Table of Contents

Chapter 1

Background: Namespace Scalability

Namespaces resolve names to data. Traditionally, namespaces are hierarchical and allow users to group similar data together. Although file system namespaces are the most well known, other examples include DNS, LAN network topologies, and scoping in programming languages. File system namespaces are popular because they fit our mental organization as humans and are part of the POSIX IO standard. The momentum of namespaces as an abstraction and the overwhelming amount of legacy code written for namespaces makes the data model relatively future proof.

In file systems, whenever a file is created, modified, or deleted, the client must access the file's metadata. File system metadata contains information about the file, like size, links, access times, attributes, permissions/access control lists (ACLs), and ownership. In single disk file systems, clients consult metadata before seeking to data, by translating the file name to an inode and using that inode to lookup metadata in an inode table located at a fixed location on disk. Distributed file systems use a

similar idea; clients look in one spot for their metadata, usually a metadata service, and use that information to find data in a storage cluster. State-of-the-art distributed file systems decouple metadata from data access so that data and metadata I/O can scale independently [6, 23, 25, 51, 53, 56]. Unfortunately, recent trends have shown that separating metadata and data traffic is insufficient for scaling to large systems and identify the metadata service as the performance critical component.

First, we discuss general file system use cases and characterize the resultant metadata workloads. Next, we describe three semantics that users expect from file systems: strong consistency, durability, and a hierarchical organization. For each semantic, we explain why it is problematic for today's metadata workloads and survey optimizations in related work.

1.1 Metadata Workloads

File system metadata is small and highly accessed so the workloads are made up of many small requests [36, 5]. This skewed workload causes scalability issues in file systems because solutions for scaling data IO do not work for metadata IO [36, 4, 6, 51]. Unfortunately, this metadata problem is becoming more common and the same challenges that plagued HPC systems for years are finding their way into the cloud. Jobs that deal with many small files (*e.g.*, log processing and database queries [48]) and large numbers of simultaneous clients (*e.g.*, MapReduce jobs [31]) are especially problematic.

If the use case is narrow enough, then developers in these domains can build application-specific storage stacks based on a thorough understanding of the workloads (*e.g.*, temperature zones for photos [32], well-defined read/write phases [15, 14], synchronization only needed during certain phases [28, 60], workflows describing computation [57, 22], etc.). Unfortunately, this "clean-slate" approach only works for one type of workload. To build a general-purpose file system, we need a thorough understanding of today's workloads and how they affect metadata services.

In this section, we describe modern applications (*i.e.* standalone programs, compilers, and runtimes) and common user behaviors (*i.e.* how users interact with file systems) that result in metadata-intensive workloads. For each use case, we provide motivation from HPC and cloud workloads; specifically, we look at users using the file system in parallel to run large-scale experiments in HPC and parallel runtimes that use the file system, such as MapReduce [15] (referred to as Hadoop, the open-source counterpart) and Spark [58].

1.1.1 Spatial Locality Within Directories

File system namespaces have semantic meaning; data stored in directories is related and is usually accessed together [51, 52]. Programs, compilers, and runtimes are usually triggered by users so the inputs/outputs to the job are stored within the user's home directory [50]. Hadoop and Spark enforce POSIX IO permissions and ownership to ensure users and bolt-on software packages operate within their assigned directories [3]. User behavior also exhibits locality. Listing directories after jobs is common and accesses

are localized to the user's working directory [36, 5].

A problem in HPC is users unintentionally accessing files in another user's directory. This behavior introduces false sharing and many file systems revoke locks and cached items for all clients to ensure consistency. While HPC tries to avoid these situations with workflows [59, 60], it still happens in distributed file systems when users unintentionally access directories in a shared file system.

1.1.2 Temporal Locality During Flash Crowds

Creates in the same directory is a problem in HPC , mostly due to checkpoint-restart [7]. Flash crowds of checkpoint-restart clients simultaneously open, write, and close files within a directory. But the workload also appears in cloud workloads: Hadoop/Spark use the file system to assign work units to workers and the performance is proportional to the open/create throughput of the underlying file system [55, 42, 43]; Big Data Benchmark jobs examined in [12] have on the order of 15,000 file opens or creates just to start a single Spark query and the Lustre system they tested on did not handle creates well, showing up to a 24× slowdown compared to other metadata operations. Common approaches to solve these types of bottlenecks is to change the application behavior or to design a new file system, like BatchFS [59] or DeltaFS [60], that uses one set of metadata optimizations for the entire namespace.

1.1.3 Listing Directories

As discussed before, this is common for general users (*e.g.*, reading a directory after a job completes), but users also use the file system for its centralized consistency. For example, users often leverage the file system to check the progress of jobs using `ls` even though this operation is notoriously heavy-weight [11, 20]. The number of files or size of the files is indicative of the progress. This practice is not too different from cloud systems that use the file system to manage the progress of jobs; Spark/Hadoop writes to temporary files, renames them when complete, and creates a "DONE" file to indicate to the runtime that the task did not fail and should not be re-scheduled on another node. For example, the browser interface lets Hadoop/Spark users check progress by querying the file system and returning a % of job complete metric.

1.1.4 Performance and Resource Utilization

The metadata workloads discussed in the previous section saturate resources on the metadata servers. Even small scale programs can show the effect; the resource utilization on the metadata server when compiling the Linux source code in a CephFS mount is shown in Figure 1.2. The `untar` phase, which is characterized by many creates, has the highest resource usage (combined CPU, network, and disk) on the metadata server because of the number of RPCs needed for consistency and durability. Many of our benchmarks use a create-heavy workload because it has high resource utilization.

Figure 1.1 shows the metadata locality for this workload. The "heat" of each directory is calculated with per-directory metadata counters, which are tempered with

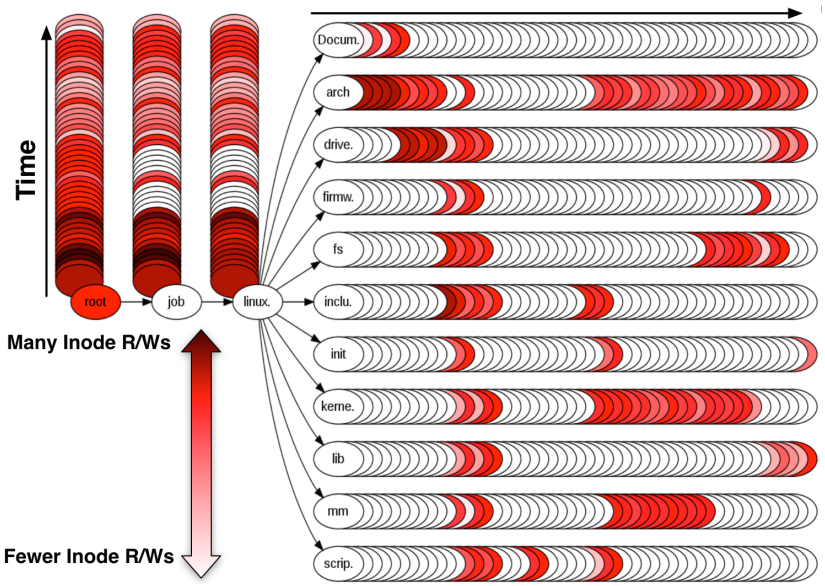

Figure 1.1: Metadata hotspots, represented by different shades of red, have spatial and temporal locality when compiling the Linux source code. The hotspots are calculated using the number of inode reads/writes and smoothed with an exponential decay.

an exponential decay. The hotspots can be correlated with phases of the job: untarring the code has high, sequential metadata load across directories and compiling the code has hotspots in the `arch`, `kernel`, `fs`, and `mm` directories.

1.2 Global Semantics: Strong Consistency

Access to metadata in a POSIX IO-compliant file system is strongly consistent, so reads and writes to the same inode or directory are globally ordered. The benefit of strong consistency is that clients and servers have the same view of the data, which makes state changes easier to reason about. The cost of this "safety" is performance.

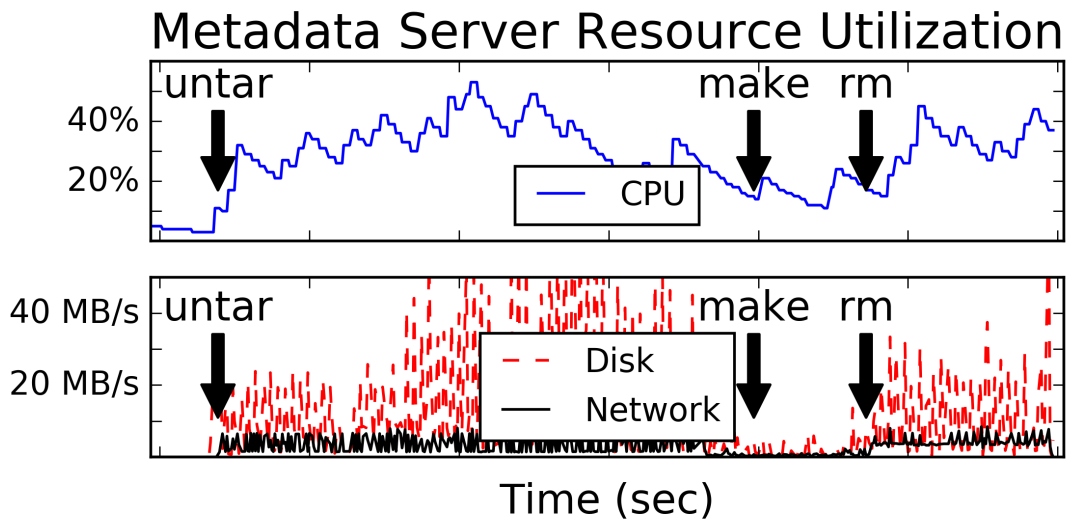

Figure 1.2: [source] **For the CephFS metadata server, create-heavy workloads (*e.g.*, `untar`) incur the highest disk, network, and CPU utilization because of consistency/durability demands.**

The synchronization and serialization machinery needed to ensure that all clients see the same state has high overhead. To make sure that all nodes or processes in the system are seeing the same state, they must come to an agreement. This limits parallelization and metadata performance has been shown to *decrease* with more sockets in Lustre [13]. As a result, and because it is simpler to implement, many distributed file systems limit the number of threads to one for all metadata servers [51, 6, 35].

Coming to an agreement has its own set of performance and accuracy trade-offs. Sophisticated, standalone consensus engines like PAXOS [29], Zookeeper [27], or Chubby [10] are common techniques for maintaining consistent versions of state in groups of processes that may disagree, but putting them in the data path is a large

bottleneck. In fact, PAXOS is used in Ceph and Zookeeper in Apache stacks to maintain cluster state but not for mediating IO.

Many distributed file systems use state machines to guard access to file system metadata. These state machines are stored with traditional file system metadata and they enforce the level of isolation that clients are guaranteed while they are reading or writing a file. CephFS [1, 50] calls the state machines "capabilities" and they are managed by authority metadata servers, GPFS [38] calls the state machines "write locks" and they can be shared, Panasas [54] calls the state machines "locks" and "callbacks", IndexFS [35] calls the state machines "leases" and they are dropped after a timeout, Lustre [39] calls the state machines "locks" and they protect inodes, extents, and file locks with different modes of concurrency [49]. Because this form of consistency is a bottleneck for metadata access, many systems optimize performance by improving locking protocols (Section §1.2.1), caching inodes (Section §1.2.2), and relaxing consistency (Section §1.2.3). We refer to these state machines as "locks" from now.

1.2.1 Lock Management

The global view of locks are read and modified with RPCs from clients. Single node metadata services, such as the Google File System (GFS) [23] and HDFS [43] have the simplest implementations and expose simple lock configurations like timeout thresholds. These implementations do not scale for metadata-heavy workloads so a natural approach to improving performance is to use a cluster to manage locks.

Distributed lock management systems spread the lock request load across a

cluster of servers. One approach is to distribute locks with the data by co-locating metadata servers with storage servers. PVFS2 [18] lets users spin up metadata servers on both storage and non-storage servers but the disadvantage of this approach is resource contention and poor file system metadata locality, respectively. Another approach is to orchestrate a dedicated metadata cluster from a centralized lock manager that accounts for load imbalance and locality. GPFS [38] assigns a process to be the "global lock manager", which is the authority of all locks and synchronizes access to metadata. Local servers become the authority of metadata by contacting the global lock manager, enabling optimizations like reducing RPCs. A decentralized version of this approach is to associate an authority process per inode. For example, Lustre, CephFS, IndexFS, and Panasas servers manage parts of the namespace and respond to client requests for locks. These approaches have more complexity but are flexible enough to service a range of workloads.

1.2.2 Caching Inodes

The discussion above refers to server-server lock exchange, but systems can also optimize client-server lock management. Caching inodes on both the client and server lets clients read/modify metadata locally. This reduces the number of RPCs required to agree on the state of metadata. For example, CephFS caches entire inodes, Lustre caches lookups, IndexFS caches ACLs, PVFS2 maintains a namespace cache and an attribute cache, Panasas lets clients read, cache, and parse directories, GPFS and Panasas cache the results of `stat()` [17], and GFS caches file location/striping

strategies. Some systems, like Ursa Minor [45] and pNFS [25] maintain client caches to reduce the overheads of NFS. These caches improve performance but the cache coherency mechanisms add significant complexity and overhead for some workloads.

1.2.3 Relaxing Consistency

A more disruptive technique is to relax the consistency semantics in the file system. Following the models pioneered by Amazon's eventual consistency [16] and the more fine-grained consistency models defined by Terry et al. [46], these techniques are gaining popularity because maintaining strong consistency has high overhead and because weaker guarantees are sufficient for many target applications.

Batching requests together is one form of relaxing consistency because updates are not seen immediately. PVFS2 batches creates, Panasas combines similar requests (*e.g.*, create and stat) together into one message, and Lustre surfaces configurations that allow users to enable and disable batching. Technically, batching requests is weaker than per-request strong consistency but the technique is often acceptable in POSIX-compliant systems.

More extreme forms of batching "decouple the namespace", where clients lock the subtree they want exclusive access to as a way to tell the file system that the subtree is important or may cause resource contention in the near-future. Then the file system can change its internal structure to optimize performance. One software-based approach is to prevent other clients from interfering with the decoupled directory until the first client commits changes back to the global namespace. This delayed merge (*i.e.* a form

of eventual consistency) and relaxed durability improves performance and scalability by avoiding the costs of RPCs, synchronization, false sharing, and serialization. BatchFS and DeltaFS clients merge updates when the job is complete to avoid these costs and to encourage client-side processing. Another example approach is to move metadata intensive workloads to more powerful hardware. For example, for high metadata load MarFS [24] uses a cluster of metadata servers and TwoTiers [21] uses SSDs for the metadata server back-end. While the performance benefits of decoupling the namespace are obvious, applications that rely on the file system's guarantees must be deployed on an entirely different system or re-written to coordinate strong consistency themselves.

Even more drastic departures from POSIX IO allow writers and readers to interfere with each other. GFS leaves the state of the file undefined rather than consistent, forcing applications to use append rather than seeks and writes; in the cloud, Spark and Hadoop stacks use the Hadoop File System (HDFS) [44], which lets clients ignore this type of consistency completely by letting interfering clients read files opened for writing [28]; and CephFS offers the "Lazy IO" option, which lets clients buffer reads/writes even if other clients have the file open and if the client maintains its own cache coherency [1]. As noted earlier, many of these relaxed consistency semantics are for application-specific optimizations.

1.3 Global Semantics: Durability

While durability is not specified by POSIX IO, users expect that files they reate or modify survive failures. The accepted technique for achieving durability is to append events to a journal of metadata updates. Similar to LFS [37] and WAFL [26] the metadata journal is designed to be large (on the order of MBs) which ensures (1) sequential writes into the storage device (*e.g.*, object store, local disk, etc.) and (2) the ability for daemons to trim redundant or irrelevant journal entries. We refer to metadata updates as a journal, but of course, terminology varies from system to system (*e.g.*, operation log, event list, etc.). Ensuring durability has overhead so many performance optimizations target the file system's journal format and mechanisms.

1.3.1 Journal Format

A big point of contention for distributed file systems is not the technique of journaling metadata updates, rather it is the format of metadata. CephFS employs a custom on-disk metadata format that behaves more like a "pile system" [50]. Alternatively, IndexFS stores its journal in LSM trees for fast insertion and lookup. TableFS [34] lays out the reasoning for using LSM trees: the size of metadata (small) and the number of files (many) fits the LSM model well, where updates are written to the local file system as large objects (*e.g.*, write-ahead logs, SSTables, large files). Panasas separates requests out into separate logs to account for the semantic meaning and overhead of different requests ("op-log" for creates and updates and "cap-log" for capabilities). Many papers

claim that an optimized journal format leads to large performance gains [34, 35, 59] but we have found that the journal safety mechanisms have a much bigger impact on performance [40].

1.3.2 Journal Safety

We define three types of durability: global, local, and none. Global durability means that the client or server can fail at any time and metadata will not be lost because it is "safe" (*i.e.* striped or replicated across a cluster). GFS achieves global durability by replicating its journal from the master local disk to remote nodes and CephFS streams the journal into the object store. Local durability means that metadata can be lost if the client or server stays down after a failure. For example, in BatchFS and DeltaFS unwritten metadata updates are lost if the client (and/or its disk) fails and stays down. None means that metadata is volatile and that the system provides no guarantees when clients or servers fail. None is different than local durability because regardless of the type of failure, metadata will be lost when components die. Storing the journal in a RAMDisk would be an example of a system with a durability level of none.

Implementations of the types of durability vary, ranging from completely software-defined storage to architectures where hardware and software are more tightly-coupled, such as Panasas. Panasas assigns durability components to specific types of hardware. The journal is stored in battery-backed NVRAM and later replicated to both remote peers and metadata on objects. The software that writes the actual operations behaves similar to WAFL/LFS without the cleaner. The system also stores different kinds of

metadata (system vs. user, read vs. write) in different places. For example, directories are mirrored across the cluster using RAID1. This domain-specific mapping to hardware achieves high performance but sacrifices cost flexibility.

1.4 Hierarchical Semantics

Users identify and access file system data with a path name, which is a list of directories completed with a file name. File systems traverse (or resolve) paths to check permissions and to verify that files exist. Files and directories inherit some of the semantics from their parent directories, like ownership groups and permissions. For some attributes, like access and modifications times, parent directories must be updated as well.

To maintain these semantics file systems implement path traversal. Path traversal starts at the root of the file system and checks each path component until reaching the desired file. This process has write and read amplification because accessing lower subtrees in the hierarchy requires RPCs to upper levels. To reduce this amplification, many systems try to leverage the workload's locality; namely that directories at the top of a namespace are accessed more often [35] and files that are close in the namespace spatially are more likely to be accessed together [51, 52].

1.4.1 Caching Paths

To leverage the fact that directories at the top of the namespace are accessed more often, some systems cache "ancestor directories", *i.e.* metadata for entire paths. In

GIGA+ [33], clients contact the parent and traverse down its "partition history" to find which authority metadata server has the data. In the follow-up work, IndexFS, improves lookups and creates by having clients cache permissions instead of attributes. Similarly, Lazy Hybrid [9] hashes the file name to locate metadata but maintains extra per-file metadata to manage permissions. Although these techniques improve performance and scalability, especially for create intensive workloads, they do not leverage the locality inherent in file system workloads. For example, IndexFS's inode cache reduces RPCs by caching ancestor paths for metadata writes but the cache can be thrashed by random reads.

Caching can also be used to exploit locality. Many file systems hash the namespace across metadata servers to distribute load evenly, but this approach sacrifices workload locality. To compensate, systems like IndexFS and SkyFS [56] achieve locality by adding a metadata cache. This approach has a large space overhead, so HBA [61] uses hierarchical bloom filter arrays. Unfortunately, caching inodes is limited by the size of the caches and only performs well for temporal metadata, instead of spatial metadata locality [52, 41, 30]. Furthermore, keeping the caches coherent requires a fair degree of sophistication, which incurs overhead and limits the file system's ability to dynamically adapt to flash crowds.

1.4.2 Metadata Distribution

File systems like GIGA+, CephFS, SkyFS, HBA, and Ursa Minor use active-active metadata clusters instead of single metadata servers. Applications perform better

with dedicated metadata servers [41, 35] but provisioning a metadata server for every client is unreasonable. So finding the right number of metadata servers per client is a challenge. This problem is exacerbated by current hardware and software trends that encourage more clients; for example, HPC architectures are transitioning from complex storage stacks with burst buffer, file system, object store, and tape tiers to more simplified stacks with just a burst buffer and object store [8]. This puts pressure on data access because more requests end up hitting the same layer and old techniques of hiding latencies while data migrates across tiers are no longer applicable.

1.4.2.1 Correctness: Addressing Metadata Inconsistency

Distributing metadata across a cluster requires distributed transactions and cache coherence protocols to ensure strong consistency. For example, file creates are fast in IndexFS because directories are fragmented and directory entries can be written in parallel but reads are subject to cache locality and lease expirations. ShardFS [55] makes the opposite trade-off because metadata reads are fast and resolve with 1 RPC while metadata writes are slow for everyone because they require serialization and multi-server locking. ShardFS achieves this by pessimistically replicating directory state and using optimistic concurrency control for conflicts, where operations fall back to two-phase locking if there is a conflict at verification time.

Another example of the overheads of addressing inconsistency is how CephFS maintains client sessions and inode caches for capabilities (which in turn make metadata access faster). When metadata is exchanged between metadata servers these session-

s/caches must be flushed and new statistics exchanged with a scatter-gather process; this halts updates on the directories and blocks until the authoritative metadata server responds [2]. These protocols are discussed in more detail in the next section but their inclusion here is a testament to the complexity of migrating metadata.

1.4.2.2 Performance: Leveraging Locality

Approaches that leverage the workload's spatial locality (*i.e.* requests targeted at a subset of directories or files) focus on metadata distribution across a cluster. File systems that hash their namespace spread metadata evenly across the cluster but do not account for spatial locality. IndexFS tries to alleviate this problem by distributing whole directories to different nodes. While this is an improvement, it does not address the fundamental data layout problem. Table-based mapping, done in systems like SkyFS, pNFS, and CalvinFS [47], is another metadata sharding technique, where the mapping of path to inode is done by a centralized server or data structure. These systems are static and while they may be able to exploit locality at system install time, their ability to scale or adapt with the workload is minimal.

Another technique is to assign subtrees of the hierarchical namespace to server nodes. Most systems use a static scheme to partition the namespace at setup, which requires a knowledgeable administrator. Ursa Minor and Farsite [19] traverse the namespace to assign related inode ranges, such as inodes in the same subtree, to servers. This benefits performance because the metadata server nodes can act independently without synchronizing their actions, making it easy to scale for breadth assuming that

incoming data is evenly partitioned. If carefully planned, assigning metadata to servers can achieve both even load distribution and locality, which facilitates multi-object operations and more efficient transactions. Unfortunately, static distribution limits the system's ability to adapt to hotspots/flash crowds and to maintain balance as data is added. Some systems, like Panasas, allow certain degrees of dynamicity by supporting the addition of new subtrees at runtime, but adapting to the current workload is ignored.

1.4.2.3 Performance: Load Balancing

One approach for improving metadata performance and scalability is to alleviate overloaded servers by load balancing metadata IO across a cluster. Common techniques include partitioning metadata when there are many writes and replicating metadata when there are many reads. For example, IndexFS partitions directories and clients write to different partitions by grabbing leases and caching ancestor metadata for path traversal; it does well for strong scaling because servers can keep more inodes in the cache which results in less RPCs. Alternatively, ShardFS replicates directory state so servers do not need to contact peers for path traversal; it does well for read workloads because all file operations only require 1 RPC and for weak scaling because requests will never incur extra RPCs due to a full cache. CephFS employs both techniques to a lesser extent; directories can be replicated or sharded but the caching and replication policies do not change depending on the balancing technique [52, 50]. Despite the performance benefits these techniques add complexity and jeopardize the robustness and

performance characteristics of the metadata service because the systems now need (1) policies to guide the migration decisions and (2) mechanisms to address inconsistent states across servers [41].

Setting policies for migrations is arguably more difficult than adding the migration mechanisms themselves. For example, IndexFS and CephFS use the GIGA+ technique for partitioning directories at a predefined threshold and using lazy synchronization to redirect queries to the server that "owns" the targeted metadata. Setting policies for when to partition directories and when to migrate the directory fragments vary between systems: GIGA+ partitions directories when the size reaches a certain number of files and migrates directory fragments immediately; CephFS partitions directories when they reach a threshold size or when the write temperature reaches a certain value and migrates directory fragments when the hosting server has more load than the other servers in the metadata cluster. Another policy is when and how to replicate directory state; ShardFS replicates immediately and pessimistically while CephFS replicates only when the read temperature reaches a threshold. There is a wide range of policies and it is difficult to maneuver tunables and hard-coded design decisions.

1.5 Conclusion

This survey suggests that storage systems struggle:

1. **handling general-purpose workloads**. General-purpose file systems are hard to optimize so many users, including their applications (*i.e.* standalone programs,

compilers, and runtimes) and behaviors (*i.e.* how users interact with file systems), resort to application-specific storage stacks.

2. **selecting optimizations**. Optimizations must work together because they are dependent on each other. For example, we have found that for some workloads the metadata protocols in CephFS are inefficient and have a bigger impact on performance and scalability than load balancing. As a result, understanding these protocols improves load balancing and helps developers select metrics that systems should use to make migration decisions (*e.g.*, which operations reflect the state of the system), what types of requests cause the most load, and how an overloaded system reacts (*e.g.*, increasing latencies, lower throughput, etc.).

3. **guiding optimizations with policies**. Policies should be shaped by applications but most policies are hard-coded into the storage system or exposed as confusing configurations. This is exacerbated by software layering and the "skinny waist" to the storage system, which results in feature duplication and long code paths.

We use the programmable storage approach to ease these burdens and to facilitate more scalable namesapces.

Bibliography

[1] Ceph Documentation. http://docs.ceph.com/docs/master/cephfs/capabilities/, December 2017.

[2] Ceph Documentation. http://docs.ceph.com/docs/jewel/dev/mds_internals/data-structures/, December 2017.

[3] Hadoop 3.0 Documentation. https://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-hdfs/HdfsPermissionsGuide.html, February 2018.

[4] C. L. Abad, H. Luu, Y. Lu, and R. Campbell. Metadata Workloads for Testing Big Storage Systems. Technical report, Citeseer, 2012.

[5] C. L. Abad, H. Luu, N. Roberts, K. Lee, Y. Lu, and R. H. Campbell. Metadata Traces and Workload Models for Evaluating Big Storage Systems. In *Proceedings of the International Conference on Utility and Cloud Computing*, UCC '12.

[6] S. R. Alam, H. N. El-Harake, K. Howard, N. Stringfellow, and F. Verzelloni. Parallel

I/O and the Metadata Wall. In *Proceedings of the Workshop on Parallel Data Storage*, PDSW '11.

[7] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. PLFS: a checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09.

[8] J. Bent, B. Settlemyer, and G. Grider. Serving Data to the Lunatic Fringe.

[9] S. A. Brandt, E. L. Miller, D. D. E. Long, and L. Xue. Efficient Metadata Management in Large Distributed Storage Systems. In *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies*, MSST '03, 2003.

[10] M. Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06.

[11] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig. Small-file Access in Parallel File Systems. In *Proceedings of the Symposium on Parallel and Distributed Processing*, IPDPS '09.

[12] N. Chaimov, A. Malony, S. Canon, C. Iancu, K. Z. Ibrahim, and J. Srinivasan. Scaling Spark on HPC Systems. In *Proceedings of the Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16.

[13] K. Chasapis, M. F. Dolz, M. Kuhn, and T. Ludwig. Evaluating Lustre's Metadata Server on a Multi-socket Platform. In *Proceedings of the 9th Parallel Data Storage Workshop*, PDSW '14, 2014.

[14] J. Dean. Evolution and future directions of large-scale storage and computation systems at Google. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10.

[15] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th USENIX Symposium on Operarting Systems Design & Implementation*, OSDI '04, 2004.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[17] B. Depardon, G. Le Mahec, and C. Séguin. Analysis of six distributed file systems. Technical report, French Institute for Research in Computer Science and Automation, 2013.

[18] A. Devulapalli and P. Wyckoff. File creation strategies in a distributed metadata file system. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10. IEEE, 2007.

[19] J. R. Douceur and J. Howell. Distributed Directory Service in the Farsite File System. In *Proceedings of the 7th Symposium on Operating Systems Design & Implementation*, OSDI '06, 2006.

[20] M. Eshel, R. Haskin, D. Hildebrand, M. Naik, F. Schmuck, and R. Tewari. Panache: A Parallel File System Cache for Global File Access. In *Proceedings of the Conference on File and Storage Technologies*, FAST '10.

[21] S. Faibish, J. Bent, U. Gupta, D. Ting, and P. Tzelnic. Slides: 2 tier storage architecture.

[22] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral. The Spack Package Manager: Bringing Order to HPC Software Chaos. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15.

[23] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03, 2003.

[24] G. Grider, D. Montoya, H.-b. Chen, B. Kettering, J. Inman, C. DeJager, A. Torrez, K. Lamb, C. Hoffman, D. Bonnie, R. Croonenberg, M. Broomfield, S. Leffler, P. Fields, J. Kuehn, and J. Bent. MarFS - A Scalable Near-Posix Metadata File System with Cloud Based Object Backend. In *Work-in-Progress at Proceedings of the Workshop on Parallel Data Storage*, PDSW'15, November 2015.

[25] D. Hildebrand and P. Honeyman. Exporting Storage Systems in a Scalable Manner

with pNFS. In *Proceedings of the 22Nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies*, MSST '05, 2005.

[26] D. Hitz, J. Lau, and M. Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Technical Conference*, WTEC '94.

[27] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10.

[28] J. D. Kamal Hakimzadeh, Hooman Peiro Sajjad. Scaling HDFS with a Strongly Consistent Relational Model for Metadata. In *Proceedings of the International Conference on Distributed Applications and Interoperable Systems*, DAIS '14.

[29] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[30] W. Li, W. Xue, J. Shu, and W. Zheng. Dynamic Hashing: Adaptive Metadata Management for Petabyte-scale Scale Systems. In *Proceedings of the 23rd IEEE/14th NASA Goddard Conference on Mass Storage Systems and Technologies*, MSST '06, 2006.

[31] K. McKusick and S. Quinlan. GFS: Evolution on Fast-forward. *Commununications ACM*, 53(3):42–49, Mar. 2010.

[32] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar. f4: Facebook's Warm BLOB Storage

System. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI' 14)*, 2014.

[33] S. V. Patil and G. A. Gibson. Scale and Concurrency of GIGA+: File System Directories with Millions of Files. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, FAST '11, 2011.

[34] K. Ren and G. Gibson. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *Proceedings of the USENIX Annual Technical Conference*, ATC '13, 2013.

[35] K. Ren, Q. Zheng, S. Patil, and G. Gibson. IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion. In *Proceedings of the Conference on High Performance Computing, Networking, Storage and Analysis*, SC '14, 2014.

[36] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In *Proceedings of the USENIX Annual Technical Conference*, ATC '00.

[37] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. In *ACM Transactions on Computer Systems*, '92.

[38] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, 2002.

[39] P. Schwan. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux Symposium*, volume 2003.

[40] M. A. Sevilla, I. Jimenez, N. Watkins, S. Finkelstein, J. LeFevre, P. Alvaro, and C. Maltzahn. Cudele: An API and Framework for Programmable Consistency and Durability in a Global Namespace. In *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium*, IPDPS '18, 2018.

[41] M. A. Sevilla, N. Watkins, C. Maltzahn, I. Nassi, S. A. Brandt, S. A. Weil, G. Farnum, and S. Fineberg. Mantle: A Programmable Metadata Load Balancer for the Ceph File System. In *Proceedings of the Conference on High Performance Computing, Networking, Storage and Analysis*, SC '15.

[42] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan. Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics. *Proceedings of the VLDB Endowment.*

[43] K. o. V. Shvachko. HDFS Scalability: The Limits to Growth.

[44] K. V. Shvachko, H. Kuang, S. Radia, and bert Chansler. The hadoop distributed file system. In *MSST2010*, Incline Village, NV, May 3-7 2010.

[45] S. Sinnamohideen, R. R. Sambasivan, J. Hendricks, L. Liu, and G. R. Ganger. A Transparently-Scalable Metadata Service for the Ursa Minor Storage System. In *USENIX ATC '10*, ATC '10, Boston, MA, June 23-25 2010.

[46] D. Terry. Replicated Data Consistency Explained Through Baseball. 56.

[47] A. Thomson and D. J. Abadi. CalvinFS: Consistent WAN Replication and Scalable Metadata Management for Distributed File Systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015.

[48] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu. Data Warehousing and Analytics Infrastructure at Facebook. In *Proceedings of the SIGMOD International Conference on Management of Data*, SIGMOD '10.

[49] F. Wang, H. S. Oral, G. M. Shipman, O. Drokin, D. Wang, and H. Huang. Understanding Lustre Internals. Technical report, Oak Ridge National Laboratory (ORNL); Center for Computational Sciences, 2009.

[50] S. A. Weil. *Ceph: Reliable, Scalable, and High-Performance Distributed Storage.* PhD thesis, University of California at Santa Cruz, December 2007.

[51] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI '06.

[52] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller. Dynamic Metadata Management for Petabyte-Scale File Systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '04.

[53] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and

B. Zhu. Scalable Performance of the Panasas Parallel File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, 2008.

[54] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable Performance of the Panasas Parallel File System. In *Proceedings of the 6th Conference on File and Storage Technologies*, FAST '08.

[55] L. Xiao, K. Ren, Q. Zheng, and G. A. Gibson. ShardFS vs. IndexFS: Replication vs. Caching Strategies for Distributed Metadata Management in Cloud Storage Systems. In *Proceedings of the Symposium on Cloud Computing*, SoCC '15.

[56] J. Xing, J. Xiong, N. Sun, and J. Ma. Adaptive and Scalable Metadata Management to Support a Trillion Files. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, 2009.

[57] A. B. Yoo, M. A. Jette, and M. Grondona. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science.

[58] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design & Implementation*, NSDI'12, 2012.

[59] Q. Zheng, K. Ren, and G. Gibson. BatchFS: Scaling the File System Control Plane

with Client-funded Metadata Servers. In *Proceedings of the Workshop on Parallel Data Storage*, PDSW '14.

[60] Q. Zheng, K. Ren, G. Gibson, B. W. Settlemyer, and G. Grider. DeltaFS: Exascale File Systems Scale Better Without Dedicated Servers. In *Proceedings of the Workshop on Parallel Data Storage*, PDSW '15, 2015.

[61] Y. Zhu, H. Jiang, J. Wang, and F. Xian. HBA: Distributed Metadata Management for Large Cluster-Based Storage Systems. *IEEE Transactions on Parallel Distribed Systems*, 19(6), June 2008.