performance improvement for an operation such as untarring a large tarball.

The choice of 15 pages per transaction is driven by the B-tree implementation. There is one infrequent operation requiring 15 pages. It involves splitting a page in a full block: a new block is allocated, block headers are updated, and the pages are moved between blocks. Most transactions touch only one or two pages.

6 DASD: Not Your Parent's Disk Drive

This section explains how we implemented MPSC using Paxos state machines (which we discuss further in Section 7). MPSC is implemented by a distributed block store, called ***DASD***[2]. A single extent is implemented by a Paxos state machine, so multipage transactions within an extent is straightforward. To implement transactions that cross extents, we use 2-phase commit.

Given that Paxos has provided us with a collection of replicated state machines, each with an attached disk, each implementing one extent, we implement two-phase commit [29,46,50] on top of Paxos. The standard problem with two-phase commit is that the transaction coordinator can fail and the system gets stuck. Our extents are replicated, so we view the participants in a transaction as being unstoppable.

It would be easy to implement two-phase commit with $3n$ messages. One could send n 'prepare' messages that set up the pages, then n 'decide' messages that switch the state to commited, and then n 'release' messages that release the resources of the transaction. (Each message drives a state transition, which is replicated by Paxos.) The challenge is to implement two-phase commit on n extents with only $2n$ messages and state changes. Every filesystem operation would benefit from the latency being reduced by $1/3$.

To perform an atomic operation with only $2n$ messages, for example on 3 pages, the system progresses through the states shown in Figure 7. The system will end up constructing and tearing down, in the Paxos state machine, a doubly-linked (not circular) list of all the extents in the transaction. Each of these steps is initiated by a message from a client, which triggers a state change in one Paxos state machine (which in turn requires several messages to form a consensus among the Paxos replicas). The client waits for an acknowledgment before sending the next message.

1. Extent A receives a prepare message. A enters the prepared state, indicated by "P(data)", and records its part of the transaction data and its part of the linked list (a null back pointer, and a pointer to B).
2. Extent B receives a prepare message, enters the prepared state, and records its data and pointers to A and C.
3. Extent C receives a prepare-and-decide message, enters the decided state (committing the transac-

[2]Direct-Access Storage Device (DASD) was once IBM's terminology for disk drives [37].

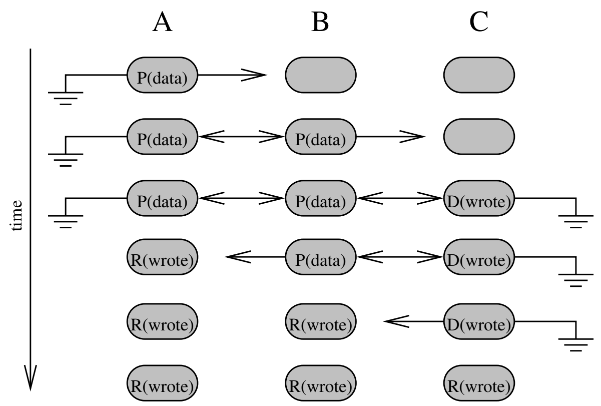

Figure 7: Three extents performing a DASD transaction. Each column is an extent, and each row is a point in time, with time moving downward. A double-arrowed line shows two extents pointing at each other. A single-arrowed line shows one extent pointing at the other, with no pointer back. Ground represents a null pointer.

tion), and records its data and the back pointer to *B*, indicated by "D(wrote)".

4. Extent *A* receives a decide-and-release message, notes that the transaction is committed, and releases its resources (such as memory) associated with the transaction, indicated by "R(wrote)". The head of the linked list is now gone.
5. Extent *B* receives a decide-and-release message, notes the commit, and releases its resources.
6. Extent *C* receives a release message (it had already decided) and releases its resources.

Thus we implement two-phase commit in exactly $2n$ messages with $2n$ state transitions. Note that the final transition of state *C* doesn't actually need to be done before replying to the client, and could be piggybacked into the prepare step of the next transaction, potentially reducing the latency to $2n-1$ messages.

The system maintains the invariant that either a prefix or a suffix of the linked list exists, which is useful if the transaction is interrupted. There are two ways that the system can be interrupted, before the commit (in which case the transaction will abort), or after (in which case the cleanup is incomplete). The prefix-suffix property helps in both of these cases. If the transaction gets aborted (at or before step 3) then a prefix exists. If we encounter a prepared state, we can follow the linked list forward until we either find a dangling pointer or a decided state. If we find a dangling pointer, we can delete the prepare record that contained the dangling pointer, preserving a prefix. (At the final point, *C*, of the linked list, we must extract a promise that *C* will never decide that the transaction commits. This can be accomplished by introducing a conflict on the read slot for the page.) If we find a decided state then the cleanup was interrupted, so it can proceed back along the linked list until we find the beginning or a dangling pointer, and move the state forward to released.

Our scheme relies on the fact that each state transition occurs one after the other, and hence the critical path of the transition is also $2n$ messages. There are schemes in which one can move the states forward in parallel. For example, one could broadcast "prepare" messages to all the extents, then have one extent decide, and then broadcast decide messages to them all, then release messages, so that the critical path would be only 4 long. This results in $3n$ state transitions (minus one or two, depending on how clever you are.) If you think that big transactions are common, that's valuable, but we have found that most transactions are short so it's better to do the transaction serially.

We optimize the case when there are several pages in a single extent to use fewer messages.

7 Pipelined Paxos

In this section we explain our Paxos implementation, and in particular how we pipeline Paxos operations.

Lamport-Paxos [44, 45] is an algorithm to achieve consensus on a single value. Lamport-Paxos requires two phases, called phase 1 and phase 2 by Lamport.

To achieve consensus on a log (as opposed to one value), one common algorithm is Multi-Paxos [17], which treats the log as an array indexed by slot, running Lamport-Paxos independently on each array element. It turns out that you can run a "vector" phase 1 for infinitely many elements of the array with a single pair of messages, and that you can reuse the outcome of phase 1 for as many phase 2 rounds as you want. In this setup, people tend to call phase-1 "master election" and infer all sorts of wrong conclusions, e.g. that there is only one master at any time and that phase 1 is some kind of "failover".

In Multi-Paxos, if the operation on slot $S+1$ depends on the state after slot S, you must wait for slot S (and all previous slots) to finish phase 2. (We don't say "commit" to avoid confusion with two-phase commit, which is a different protocol.) This Multi-Paxos is not pipelined.

You can pipeline Multi-Paxos with a small modification. You tag each log entry with a unique log sequence number (LSN) and you modify Paxos so that an acceptor accepts slot $S+1$ only if it agrees on the LSN of slot S. Thus, the Paxos phase 2 message is the Lamport phase 2 plus the LSN of the previous slot. By induction, two acceptors that agree on a LSN agree on the entire past history.

Now you can issue phase 2 for $S+1$ depending on S without waiting for S to complete, because the acceptance of $S+1$ retroactively confirms all speculations that you made.

The pipelined Multi-Paxos state, per slot, is the Lamport-Paxos state (a ***ballot*** B and the slot's contents) plus the LSN. You can use whatever you want as LSNs, as long as they are unique, but a convenient way to generate LSNs is to use the pair $\langle E, S \rangle$ where the ***epoch*** E must be unique. As it happens, Lamport phase 1 designates a single winner of ballot B, so you can identify E with the winner of ballot B in phase 1, and be guaranteed that nobody else wins that ballot. In the $E = B$ case, you can reduce the per-slot state to the single-value E, with the dual-role of LSN for pipelining and of ballot for Lamport-Paxos.

Our Paxos algorithm is almost isomorphic to Raft [60]. Essentially Raft is Multi-Paxos plus conditional LSNs plus $E = B$. However, Raft always requires an extra log entry in order to make progress, and cannot be done in bounded space. If you recognize that you are just doing good-old Paxos, then you can make progress by storing a separate ballot B in constant space.

The idea of the acceptance conditional on the previous LSN appeared in viewstamped replication [58] (which didn't discuss pipelining). It is used specifically for pipelining in Zookeeper, except that Zookeeper tries to reinvent Paxos, but incorrectly assumes TCP is an ideal pipe [6]. Conditional acceptance is also used in Raft in the same way as in viewstamped replication, except that Raft lost the distinction between proposer and acceptor, which prevents it from having a speculative proposer state that runs ahead of acceptors.

Recovery. Here we explain how our Paxos system recovers from failures.

The on-disk state of a Paxos acceptor has two main components: the log (of bounded size, a few MB), and a large set of page shards (tens of GB). A shard comprises an erasure-coded fragment of a page and some header information such as a checksum. To write a shard, the Paxos proposer appends the write command to the log of multiple acceptors. When a quorum of acceptors has accepted the command, the write is considered done (or "learned" in Paxos terminology). The proposer then informs acceptors that a command has been learned, and acceptors write the erasure-coded shard to disk.

As long as all acceptors receive all log entries, this process guarantees that all acceptors have an up-to-date and consistent set of shards. However, acceptors may temporarily disappear for long enough that the only way for the acceptor to make progress is to incur a log discontinuity. We now must somehow rewrite all shards modified by the log entries that the acceptor has missed, a process called recovery.

The worst-case for recovery is when we must rewrite the entire set of shards, for example because we are adding a new acceptor that is completely empty. In this ***long-term*** recovery, as part of their on-disk state, acceptors maintain a range of pages that need to be recovered, and they send this ***recovery state*** back to the proposer. The proposer iterates over such pages and overwrites them by issuing a Paxos read followed by a conditional Paxos write, where the condition is on the page still being the same since the read. When receiving a write, the acceptor subtracts the written page range from the to-be-recovered page range, and sends the updated range back to the proposer.

Long-term recovery overwrites the entire extent. For discontinuities of short duration, we use a less expensive mechanism called ***short-term recovery***. In addition to the long-term page range, acceptors maintain a range of log slots that they have lost, they update this range when incurring a discontinuity, and communicate back this slot range to the proposer. The proposer, in the Paxos state machine, maintains a small pseudo-LRU cache of identifiers of pages that were written recently, indexed by slot. If the to-be-recovered slot range is a subset of the slot range covered by the cache, then the proposer issues all the writes in the slot range, in slot order, along with a range R whose meaning is that the present write is the only write that occurred in slot range R. When receiving the write, the acceptor subtracts R from its to-be-recovered slot range and the process continues until the range is empty. If the to-be-recovered slot range overflows the range of the cache, the acceptor falls into long-term recovery.

In practice, almost all normal operations (e.g., software deployments) and unscheduled events (e.g., power loss, network disruption) are resolved by short-term recovery. We need long-term recovery when loading a fresh replica, and (infrequently) when a host goes down for a long time.

Checkpointing and logging. Multi-Paxos is all about attaining consensus on a log, and then we apply that log to a state machine. All memory and disk space in FSS is statically allocated, and so the logs are of a fixed size. The challenge is to checkpoint the state machine so that we can trim old log entries. The simplest strategy is to treat the log as a circular buffer and to periodically write the entire state machine into the log. Although for DASD extents, the state machine is only a few MB, some of our other replicated state machines are much larger. For example we use a 5-way replicated hash table, called Minsk, to store configuration information for bootstrapping the system: given the identity of the five Minsk instances, a Dendron instance can determine the identity of all the other Dendron instances. If the Paxos state machine is large, then checkpointing the state machine all at once causes a performance glitch.

Here's a simple scheme to deamortize checkpointing. Think of the state machine as an array of bytes, and every operation modifies a byte. Now, every time we log an update to a byte, we also pick another byte from the hash table and log its current value. We cycle through all the bytes of the table. Thus, if the table is K bytes in size, after K update operations we will have logged every byte in the hash table, and so the most recent $2K$ log entries have enough information to reconstruct the current state of the hash table. We don't need to store the state machine anywhere, since the log contains everything we need.

This game can be played at a higher level of abstraction. For example, suppose we think of the hash table as an abstract data structure with a `hash_put` operation that is logged as a logical operation rather than as operations on bytes. In that case every time we log a `hash_put` we also log the current value of one of the hash table entries, and take care to cycle through all the entries. If the hash table contains K key-value pairs, then the entire hash table will be reconstructable using only the most recent $2K$ log entries. This trick works for a binary tree too.

8 Avoiding Conflicts

This section outlines three issues related to transaction conflicts: avoiding too much retry work, avoiding congestion collapse, and reducing conflicts by serializing transactions that are likely to conflict.

In a distributed system one must handle errors in a disciplined fashion. The most common error is when a transaction is aborted because it conflicts with another transaction. Retrying transactions at several different places in the call stack can cause an exponential amount of retrying. Our strategy is that the storage host does not retry transactions that fail. Instead it attempts to complete one transaction, and if it fails the error is returned all the way back to Orca which can decide whether to retry. Orca typically sets a 55 s deadline for each NFS operation, and sets a 1 s deadline for each MPSC. Since the NFS client will retry its operation after 60 s, it's OK for Orca to give up after 55 s.

In order to avoid performance collapse, Orca employs a congestion control system similar to TCP's window-size management algorithm [71]. Some errors, such as transaction conflicts, impute congestion. In some situations the request transaction did not complete because some "housekeeping" operation needed to be run first (such as to rebalance two nodes of the B-tree). Doing the housekeeping uses up the budget for a single transaction, so an error must returned to Orca, but in this case the error does not impute congestion.

When two transactions conflict, one aborts, which is inefficient. We use in-memory locking to serialize transactions that are likely to conflict. For example, when Orca makes an FSS call to access an inode, it sends the request to the storage host that is likely to be the Paxos leader for the extent where that inode is stored. That storage host then acquires an in-memory lock so that two concurrent calls accessing the same inode will run one after another. Orca maintains caches that map key ranges to extent numbers and extent numbers to the leader's IP address. Sometimes one of the caches is wrong, in which case, as a side effect of running the transaction, the storage host will learn the correct cache entries, and inform Orca, which will update its cache. The in-memory lock is used for performance and is not needed for correctness. The technique of serializing transactions that are likely to conflict is well known in the transactional-memory literature [48, 73].

9 Performance

In the introduction we promised customers a convex combination of 100MB/s of bandwidth and 3000 IOPS for every terabyte stored. Those numbers are throughput numbers, and to achieve those numbers the NFS clients may need to perform operations in parallel. This section first explains where those throughput numbers come from, and then discusses FSS latency.

In order to make concrete throughput statements, we posit a simplified model in which the network bandwidth determines performance. The network bottleneck turns out to be on the storage hosts. If VNICs on the NFS clients are the bottleneck, then the customer can add NFS clients. If the presentation host is the bottleneck, then additional mount targets can be provisioned. The performance of the NVMe is fast compared to the several round trips required by Paxos (in contrast to, e.g., Gaios, which needed to optimize for disk latency instead of network latency because disks were so slow [11]).

We define performance in terms of the ratio of bandwidth to storage capacity. There are four components to the performance calculation: raw performance, replication, scheduling, and oversubscription. The ***raw performance*** is the network bandwidth divided by the disk capacity, without accounting for replication, erasure coding, scheduling, or oversubscription.

Replication consumes both bandwidth and storage capacity. Using 5:2 erasure coding, for each page of data, half a page is stored in each of five hosts. This means we can sell only 40% of the raw storage capacity. The network bandwidth calculation is slightly different for writes and reads. For writes, each page must be received by 5 different storage hosts running Paxos. That data is erasure-coded by each host then written to disk. Thus, for writes, replication reduces the raw network bandwidth by a factor of 5.

For reads we do a little better. To read a page we collect all five erasure-coded copies, each of which is half a page and reconstruct the data using two of the copies. We could probably improve this further by collecting only two of the copies, but for now our algorithm collects all five copies. So for reads, replication reduces the bandwidth by a factor of 5/2.

Scheduling further reduces the bandwidth, but has a negligible effect on storage capacity. Queueing theory tells us that trying to run a system over about 70% utilization will result in unbounded latencies. We don't do quite that well. We find that we can run our system at about 1/3 of peak theoretical performance without affecting latency. This factor of 3 is our ***scheduling overhead***.

Since not all the customers are presenting their peak load at the same time, we can sell the same performance several times, a practice known as oversubscription. In our experience, we can oversubscribe performance by about a factor of 5.

The units of performance simplify from MB/s/TB to s^{-1}, so 100 MB/s/TB is one overwrite per 10 000 s.

For input-outputs per second (IO/s) we convert bandwidth to IOPS by assuming that most IOs are operations on 32 KiB pages, so we provide 3000 IO/s/TB. The cost of other IOs can be expressed in terms of reads: A write costs 2.5 reads, a file creation costs 6 reads, an empty-file deletion costs 8 reads, and a file renaming costs about 10 reads.

This performance model appears to work well on every parallel workload we have seen. To test this model, we measured how much bandwidth a relatively small test fleet can provide. (We aren't allowed to do these sorts of experiments on the big production fleets.) We measured on multiple clients, where each client has its own mount target on its own Orca. This fleet has 41 storage instances each with a 10 Gbit/s VNIC for a total raw performance of 51.25 GB/s. After replication that's 10.25 GB/s of salable bandwidth. Dividing by 3 to account for scheduling overhead is 3.4 GB/s. Those machines provide a total of 200 TB of salable storage, for a ratio of 17 MB/s/TB. According to our model, with 5-fold oversubscription, this fleet should promise customers 85 MB/s/TB.

Figure 8 shows measured bandwidth. The variance was small so we measured only 6 runs at each size. The measured performance is as follows. When writing into an empty file, block allocation consumes some time, and a single client can get about 434 MB/s, whereas 12 clients can get about 2.0 GB/s. When writing into an existing file, avoiding block allocation overhead, the performance is about 536 MB/s and 2.4 GB/s for 1 and 12 clients respectively.

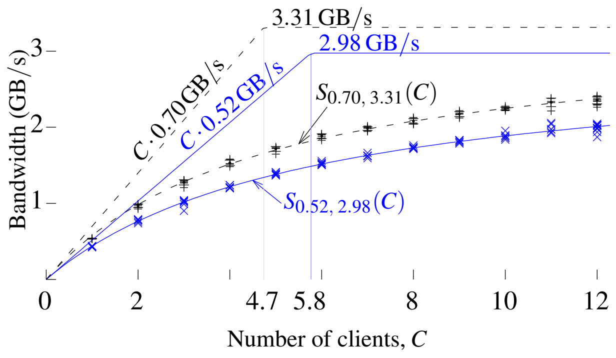

Figure 8: Measured bandwidth. The X-axis is the number of NFS clients. The Y-axis is the cumulative bandwidth achieved. The crosses (in black) show measured performance writing into a preallocated file. The x's (in blue) show measured performance including block allocation. The functions S are the curves fit to a simple-speedup model, with the corresponding linear-speedup shown as lines passing through the origin and the asymptotic speedups shown as horizontal lines. The number of clients at the intercepts are also shown.

We hypothesized that we could model this data as a ***simple-speedup*** curve [9] (a variant of Amdahl's law or of Brent and Graham's Theorem [5,12,27]). In a simple-speedup scenario, as we increase the number of clients, we see a linear speedup which eventually flattens out to give an asymptotic speedup. The curve is parameterized by two numbers l and a. The first value, l, is the ***linear-speedup*** slope which applies when the number of clients C is small where the performance will be $l \cdot C$. The second value, a, is the ***asymptotic speed***, and indicates the performance for large numbers of clients. The simple speedup curve,

$$S_{l,a}(C) = 1/(1/lC + 1/a),$$

is simply half the harmonic mean of the linear-speedup curve lC and the asymptotic speed a.

We fitted of our data to the simple-speedup model and plotted the resulting curves in Figure 8. The asymptotic standard error for the curve fit is less than 1.7%. Visually, the curves fit the data surprisingly well.

We can interpret these curves as follows: When writing to an empty file (which includes block allocation), a few clients can each achieve about 0.52 GB/s, and many clients can achieve a total of 2.98 GB/s. The cutover between "few" and "many" is about 6 clients for this fleet. When writing to a preallocated file, a few clients can each achieve 0.70 GB/s, and many clients can achieve a total of 3.31 GB/s, which is close to our estimate of 3.4 GB/s. The intercept of the speedup curve lines tells us the ***half-power*** points, where half the peak capacity is consumed: 4.7 clients consume half of the fleet's allocate-and-write capacity, and 5.8 clients consume half of the write-without-allocation capacity.

Low latency is hard to achieve in a replicated distributed system. Latency is the elapsed time from an NFS client's request to response in an otherwise unloaded system. For serial workloads, latency can dominate performance. For example, when running over NFS, the `tar` program creates files one at a time, waiting for an acknowledgment that each file exists on stable storage before creating the next file. After looking at various benchmarks, we concluded that we should simply measure `tar`'s runtime on a well-known tarball such as the Linux 4.19.2 source code distribution, which is 839 MB and contains 65 825 objects. Untarring Linux onto local NVMe device takes about 10 s. The same NVMe served over NFS finishes in about 2.5 minutes. FSS finishes in about 10 minutes. Amazon's EFS, which replicates across a metropolitan region, finishes in about an hour. According to this limited experiment, NFS costs a factor of 15, replication within a datacenter costs another factor of 4, and synchronous replication over metropolitan-scale distances costs another factor of 6. Achieving local-filesystem performance in a replicated distributed fault-tolerant filesystem appears ... difficult.

10 Related Work

MPSC is a variation of load-link/store-conditional [41], and seems less susceptible to the ABA problem (in which the same location is read twice and has the same value for both reads, tricking the user into thinking that no transaction has modified the location in the meanwhile) than compare-and-swap [20,21,33]. Version tagging and the ABA problem appeared in [38, p. A-44].

Sinfonia has many similarities to our system. Sinfonia minitransactions [1] are similar to MPSC. Sinfonia uses uses primary-copy replication [14] and can suffer from the split-brain problem, where both primary and replica become active and lose consistency [19]. To avoid split-brain, Sinfonia remotely turns off power to failed machines, but that's just another protocol which can, e.g., suffer from delayed packets, and doesn't solve the problem. We employ Paxos [44], which is a correct distributed consensus algorithm.

Many filesystems have stored at least some their metadata in B-trees [8, 22, 39, 53, 63, 66, 67] and some have gone further, storing both metadata and data in a B-tree or other key-value store [18, 40, 64, 74, 75]. Our B-tree usage is fairly conventional in this regard, except that we store many filesystems in a single B-tree.

ZFS and HDFS [30,69,70,72] support multiple block sizes in one file. Block suballocation and tail merging filesystems [3,63,66] are special cases of this approach.

Some filesystems avoid using Paxos on every operation. For example, Ceph [42] uses Paxos to run its monitors, but replicates data asynchronously. Ceph's crash consistency can result in replicas that are not consistent with each other. Some systems (e.g., [23, 24, 52]) use other failover schemes that have not been proved correct. Some filesystems store all metadata in memory [24, 31, 36, 42, 57], resulting in fast metadata access until the metadata gets too big to fit in RAM. We go to disk on every operation, resulting in no scaling limits.

11 Conclusion

History: The team started with Frigo and Kuszmaul, and the first code commit was on 2016-07-04. Paxos and DASD were implemented by the end of July 2016, and the B-tree was working by November 2016. Sandler joined and started Orca implementation on 2016-08-03. Mazzola Paluska joined on 2016-09-15 and implemented the filesystem schema in the B-tree. The team grew to about a dozen people in January 2017, and is about two dozen people in spring 2019. Control-plane work started in Spring 2017. Limited availability was launched on 2017-07-01, less than one year after first commit (but without a control plane — all configuration was done manually). General availability started 2018-01-29. As of Spring 2019, FSS hosts over 10,000 filesystems containing several petabytes of paid customer data and is growing at an annualized rate of 8- to 60-fold per year (there's some seasonal variation).

Acknowledgments: In addition to the authors, the team that built and operates FSS has included: the data-plane team of Yonatan (Yoni) Fogel, Michael Frasca, Stephen Fridella, Jan-Willem Maessen, and Chris Provenzano; the control-plane team of Vikram Bisht, Bob Naugle, Michael Chen, Ligia Connolly, Yi Fang, Cheyenne T. Greatorex, Alex Goncharov, David Hwang, Lokesh Jain, Uday Joshi, John McClain, Dan Nussbaum, Ilya Usvyatsky, Mahalakshmi Venkataraman, Viktor Voloboi, Will Walker, Tim Watson, Hualiang Xu, and Zongcheng Yang; the product- and program-management team of Ed Beauvais, Mona Khabazan, and Sandeep Nandkeolyar; solutions architect Vinoth Krishnamurthy; and the engineering-management team of Thomas (Vinod) Johnson, Alan Mullendore, Stephen Lewin-Berlin, and Rajagopal Subramaniyan. Heidi Peabody provided administrative assistance. We further rely on the many hundreds of people who run the rest of Oracle Cloud Infrastructure.

References

[1] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamoanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 27(3), November 2009. `doi:10.1145/1629087.1629088`.

[2] Alibaba elastic block storage. Viewed 2018-09-26. `https://www.alibabacloud.com/help/doc-detail/25383.htm`.

[3] Hervey Allen. Introduction to FreeBSD additional topics. In *PacNOG I Workshop*, Nadi, Fiji, 20 June 2005.

[4] Amazon elastic block store. Viewed 2018-09-26. `https://aws.amazon.com/ebs/`.

[5] G. M. Amdahl. The validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, 1967.

[6] Apache Software Foundation. Zookeeper internals, December 2009. `https://zookeeper.apache.org/doc/r3.1.2/zookeeperInternals.html`.

[7] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, February 1972. `doi:10.1145/1734663.1734671`.

[8] Steve Best and Dave Kleikamp. JFS layout. IBM Developerworks, May 2000. `http://jfs.sourceforge.net/project/pub/jfslayout.pdf`.

[9] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 25 1996. (An early version appeared in the *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, pages 207–216, Santa Barbara, California, July 1995.).

[10] Hans-J. Boehm. Transactional memory should be an implementation technique, not a programming interface. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism (HotPar'09)*, pages 15:1–15:6, Berkeley, CA, 30–31 March 2009. `https://www.usenix.org/legacy/events/hotpar09/tech/full_papers/boehm/boehm_html/index.html`.

[11] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos replicated state machines at the basis of a high-performance data store. In *Proceedings of the Eighth USENIX Conference on Networked Systems Design and Implementation*, pages 141–154, Boston, MA, USA, 30 March–1 April 2011. `http://static.usenix.org/event/nsdi11/tech/full_papers/Bolosky.pdf`.

[12] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, April 1974.

[13] Gerth Stølting Brodal, Konstantinos Tsakalidis, Spyros Sioutas, and Kostas Tsichlas. Fully persistent b-trees. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'12)*, pages 602–614, Kyoto, Japan, 17–19 January 2012. `doi:10.1137/1.9781611973099.51`.

[14] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The primary-backup approach. In *Distributed Systems*, chapter 8, pages 199–216. ACM Press/Addison-Wesley, New York, NY, USA, second edition, 1993.

[15] Brent Callaghan, Brian Pawlowski, and Peter Staubach. NFS version 3 protocol specification. IETF RFC 1813, June 1995. `https://www.ietf.org/rfc/rfc1813`.

[16] Călin Casçaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy. *ACM Queue*, 6(5), September 2008. `doi:10.1145/1454456.1454466`.

[17] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC'07)*, pages 398–407, Portland, OR, USA, 12–15 August 2007. `doi:10.1145/1281100.1281103`.

[18] Alexander Conway, Ainesh Bakshi, Yizheng Jiao, Yang Zhan, Michael A. Bender, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E.

Porter, Jun Yuan, and Martin Farach-Colton. File systems fated for senescence? Nonsense, says science! In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*, pages 45–58, 27 February–2 March 2017. https://www.usenix.org/conference/fast17/technical-sessions/presentation/conway.

[19] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in partitioned network. *Computing Surveys*, 17(3):341–370, September 1985. doi:10.1145/5505.5508.

[20] David L. Detlefs, Christine H. Flood, Alexander T. Garthwaite, Paul A. Martin, Nir N. Shavit, and Guy L. Steele Jr. Even better DCAS-based concurrented deques. In *Proceedings of the 14th International Conference on Distributed Computing (DISC'00)*, pages 59–73, 4–6 October 2000.

[21] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele Jr. Lock-free reference counting. *Distributed Computing*, 15(4):255–271, December 2002. Special Issue: Selected papers from PODC'01. doi:10.1017/s00446-002-0079-z.

[22] Matthew Dillon. The hammer filesystem, 21 June 2008. https://www.dragonflybsd.org/hammer/hammer.pdf.

[23] Mark Fasheh. OCFS2: The oracle clustered file system version 2. In *Proceedings of the 2006 Linux Symposium*, pages 289–302, 2006. https://oss.oracle.com/projects/ocfs2/dist/documentation/fasheh.pdf.

[24] GlusterFS. http://www.gluster.org.

[25] Google persistent disk. Viewed 2018-09-26. https://cloud.google.com/persistent-disk/.

[26] Goetz Graefe. A survey of B-tree locking techniques. *ACM Transactions on Database Systems*, 35(3), July 2010. Article No. 16. doi:10.1145/1806907.1806908.

[27] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, March 1969.

[28] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[29] Jim N. Gray. Notes on data base operating systems. In *Operating Systems—An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, chapter 3. Springer-Verlag, 1978.

[30] Add support for variable length block. HDFS Ticket, July 2012. https://issues.apache.org/jira/browse/HDFS-3689.

[31] Hdfs architecture, 2013. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html#Large_Data_Sets.

[32] M. Herlihy and J. E. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture (ISCA'93)*, pages 289–300, San Diego, CA, USA, 16–19 May 1993. doi:10.1145/173682.165164.

[33] Maurice Herlihy. Wait-free synchronizatoin. *ACM Transactions on Programming Languages and Systems (TOPLAS))*, 11(1):124–149, January 1991. doi:10.1145/114005.102808.

[34] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS))*, 12(3):463–492, July 1990. doi:10.1145/78969.78972.

[35] 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 19–19, 17–21 January 1994. http://usenix.org/publications/library/proceedings/sf94/full_papers/hitz.a.

[36] Valentin Höbel. LizardFS: Software-defined storage as it should be, 27 April 2016. In German. https://www.golem.de/news/lizardfs-software-defined-storage-wie-es-sein-soll-1604-119518.html.

[37] IBM. *Data File Handbook*, March 1966. C20-1638-1. http://www.bitsavers.org/pdf/ibm/generalInfo/C20-1638-1_Data_File_Handbook_Mar66.pdf.

[38] IBM. *IBM System/370 Extended Architecture—Principles of Operation*, March 1983. Publication number SA22-7085-0. https://archive.org/details/bitsavers_

ibm370prinrinciplesofOperationMar83_40542805.

[39] Apple Inc. Hfs plus volume format. Technical Note TN1150, Apple Developer Connection, 5 March 2004. https://developer.apple.com/library/archive/technotes/tn/tn1150.html.

[40] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. BetrFS: A write-optimization in a kernel file system. *ACM Transactions on Storage (TOS)*, 11(4), November 2015. doi:10.1145/2798729.

[41] Eric H. Jensen, Gary W. Hagensen, and Jeffrey M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Laboratory, Livermore, California, November 1987. https://e-reports-ext.llnl.gov/pdf/212157.pdf.

[42] M. Tim Jones. Ceph: A Linux petabyte-scale distributed file system, 4 June 2004. https://www.ibm.com/developerworks/linux/library/l-ceph/index.html.

[43] Sakis Kasampalis. Copy on write based file systems performance analysis and implementation. Master's thesis, Department of Informatics, The Technical University of Denmark, October 2010. http://sakisk.me/files/copy-on-write-based-file-systems.pdf.

[44] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, May 1998. doi:10.1145/279227.279229.

[45] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4 (Whole Number 121)):51–58, December 2001. https://www.microsoft.com/en-us/research/publication/paxos-made-simple/.

[46] Butler Lampson. Atomic transactions. In *Distributed Systems—Architecture and Implementation*, volume 100. Springer Verlag, 1980.

[47] Philip L. Lehman and S. Bing Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, December 1981. doi:10.1145/319628.319663.

[48] Yossi Lev, Mark Moir, and Dan Nussbaum. PhTM: Phased transactional memory. In *The Second ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, USA, 16 August 2007.

[49] A. J. Lewis. LVM howto, 2002. http://tldp.org/HOWTO/LVM-HOWTO/.

[50] Bruce G. Lindsay. Single and multi-site recovery facilities. In I. W. Draffan and F. Poole, editors, *Distributed Data Bases*, chapter 10. Cambridge University Press, 1980. Also available as [51].

[51] Bruce G. Lindsay, Patricia G. Selinger, Cesare A. Galtieri, James N. Gray, Raymond A. Lorie, Thomas G. Price, Franco Putzolu, Irving L. Traiger, and Bradford W. Wade. Notes on distributed databases. Research Report RJ2571, IBM Research Laboratory, San Jose, California, USA, July 1979. http://domino.research.ibm.com/library/cyberdig.nsf/papers/A776EC17FC2FCE73852579F100578964/$File/RJ2571.pdf.

[52] The Lustre file system. lustre.org.

[53] Avantika Mathur, MingMing Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: Current status and future plans. In *Proceedings of the Linux Symposium*, Ottawa, Ontario, Canada, 27–30 June 2007.

[54] Microsoft azure blob storage. Viewed 2018-09-26. https://azure.microsoft.com/en-us/services/storage/blobs/.

[55] Microsoft SMB Protocol and CIFS Protocol overview, May 2018. https://docs.microsoft.com/en-us/windows/desktop/FileIO/microsoft-smb-protocol-and-cifs-protocol-overview.

[56] Barton P. Miller, Louis Fredersen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM (CACM)*, 33(12):32–44, December 1990. doi:10.1145/96267.96279.

[57] MooseFS fact sheet, 2018. `https://moosefs.com/factsheet/`.

[58] Brian Oki and Barbara Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC'88)*, pages 8–17, Toronto, Ontario, Canada, 15–17 August 1988. `doi:10.1145/62546.62549`.

[59] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of USENIX ATC'14: 2014 USENIX Annual Technical Conference*, Philadelphia, PA, USA, 19–20 June 2014. `https://www.usenix.org/node/184041`.

[60] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm (extended version), 20 May 2014. Extended version of [59]. `https://raft.github.io/raft.pdf`.

[61] Oracle cloud infrastructure block volumes. Viewed 2018-09-26. `https://cloud.oracle.com/en_US/storage/block-volume/features`.

[62] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, June 1960. `doi:10.1137/0108018`.

[63] Hans T. Reiser. Reiser4, 2006. Archived from the original on 6 July 2006. `https://web.archive.org/web/20060706032252/http://www.namesys.com:80/`.

[64] Kai Ren and Garth Gibson. TABLEFS: Enhancing metadata efficiency in the local file system. In *USENIX Annual Technical Conference*, pages 145–156, 2013. `https://www.usenix.org/system/files/conference/atc13/atc13-ren.pdf`.

[65] Ohad Rodeh. B-trees, shadowing, and clones. *ACM Transactions on Computational Logic*, 3(4):15:1–15:27, February 2008. `doi:10.1145/1326542.1326544`.

[66] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3), August 2013. Article No. 9. `doi:10.1145/2501620.2501623`.

[67] Mark Russinovich. Inside Win2K NTFS, part 1. *ITProToday*, 22 October 2000. `https://www.itprotoday.com/management-mobility/inside-win2k-ntfs-part-1`.

[68] Spencer Shepler, Brent Callaghan, David Robinson, Robert Thurlow, Carl Beame, Mike Eisler, and David Noveck. Network File System (NFS) version 4 protocol. IETF RFC 3530, April 2003. `https://www.ietf.org/html/rfc3530`.

[69] Chris Siebenmann. ZFS's `recordsize`, holes in files, and partial blocks, 27 September 2017. Viewed 2018-08-30. `https://utcc.utoronto.ca/~cks/space/blog/solaris/ZFSFilePartialAndHoleStorage`.

[70] Chris Siebenmann. What ZFS gang blocks are and why they exist, 6 January 2018. Viewed 2018-08-30. `https://utcc.utoronto.ca/~cks/space/blog/solaris/ZFSGangBlocks`.

[71] W. Ricxhard Stevens. TCP slow start, congestion avoidance, fast retransmit and fast recovery algorithms. IETF RFC 2001, January 1997. `https://www.ietf.org/html/rfc2001`.

[72] Sun Microsystems. ZFS on-disk specification—draft, August 2006. `http://www.giis.co.in/Zfs_ondiskformat.pdf`.

[73] Lingxiang Xiang and Michael L. Scott. Conflict reduction in hardware transactions using advisory locks. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'15)*, pages 234–243, Portland, OR, USA, 13–15 June 2015. `doi:10.1145/2755573.2755577`.

[74] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Writes wrought right, and other adventures in file system optimization. *Transactions on Storage—Special Issue on USENIX FAST 2016*, 13(1):3:1–3:21, March 2017. `doi:10.1145/3032969`.

[75] Yang Zhan, Alexander Conway, Yizheng Jiao, Eric Knorr, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. The full path to full-path indexing. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*, pages 123–138,

Oakland, CA, USA, 12–15 February 2018.
https://www.usenix.org/conference/fast18/presentation/zhan.