Tintenfisch: File System Namespace Schemas and Generators

Michael A. Sevilla, Reza Nasirigerdeh, Carlos Maltzahn, Jeff LeFevre, Noah Watkins,
Peter Alvaro, Margaret Lawson*, Jay Lofstead*, Jim Pivarski[a]

University of California, Santa Cruz. {msevilla, rnasirig, carlosm, jlefevre, nmwatkin, palvaro}@ucsc.edu

**Sandia National Laboratories. {mlawso, gflofst}@sandia.gov, [a]Princeton University. pivarski@pinceton.edu*

1 Introduction

The file system metadata service is the scalability bottleneck for many of today's workloads [22, 2, 3, 4, 29]. Common approaches for attacking this "metadata scaling wall" include: caching inodes on clients and servers [8, 27, 13, 9, 31], caching parent inodes for path traversal [19, 21, 6, 30, 21], and dynamic caching policies that exploit workload locality [32, 34, 17]. These caches reduce the number of remote procedure calls (RPCs) but the effectiveness is dependent on the overhead of maintaining cache coherence and the administrator's ability to select the best cache size for the given workloads. Recent work reduces the number of metadata RPCs to 1 without using a cache at all, by letting clients "decouple" the subtrees from the global namespace so that they can do metadata operations locally [33, 24]. *Even with* this technique, we show that file system metadata is still a bottleneck because namespaces for today's workloads can be very large. The size is problematic for reads because metadata needs to be transferred and materialized.

The management techniques for file system metadata assume that namespaces have no structure but we observe that this is not the case for all workloads. We propose Tintenfisch, a file system that allows users to succinctly express the structure of the metadata they intend to create. If a user can express the structure of the namespace, Tintenfisch clients and servers can (1) compact metadata, (2) modify large namespaces more quickly, and (3) generate only relevant parts of the namespace. This reduces network traffic, storage footprints, and the number of overall metadata operations needed to complete a job.

Figure 1 provides an architectural overview: clients first decouple the file system subtree they want to operate on[1] then clients and metadata servers lazily generate subtrees as needed using a "namespace generator". The namespace generator is stored in the root inode of the decoupled subtree and can be used later to efficiently merge new metadata (that was not explicitly stated up front) into the global namespace. The fundamental insight is that the client and server both understand the final structure of the file system metadata. Our contributions:

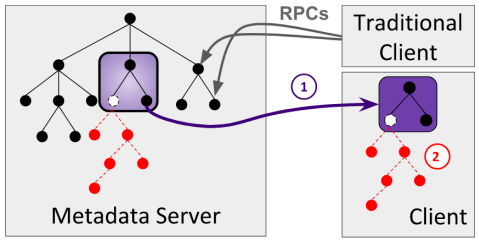


Figure 1: In (1), clients decouple file system subtrees and interact with their copies locally. In (2), clients and metadata servers generate subtrees, reducing network/storage usage and the number of metadata operations.

- observing namespace structure in high performance computing, high energy physics, and large fusion simulations (§2)
- based on these observations, we defined namespace schemas for categorizing namespaces and their amenability to compaction and generation (§3.1)
- a generalization of existing file system services to implement namespace generators that efficiently compact and generate metadata (§3.2)

2 Motivating Examples

We look at the namespaces for 3 large-scale applications. Each is from a different domain and this list is not meant to be exhaustive. Large lists are a problem in each of these domains, so building a file system with just general metadata (*e.g.*, extended attributes) would reduce the size of the metadata but the architecture would still suffer from managing a large number of names. To make

[1] This is not a contribution. This functionality and details on merging updates (*e.g.,* when to merge, how to merge, and how to manage conflicts) were presented in DeltaFS [33] and Cudele [24].

our results reproducible, this paper adheres to The Popper Convention [14] so experiments can be examined in more detail, or even re-run, by visiting the `[source]` link next to each figure.

2.1 High Performance Computing: PLFS

Checkpointing performs small writes to a single shared file but because file systems are optimized for large writes, performance is poor. PLFS [5] solved the checkpoint-restart problem by mapping logical files to physical files on the underlying file system. The solution targets N-1 strided checkpoints, where many processes write small IOs to offsets in the same logical file. Each process sequentially writes to its own, unshared data file in the hierarchical file system and records an offset and length in an index file. Reads aggregate index files into a global index file, which it uses as a lookup table for identifying offsets into the logical file.

Namespace Description: when PLFS maps a single logical file to many physical files, it deterministically creates the namespace in the backend file system. For metadata writes, the number of directories is dependent on the number of clients nodes and the number of files is a function of the number of client processes. A directory called a container is created per node and processes write data and index files to the container assigned to their host. So for a write workload (*i.e.* a checkpoint) the underlying file system creates a deep and wide directory hierarchy, as shown in Figure 2a. The `host*` directory and `data*/index` files (denoted by the solid red line) are created for every node in the system. The pattern is repeated twice (denoted by the dashed blue line) in the Figure, representing 2 additional hosts each with 1 process.

Namespace Size: Figure 2b scales the number of clients and plots the total number of files/directories (text annotations) and the number of metadata operations needed to write and read a PLFS file. The number of files is $2 \times$ (# of processes). So for 1 million processes each checkpointing a portion of a 3D simulation, the size of the namespace will be 2 million files. RPC-based approaches like IndexFS [21] have been shown to struggle with metadata loads of this size but decoupled subtree approaches like DeltaFS [33] report up to 19.69 million creates per second, so writing checkpoints is largely a solved problem.

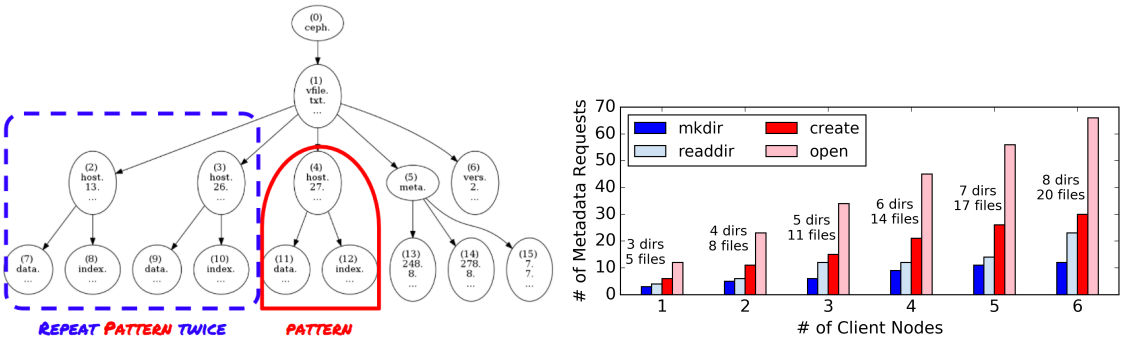
For reading a checkpoint, clients must coalesce index files to reconstruct the PLFS file. Figure 2b shows that the read metadata requests ("readdir" and "open") outnumber the create requests by a factor of $4\times$. Metadata read requests are notoriously slow [7, 10], so like create requests, RPCs are probably untenable. If the checkpoint had been written with the decoupled namespace approach, file system metadata would be scattered across clients so metadata would need to be coalesced before restarting the checkpoint. If the metadata had already been coalesced at some point they would still need to be transferred to the client. Regardless, both decoupled subtree scenarios require moving and materializing the file system subtree. Current efforts improve read scalability by reducing the space overhead of the index files themselves [12] and transferring index files after each write [11] but these approaches target the transfer and materialization of the index file data, not the index file metadata.

Takeaway: the PLFS namespace scales with the number of client processes so RPCs are not an option for reading or writing. Decoupling the namespace helps writes but then the read performance is limited by the speed of transferring file system metadata across the network to the reading client *in addition* to reading the contents of the index files themselves.

2.2 High Energy Physics: ROOT

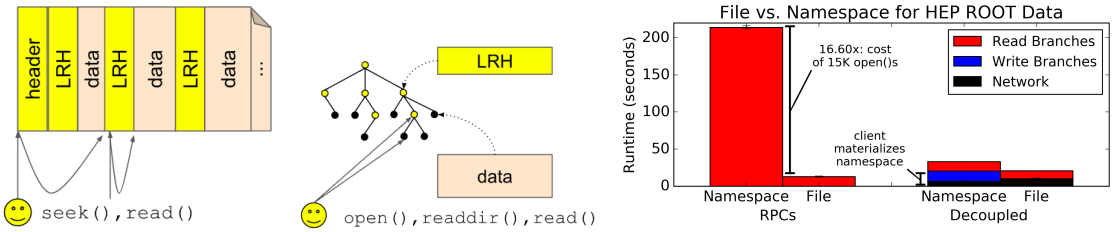
The High Energy Physics (HEP) community uses a framework called ROOT to manipulate, manage, and visualize data about proton-proton collisions collected at the large hadron collider (LHC). The data is used to resimulate phenomena of interest for analysis and there are different types of reconstructions each with various granularities. The data is organized as nested, object oriented event data of arbitrary type (*e.g.*, particle objects, records of low-level detector hit properties, etc.). Physicists analyze the data set by downloading interesting events, which are stored as a list of objects in ROOT files. ROOT file data is accessed by consulting metadata in the header and seeking to a location in the bytestream, as shown in Figure 3a. The ROOT file has both data and ROOT-specific metadata called Logical Record Headers (LRH). For this discussion, the following objects are of interest: a "Tree" is a table of a collection of events, listed sequentially and stored in a flat namespace; a "Branch" is a data container representing columns of a Tree; and "Baskets" are byte ranges partitioned by events and indexed by LRHs. Clients request Branches and data is transferred as Baskets; so Branches are the logical view of the data for users and Baskets are the compression, parallelization, and transfer unit. The advantages of the ROOT framework is the ability to (1) read only parts of the data and (2) easily ingest remote data over the network.

Namespace Description: the HEP community is running into scalability problems. The current effort is to integrate the ROOT framework with Ceph. But naive approaches such as storing ROOT files as objects in an object store or files in a file system have IO read amplification (*i.e.* read more than is necessary); storing as an object would pull the entire GB-sized blob and storing as



(a) PLFS file system tree (b) [source] PLFS metadata size and operations.

Figure 2: PLFS file system metadata. (a) shows that the namespace is structured and predictable; the pattern (solid line) is repeated for each hosts. In this case, there are three hosts so the pattern is repeated two more times. (b) shows that the namespace scales linearly with the number of clients. This makes reading and writing difficult using RPCs so decoupled subtrees must be used to reduce the number of RPCs.



(a) file approach (b) namespace approach (c) [source] ROOT metadata size and operations

Figure 3: ROOT file system metadata. (a) file approach: stores data in a single ROOT file, where clients read the header and seek to data or metadata (LRH); a ROOT file stored in a distributed file system will have IO read amplification because the striping strategies are not aligned to Baskets. (b) namespace approach: stores Baskets as files so clients read only data they need. In (c), "Namespace" is the runtime of reading a file per Basket and "File" is the runtime of reading a single ROOT file. RPCs are slower because of the metadata load and the overhead of pulling many objects. Decoupling the namespace uses less network (because only metadata and relevant Baskets get transferred) but incurs a metadata materialization overhead.

a file would pull more data than necessary because the file stripe size is not aligned to Baskets. To reduce IO read amplification the namespace approach [20] views a ROOT file as a namespace of data. Physicists ask for Branches, where each Branch can be made up of multiple sub-Branches (*i.e.* `Events/Branch0/Branch1`), similar to pathname components in a POSIX IO file name. The namespace approach partitions the ROOT file onto a file system namespace, as shown in Figure 3b. File system directories hold Branch metadata, files contain Baskets, and clients only pull Baskets they care about.

Namespace Size: storing this metadata in a file system would overwhelm most file systems in two ways: (1) too many inodes and (2) per-file overhead. To quantify (1), consider the Analysis Object Dataset which has a petabyte of data sets made up of a million ROOT files each containing thousands of Branches, corresponding to a billion files in the namespace approach. To quantify (2), the read and write runtime over six runs of replaying a trace of Branch access from the NTupleMaker application is shown in Figure 3c, where the *x*-axis is approaches for storing ROOT data. Using the namespace approach with RPCs is far slower because of the metadata load and because many small objects are pulled over the network. Although the file approach reads more data than is necessary since the stripe size of the file is not aligned to Baskets, the runtime is still 16.6× faster. Decoupling the namespace is much faster for the namespace approach but the cost of materializing file system metadata makes it slower than the file approach. Note that this is one (perhaps pessimistic) example workload; the ROOT file is 1.7GB and 65% of the file is accessed so the namespace approach might be more scalable for workloads that access fewer Baskets.

Takeaway: the ROOT namespace stores billions of files and we show that RPCs overwhelm a centralized metadata server. Decoupling the namespace helps writes but then the read performance is limited by (1) the speed of transferring file system metadata across the network and (2) the cost of materializing parts of the namespace that are not relevant to the workload.

2.3 Large Scale Simulations: SIRIUS

SIRIUS [15] is the Exascale storage system being designed for the Storage System and I/O (SSIO) initiative [23]. The core tenant of the project is application

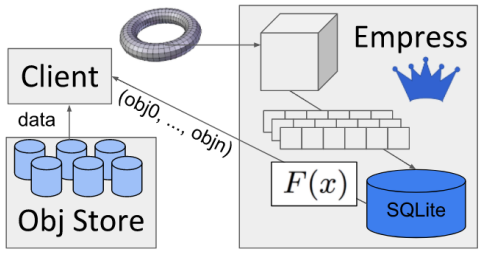


Figure 4: One potential EMPRESS design for storing bounding box metadata. Coordinates and user-defined metadata are stored in SQLite while object names are calculated using a partitioning function ($F(x)$) and returned as a list of object names to the client.

hints that allow the storage to reconfigure itself for higher performance using techniques like tiering, management policies, data layout, quality of service, and load balancing. SIRIUS uses a metadata service called EMPRESS [16], which is an SQLite instance that stores user-defined metadata for bounding boxes (*i.e.* a 3-dimensional coordinate space). EMPRESS is designed to be used at any granularity, which is important for a simulation space represented as a 3D mesh. By granularity, we mean that metadata access can be optimized per variable (*e.g.*, temperature, pressure, etc.), per timestamp, per run, or even per set of runs (which may require multiple queries). At this time, EMPRESS is single node but it is designed to scale-out via additional independent instances.

Namespace Description: the global space is partitioned into non-overlapping, regular shaped cells. The EMPRESS database has columns for the application ID, run ID, timestamp, variable name, feature name, and bounding box coordinates for these cells. Users can also add custom-defined metadata. The namespace we are referring to here is the list of objects containing simulation data associated to a bounding box (or row in the database). Variables affect how the space is partitioned into objects; temperature may be computed for every cell while pressure is computed for every n cells. For most simulations, there are a minimum of 10 variables.

Namespace Size: a back-of-the-envelope calculation for the number of object names for a single run is:

$$\frac{(\text{processes}) \times (\text{data/process}) \times (\text{variables}) \times (\text{timesteps})}{(\text{object size})}$$

We calculate $1 * 10^{12}$ (1 trillion) objects for a simulation space of $1\text{K} \times 1\text{K} \times 1\text{K}$ cells containing 8 byte floats. We use 1 million processes, each writing 8GB of data for 10 variables over 100 timesteps and an object size of 8MB (the optimal object size of Ceph's object store). The data per process and number of variables are scaled to be about 1/10 of each process's local storage space, so about 80GB. 100 timesteps is close to 1 timestep every 15 minutes for 24 hours.

As we integrate EMPRESS with a scalable object store, mapping bounding box queries to object names for data sets of this size is a problem. Clients query EMPRESS with bounding box coordinates[2] and EMPRESS must provide the client with a list of object names. One potential design is shown in Figure 4; coordinates for variables are stored in the database and object name lists are calculated using the $F(x)$ partitioning function at read time. The problem is that object name lists can be very large when applications query multiple runs each containing trillions of objects, resulting in long transfer times as the metadata is sent back to the client. Even after receiving the object name list, the client may need to manage and traverse the list, doing things like filtering for object names at the "edge" of the feature of interest.

Takeaway: SIRIUS stores trillions of objects for a single large scale simulation run and applications often access multiple runs. These types of queries return a large list of object names so the bottleneck is managing, transferring, and traversing these lists. The size of RPCs is the problem, not the number. POSIX IO hierarchical namespaces may be a good model for applications to access simulation data but another technique for handling the sheer size of these object name lists is needed.

3 Methodology: Compact Metadata

Namespace schemas and generators help clients and servers establish an understanding of the final file system metadata shape and size that eliminates the metadata overheads highlighted above.

3.1 Namespace Schemas

Namespace schemas describe the structure of the namespace. A "balanced" namespace means that subtree patterns (files per directory) are repeated and a "bounded" namespace means that the range of file/directory names can be defined *a-priori* (before the job has run but after reading metadata). Traditional shared file systems are designed for general file system workloads, like user home directories, which have an unbalanced and unbounded namespace schema because users can create any number of files in any pattern. PLFS has a balanced and bounded namespace because the distribution of files per directory is fixed (and repeated) and any subtree can be generated using the client hostnames and the number of processes. ROOT and SIRIUS are examples of unbalanced and bounded namespace schemas. The file per directory shape is not repeated (it is determined by application-specific metadata, LRH for ROOT or variables for SIR-

[2]Users usually track bounding boxes of interest by tagging features at write time.

For n processes on m servers:

```
# of dirs = m × mkdir()
# of file = 2 × n
# of file per dir = n/m
```

(a) Function generator for PLFS

```
local box require 'box2d'
for i=_x,_x+x do for j=_y,_y+y do
  if t>30 then
    obj_list.insert(box(x,y,z))
  else
    b0,b1,b2,=box.nsplit(4)
    obj_list.insert(b0,b1,b2)
end end end
return obj_list
```

(b) Code generator for SIRIUS

```
void recurseBranch(TObjArray *o){
  TIter i(o);
  for(TBranch *b=i.Next();
      i.Next()!=0;
      b=i.Next()){
    processBranch(b);
    recurseBranch(b->GetListOfBranches());
  }
}
```

(c) Code generator for HEP

Figure 5: Namespace generators for 3 motivating examples. The code generator in Figure 5c is coupled with a pointer generator.

IUS) but the range of file/directory names can be determined before the job starts.

3.2 Namespace Generators

A namespace generator is a compact representation of a namespace with a balanced or bounded schema that lets clients/servers generate file system metadata. Next we discuss 3 example namespace generators.

Formula Generator: takes domain-specific information as input and produces a list of files and directories. For example, PLFS creates files and directories based on the number of clients, so administrators can use the formula in Figure 5a, which takes as input the number of processes and hosts in the cluster and outputs the number of directories, files, and files per directory. The namespace drawn in Figure 2a can be generated using an input of 3 hosts each with 1 process.

Code Generator: gives users the flexiblity to write programs that generate the namespace. This is useful if the logic is too complex to store as a formula or requires external libraries to interpret metadata. For example, SIRIUS constructs the namespace using domain-specific partitioning logic written in Lua. Figure 5b shows how the namespace can be constructed by iterating through bounding box coordinates and checking if a threshold temperature is eclipsed. If it is, extra names are generated using the `box2d` package. Although the partitioning function itself is not realistic, it shows how code generators can accommodate namespaces that are complex and/or require external libraries.

Pointer Generator: references metadata in scalable storage and avoids storing large amounts of metadata in inodes, which is a frowned upon in distributed file system communities [1]. This is useful if there is no formal specification for the namespace. For example, ROOT uses self-describing files so headers and metadata need to be read for each ROOT file. A code generator is insufficient for generating the namespace because all necessary metadata is in objects scattered in the object store. A code generator containing library code for the ROOT framework *and* a pointer generator for referencing the input to the code can be used to describe a ROOT file system namespace. Figure 5c shows a code generator example where clients requesting Branches follow the pointer generator (not pictured) to objects containing metadata. An added benefit is that Tintenfisch can lazily construct parts of the namespace as needed, avoiding the inode problem discussed in §2.2.

3.3 Discussion

We designed these generators by matching the patterns of the namespace to the application source code. Often times, we had to consult documentation or the developers themselves to create this mapping; automatically creating generators based on source code analysis or namespace traversals is left as future work.

Advantages: using generators, Tintenfisch clients and servers improve the read performance of large file system metadata jobs by *compacting metadata*, which speeds up network transfers and reduces the storage footprint of metadata. Metadata compaction also gives clients/servers the ability to *modify large namespaces*. For the examples in Section §2: if a PLFS namespace was constructed with 1 million processes, scaling to 2 million processes only requires sending a new input to the formula generator; ROOT Branches can be added to the namespace by changing the metadata referenced by the pointer generator; and if SIRIUS objects need to be repartitioned, only the logic in the code generator needs to be updated. Metadata compaction also gives clients/servers the ability to *generate relevant parts of the namespace* because only a fraction of the metadata is needed. This is especially important for the SIRIUS use case, which has very large namespaces that are often pruned with prefixes.

Disadvantages: introducing administrator-level functionality into the file system may jeopardize security and correctness. This work shares many of the risks of programmable storage [25, 26], such as introducing generators with bad logic (*e.g.*, endless looping of names, nondeterministic naming, and creating corrupted names) or malicious intent. To a certain extent, the environment we choose may help sandbox these negative behaviors (*e.g.* with something like Lua's sandbox features) but these issues are largely left as future work.

Generality: our generator types work well for our use cases, but this is not an exhaustive list. We only argue that our generators work well for namespaces that are balanced or bounded (not mutually exclusive), so any workload that produces this structure should benefit. Although the generators themselves may not generalize to other namespace schemas, the approach of compacting metadata should work for workloads that modify or read the namespace with definable (or "express-able") access pattern. Similarly, the approach should also work for non-POSIX IO compliant namespaces, such as network ones, as long as the namespace has structure.

Current Implementation: Tintenfisch is built on Cudele [24] so a centralized, globally consistent metadata service can decouple subtrees and clients can do metadata IO locally with the consistency/durability semantics they require. Namespace generators are integrated into file system metadata servers and clients instead of the application itself because we find namespace schemas to be common across domains and use cases. The generator itself is stored in the directory inode of the decoupled subtree using a "file type" interface [28] and our prototype is built using a programmable storage approach [25, 18].

4 Conclusion

Contrary to common belief, global file system namespaces can be scalable if they are given enough domain-specific knowledge. We show that many of today's applications are specialized, so they have regular, large namespaces. As a result, the file system should be changing its internal mechanisms to leverage the bounded and balanced nature of these namespaces to optimize metadata performance. Namespace schemas and generators solve many file system mtadata read problems because clients and servers avoid exchanging large lists. Our examples benefit from *metadata compaction*, which speeds up network/storage overheads and gives clients and servers the ability to *modify large namespaces* and *generate relevant parts of the namespace*.

Acknowledgments

We thank the HotStorage reviewers for their helpful suggestions. This work is supported by the Center for Research in Open Source Software (cross.soe.ucsc.edu), the U.S. Department of Energy's Office of Science, under the grant number DE-SC0016074, and the NSF, under the award number 1450488.

References

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

[2] ABAD, C. L., LUU, H., LU, Y., AND CAMPBELL, R. Metadata Workloads for Testing Big Storage Systems. Tech. rep., Citeseer, 2012.

[3] ABAD, C. L., LUU, H., ROBERTS, N., LEE, K., LU, Y., AND CAMPBELL, R. H. Metadata Traces and Workload Models for Evaluating Big Storage Systems. In *Proceedings of the International Conference on Utility and Cloud Computing*, UCC '12.

[4] ALAM, S. R., EL-HARAKE, H. N., HOWARD, K., STRINGFELLOW, N., AND VERZELLONI, F. Parallel I/O and the Metadata Wall. In *Proceedings of the Workshop on Parallel Data Storage*, PDSW '11.

[5] BENT, J., GIBSON, G., GRIDER, G., MCCLELLAND, B., NOWOCZYNSKI, P., NUNEZ, J., POLTE, M., AND WINGATE, M. PLFS: A Checkpoint Filesystem for Parallel Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09.

[6] BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND XUE, L. Efficient Metadata Management in Large Distributed Storage Systems. In *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies* (2003), MSST '03.

[7] CARNS, P., LANG, S., ROSS, R., VILAYANNUR, M., KUNKEL, J., AND LUDWIG, T. Small-file Access in Parallel File Systems. In *Proceedings of the Symposium on Parallel and Distributed Processing*, IPDPS '09.

[8] DEPARDON, B., LE MAHEC, G., AND SÉGUIN, C. Analysis of six distributed file systems. Tech. rep., French Institute for Research in Computer Science and Automation, 2013.

[9] DEVULAPALLI, A., AND WYCKOFF, P. File creation strategies in a distributed metadata file system. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International* (2007), IEEE, pp. 1–10.

[10] ESHEL, M., HASKIN, R., HILDEBRAND, D., NAIK, M., SCHMUCK, F., AND TEWARI, R. Panache: A Parallel File System Cache for Global File Access. In *Proceedings of the Conference on File and Storage Technologies*, FAST '10.

[11] GRIDER, G. If the plfs index is still too large too efficiently read/distributepull out the hammerplfs collectives, reduces plfs index to nearly nothingdnabling strategy for optimizing reads and active analysis.

[12] HE, J. Io acceleration with pattern detection.

[13] HILDEBRAND, D., AND HONEYMAN, P. 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* (2005), MSST '05.

[14] JIMENEZ, I., SEVILLA, M. A., WATKINS, N., MALTZAHN, C., LOFSTEAD, J., MOHROR, K., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. The Popper Convention: Making Reproducible Systems Evaluation Practical. In *Proceedings of the International Parallel and Distributed Processing Symposium Workshop*, IPDPSW '17.

[15] KLASKY, S. A., ABBASI, H., AINSWORTH, M., CHOI, J., CURRY, M., KURC, T., LIU, Q., LOFSTEAD, J., MALTZAHN, C., PARASHAR, M., PODHORSZKI, N., SUCHYTA, E., WANG, F., WOLF, M., CHANG, C. S., CHURCHILL, M., AND ETHIER, S. Exascale Storage Systems the SIRIUS Way. *Journal of Physics: Conference Series*.

[16] LAWSON, M., ULMER, C., MUKHERJEE, S., TEMPLET, G., LOFSTEAD, J. F., LEVY, S., WIDENER, P. M., AND KORDENBROCK, T. Empress: Extensible Metadata Provider for Extreme-Scale Scientific Simulations. In *Proceedings of the Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, PDSW '17.

[17] LI, W., XUE, W., SHU, J., AND ZHENG, W. 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* (2006), MSST '06.

[18] OLDFIELD, R. A., WARD, L., RIESEN, R., MACCABE, A. B., WIDENER, P., AND KORDENBROCK, T. Lightweight I/O for Scientific Applications. In *International Conference on Cluster Computing*, Cluster Computing '06.

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

[20] PIVARSKI, J. How to make a petabyte ROOT file: proposal for managing data with columnar granularity.

[21] REN, K., ZHENG, Q., PATIL, S., AND GIBSON, G. 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* (2014), SC '14.

[22] ROSELLI, D., LORCH, J. R., AND ANDERSON, T. E. A Comparison of File System Workloads. In *Proceedings of the USENIX Annual Technical Conference*, ATC '00.

[23] ROSS, R., AND ET. AL. Storage Systems and Input/Output to Support Extreme Scale Science. In *Report of the DOE Workshops on Storage Systems and Input/Output*.

[24] SEVILLA, M. A., JIMENEZ, I., WATKINS, N., FINKELSTEIN, S., LEFEVRE, J., ALVARO, P., AND MALTZAHN, C. 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* (2018), IPDPS '18.

[25] SEVILLA, M. A., WATKINS, N., JIMENEZ, I., ALVARO, P., FINKELSTEIN, S., LEFEVRE, J., AND MALTZAHN, C. Malacology: A Programmable Storage System. In *Proceedings of the European Conference on Computer Systems* (2017), EuroSys '17.

[26] SEVILLA, M. A., WATKINS, N., MALTZAHN, C., NASSI, I., BRANDT, S. A., WEIL, S. A., FARNUM, G., AND FINEBERG, S. Mantle: A Programmable Metadata Load Balancer for the Ceph File System. In *Proceedings of the Conference on High Performance Computing, Networking, Storage and Analysis*, SC '15.

[27] SINNAMOHIDEEN, S., SAMBASIVAN, R. R., HENDRICKS, J., LIU, L., AND GANGER, G. R. A Transparently-Scalable Metadata Service for the Ursa Minor Storage System. In *USENIX ATC '10* (Boston, MA, June 23-25 2010), ATC '10.

[28] WATKINS, N., MALTZAHN, C., BRANDT, S., AND MANZANARES, A. DataMods: Programmable File System Services. In *PDSW'12* (2012).

[29] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI '06.

[30] WEIL, S. A., POLLACK, K. T., BRANDT, S. A., AND MILLER, E. L. Dynamic Metadata Management for Petabyte-Scale File Systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '04.

[31] WELCH, B., UNANGST, M., ABBASI, Z., GIBSON, G., MUELLER, B., SMALL, J., ZELENKA, J., AND ZHU, B. Scalable Performance of the Panasas Parallel File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (2008), FAST'08.

[32] XING, J., XIONG, J., SUN, N., AND MA, J. Adaptive and Scalable Metadata Management to Support a Trillion Files. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (2009), SC '09.

[33] ZHENG, Q., REN, K., GIBSON, G., SETTLEMYER, B. W., AND GRIDER, G. DeltaFS: Exascale File Systems Scale Better Without Dedicated Servers. In *Proceedings of the Workshop on Parallel Data Storage* (2015), PDSW '15.

[34] ZHU, Y., JIANG, H., WANG, J., AND XIAN, F. HBA: Distributed Metadata Management for Large Cluster-Based Storage Systems. *IEEE Transactions on Parallel Distribed Systems 19*, 6 (June 2008).