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. pivarski@pinceton.edu*

1 Introduction

The file system metadata service is the scalability bottleneck for many of today's workloads [20, 2, 3, 4, 25]. Common approaches for attacking this "metadata scaling wall" include: caching inodes on clients and servers [7, 24, 11, 8, 27], caching parent inodes for path traversal [17, 19, 6, 26, 19], and dynamic caching policies that exploit workload locality [28, 30, 15]. 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 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 [29, 22]. *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 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

[1]This is not a contribution as it was presented in [22].

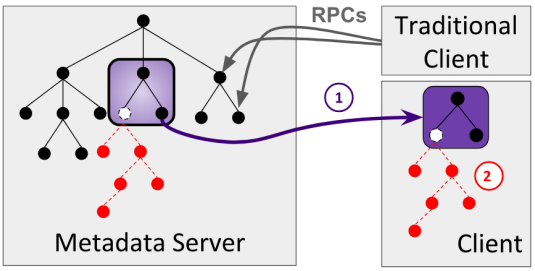


Figure 1: In (1), clients decouple file system subtrees and interact with their copies locally. In (2), clients and metadata servers generate subtrees, reducing network/storage usage and the number of metadata operations.

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:

- observing namespace structure in high performance computing, high energy physics, and large simulations (§2)
- based on these observations, we defined namespace schemas for categorizing namespaces and their amenability to compaction and generation (§4.1)
- a generalization of existing file storage system services to implement namespace generators that compact, modify, and generate metadata (§4.2)

2 Motivating Examples

We look at the namespaces of 3 applications. Each is from different domains and this list is not meant to be exhaustive. Similar organizations exist for many domains, even something as distant as the mail application on a Mac. To highlight the scalability challenges for file system metadata management, we focus on large

scale systems in high performance computing, high energy physics, and large scale simulations. Large lists represent common problems in each of these domains. To make our results reproducible, this paper adheres to The Popper Convention [12] 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 2. 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 3 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 a 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 have been shown to struggle with metadata loads of this size but decoupled namespace approaches like DeltaFS 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 2 shows that the read metadata requests ("readdir" and "open") outnumber the create requests by a factor of $4\times$ making RPCs even more untenable. Even worse, 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. Current efforts improve read scalability by reducing the space overhead of the index files themselves [10] and transferring index files after each write [9] but these approaches do not reduce the overhead of transferring file system metadata.

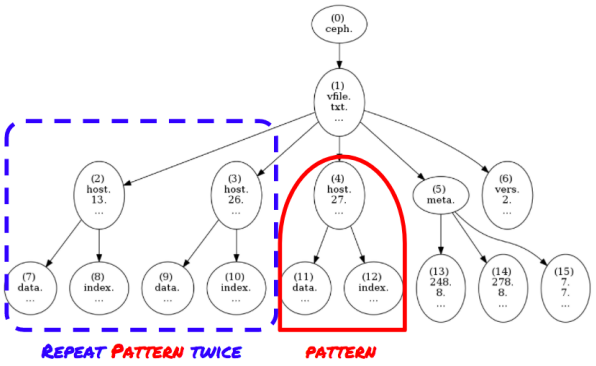


Figure 2: The PLFS file system namespace is structured and predictable; the pattern (solid line) is repeated for each hosts. In this case, there are three hosts so the pattern is repeated two more times.

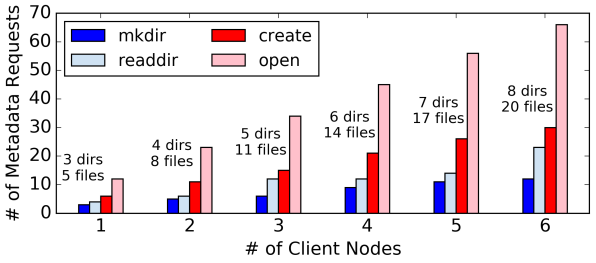
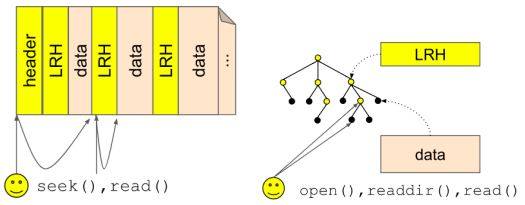


Figure 3: [source] For PLFS, the file system namespace scales linearly with the number of clients. This makes reading and writing difficult using RPCs so decoupled namespace must be used to reduce the number of RPCs.

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 because metadata is coalesced at the reading client.

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



(a) file approach (b) namespace approach

Figure 4: (a) file approach: stores data in a single ROOT file, where clients read the header and seek to data/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 in the file system namespace so clients read only the data they need.

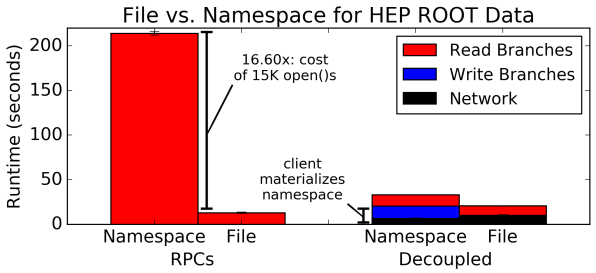


Figure 5: [source] "Namespace" is the runtime of reading a file per Basket and "File" is the runtime of reading a single ROOT file. RPCs are slower because of the metadata load and the overhead of pulling many objects. Decoupling the namespace has less network (because only metadata and relevant Baskets get transferred) but the benefit is offset by the cost of namespace materialization.

different types of reconstructions each with various granularities. The data is organized as nested, object oriented event data and the length of the runs (and thus the number of events) are of arbitrary length and type (*e.g.*, particle objects, records of low-level detector hit properties, etc.). Reconstruction takes detector conditions (*e.g.*, alignment, position of the beam, etc.) as input. Data is streamed from the LHC into large immutable datasets, stored publicly in data centers around the world. Physicists analyze the dataset 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 4a. Scattered in the ROOT file is both data and ROOT file 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 tranfer 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 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 [18] views a ROOT file as a namespace of data. Physicists ask for Branches, where each Branch can be made up of multiple subBranches (*i.e.* `Events/Branch0/Branch1`), similar to pathname components in a file system file name. The namespace approach partitions the ROOT file onto a file system namespace. As shown in Figure 4b, file system directories hold Branch metadata and files contain Baskets. Clients only pull Baskets they care about, which prevents IO read amplification.

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 Oject 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 5, 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. We 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 the speed of

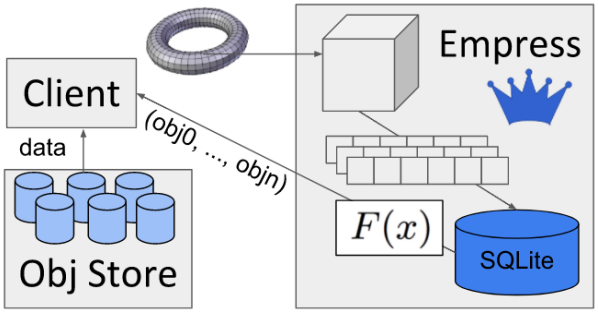


Figure 6: The SIRIUS project uses EMPRESS to store metadata for bounding boxes in a 3D torus. Bounding box coordinates and a list of object names are stored in SQLite. The object names are calculated using an objecter function, labeled $F(x)$ above. For reads, clients query EMPRESS for the object list before reading from the object store.

transferring file system metadata across the network as clients read Baskets. Read performance is also limited by the cost of materializing and scanning parts of the namespace that are not relevant to the workload.

2.3 Large Scale Simulations: SIRIUS

SIRIUS [13] is the Exascale storage system being designed for the Storage System and I/O (SSIO) initiative [21]. The core tenant of the project is application hints that allow the storage to reconfigure itself for higher performance using tehcniques like tiering, management policies, data layout, quality of service, and load balancing. SIRIUS uses a metadata service called EMPRESS [14], which is a query-able 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). EMPRESS is also designed to scale-out via additional independent instances, where one client per node queries the entire space of interest by contacting EMPRESS servers, coalesces results, and distributes them using MPI messages.

Namespace Description: the seven columns in the database are six coordinates, which are integers representing the contents at a location in the global space, and a string representing the object name. The global space is partitioned into non-overlapping, regular shaped cells. Each row in the database is duplicated for each variable in the simulation because variables may have a different partitioning of the global space; for example temperature is computed for every cell while pressure is computed for every n cells. Figure 6 only has three variables, represented by the rows of boxes but most simulations will have a minimum of 10 variables.

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

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

$$= \frac{(1*10^6) \times (8*10^9) \times (10) \times (100)}{(8*10^6)} = 1*10^{12} \text{ objects}$$

These values are 1 million processes, each writing 8GB of data for 10 variables; the data per process and number of variables are scaled to be about 1/10 of each processes local storage space, so about 80GB. 100 timesteps is close to 1 timestep every 15 minutes for 24 hours. This represents a simulation space of $1\text{K} \times 1\text{K} \times 1\text{K}$ cells containing 8 byte floats. The 8MB object size is the optimal object size in RADOS.

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 (possibly overlapping) object names. These lists can be very large when applications query multiple runs each containing trillions of objects. One option is to calculate object names at write time and store them with bounding box coordinates but this would result in a large storage footprint and long transfer times for sending the object name list to the client. For distributed EMPRESS, the storage footprint may not be as much of an issue but the trade-off is transferring parts of the object name list over the network as reads must be centralized at the designated read process on each client. Another option is shown in Figure 6; coordinates for variables (represented by the 3 rows of grey boxes) are stored in the database and object name lists are calculated using the $F(x)$ function at read time and sent back to the client. This solves the storage footprint problem but not the long transfer times for sending the large object name list back to the client. Even after receiving the object name list, the client may need to do more 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

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

simulation data but another technique for handling the sheer size of these object name lists is needed.

3 Methodology: Compact Metadata

4 sec:methodology

For three domain-specific applications and use cases, we have identified scalability challenges because of the size of the namespace. Tintensfisch compacts metadata by defining namespace schemas and proposing namespace generators. Namespace schemas and generators help clients and servers establish an understanding of the final file system metadata shape and size (*i.e.* a a "generation" contract) that eliminates metadata overheads.

4.1 Namespace Schemas

Namespace schemas describes the structure of the namespace. A "balanced" namespace means that subtree patterns (*i.e.* files per directory) are repeated and a "bounded" namespace means that the range of file/directory names can be defined *a-priori*[3]. In the previous section, we observe three types of namespace schemas (1) unbalanced and unbounded, (2) balanced and bounded, and (3) unbalanced and bounded.

General file system workloads, like user home directories, have an unbalanced and unbounded namespace schema because users can create any combination of files per directory and the range of file/directory names is unknown *a-priori*. Traditional shared file systems are designed to handle these types of workloads. PLFS is an example of a balanced and bounded namespace because the distribution of files per directory is fixed (and repeated) and any subtree can be generated just using the client hostname and number of processes. The ROOT and SIRIUS use cases are examples of the unbalanced and bounded namespace schema. The file per directory shape is not repeated (it is determined by application-specific metadata, LRH metadata for ROOT and variable names like temperature or pressure for SIRIUS) but the range of file/directory names can be determined before the job starts.

4.2 Namespace Generators

A namespace generator is a compact representation of a namespace that allows clients and servers to generate file system metadata. Namespace generators can be used for a namespace schema that is balanced or bounded (not mutually exlusive). Tintenfisch is built on Cudele [22] so a centralized, globally consistent metadata service (either a single metadata server or a cluster of active-active metadata servers) provides clients with the root inode of the subtree of interest and clients can do metadata IO locally with the consistency/durability semantics they require. This concept is similar to LWFS [16], which supplied a core set of functionality needed by all applications needing to do IO with the assumption that applications would bolt-on any functionality they needed later. In Tintenfisch, the namespace generators are stored in the directory inode of the root of the subtree that the client cares about. We use the "file type" interface from the Malacology [23] project to facilitate this domain-specific functionality. Next we discuss three example namespace generators: formula, code, and pointers.

Formula Generator: a formula that generates the file system namespace is stored in the inode. This formula takes domain-specific information as input and produces a list of files and directories. For example, because PLFS clients deterministically create files and directories based on the number of clients, Tintenfisch can use a formula generator like the one in Figure 7a. The function takes as input the number of processes and hosts in the cluster and outputs the number of directories, the number of files, and the number of files per directory. For the example, the namespace drawn in Figure 2 can be generated with the formula in Figure 7a using an input of 3 hosts each with 1 process. The output is 3 directories and 6 files, with 2 files per directory. With this namespace generator, clients can open just the container inode and then compute and access its contents without `lookup()` and `open()` RPCs to a centralized metadata service.

Code Generator: sometimes the namespace generator logic is too complex to store as a single function or requires external libraries to interpret metadata. For example, the SIRIUS use case constructs the namespace using domain-specific partitioning logic written in Lua (for sandboxing purposes). Tintenfisch provides a code generator that gives users the flexiblity to write programs that generate the namespace.

A code generator for the SIRIUS project is shown in Figure 7b. A namespace is constructed by iterating through the bounding box coordinates and checking if a threshold temperature is eclipsed. If it is, then extra names are generated using the `box2d` Lua package. Although the partitioning function itself is not realistic, it shows how code generators can accomodate domain-specific data layout policies that are complex and/or require external libraries.

Pointer Generator: sometimes there is no formal specification for the namespace. For example, the ROOT framework uses self-describing files so headers and metadata need to be read for each ROOT file. In these scenarios, a code generator is insufficient for generating

[3] By *a-priori* we mean before the job has run but after metadata has been read.

For n processes on m servers:

```
# of dirs = m × mkdir()
# of file = 2 × n × m
# 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 b0,b1=box.nsplit(2)
  else b0,b1,b2,=box.nsplit(4)
  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 generators for HEP

Figure 7: Namespace generators subtrees for 3 motivating examples.

the file system namespace because all necessary metadata is in objects scattered in the object store. Pointer generators reference data in scalable storage and avoids storing large amounts of metadata in inodes, which is a frowned upon in distributed file systems like CephFS [1].

For example, 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. The pointer generator would point to objects in the object store that contain the necessary metadata for constructing the file system namespace. This is shown in Figure 7c; clients requesting Branches would follow the pointer generator to the objects containing metadata and would read them to locate Baskets using the code generator. An added benefit to this solution is that Tintenfisch can lazily construct parts of the namespace as needed, which avoids the problem of running out of inodes discussed.

5 Conclusion

Namespace schemas and generators solve the read problems from Section §2 because clients and servers exchange namespace generators instead of the file system metadata in its entirety. This means clients/servers can (1) compact metadata, (2) modify large namespaces more quickly and (3) generate relevant parts of the namespace, and (3) modify large namespaces more quickly. (1) benefits all use cases because it speeds up network transfers and reduces the storage footprint of metadata. All use cases also benefit from (2): PLFS client processes can be dynamically added/removed by changing the input to the formula so if the original namespace was constructed with 1 million processes, scaling to 2 million processes only requires sending a new input to the formula instead of transferring the metadata for 1 million additional files; ROOT Branches can be added to the namespace by changing the metadata referenced by the pointer; and SIRIUS object naming can be altered by changing the logic in the code schema so if the objects need to be partitioned into different sized blobs, only the new code schema with updated stripe size logic needs to be sent to clients and servers instead of a list of billions of new object names. SIRIUS and ROOT benefit from (3), where long lists are the bottleneck in scenarios where only a fraction of the metadata is needed to complete the job.

Global file systems can be scalable if programmed correctly. For example, the following notions are out-dated:

- robust so they are fast... but we show that today's apps are so large that we need to specialized storage systems
- general because they have been around for so long... but we show that most apps don't need fs metadata
- subject to data IO performance... but we show that metadata is slow

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] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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*.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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*.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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).