Tintenfisch: Namespace Schemas for Lazy File System Metadata Generation

Michael A. Sevilla
University of California, Santa Cruz
msevilla@soe.ucsc.edu

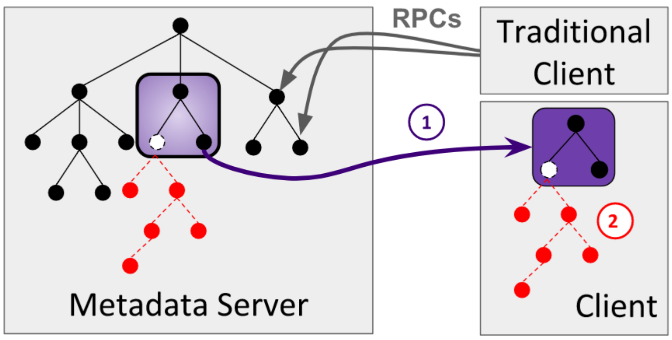


Figure 1: In (1), clients decouple file system subtrees and interact with their copies locally for high performance. In (2), clients and metadata servers generate subtrees using "namespace schemas", thus reducing RPC load.

ABSTRACT

blah blah

ACM Reference format:
Michael A. Sevilla. 2018. Tintenfisch: Namespace Schemas for Lazy File System Metadata Generation. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17)*, 6 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

We propose Tintenfisch, a file system that allows users to succinctly express the structure and patterns of the metadata they intend to create. They can also merge new metadata (that they did not explicitly state up front) into the global namespace. Using this semantic knowledge, Tintenfisch can optimize performance by reducing the number of RPCs needed for:

- metadata writes because clients/servers can create metadata independently
- metadata reads because clients can construct metadata and pull data directly from the object store

The fundamental insight is that the client and server both understand the final structure of the file system metadata so there is no need to communicate. The idea uses concepts from decoupled namespaces [13, 14] and patterned IO [6] to build a scalable global namespace. Less work is done on the metadata servers and clients pick up some of the metadata load. This approach is similar to predicate push down in databases, where structure is described using XML or JSON and pushed as predicates to the lower storage layers [4]. It is our hope that Tintenfisch will also be able to change the representation or structure of the file system metadata according to the file type or workload. We have the following contributions:

- a prototype implementation that leverages metadata compaction and reduces RPC amplification to improve performance
- structured and unstructured namespaces, a paradigm that helps applications optimize performance by conveying semantic knowledge to the storage system.

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, as similar organizations exist for many domains, even something as distant as the mail application on a Mac. For scalability reasons, we focus on large scale systems in high performance computing (HPC) and high energy physics (HEP).

Experimental Setup

We benchmark over Ceph (Jewel version) with n object storage daemons (OSDs), 1 metadata server (MDS), 1 monitor server (MON), and 1 client. We use 3 OSDs because it sustains 16 concurrent writes of 4MB at 600MB/s for 2 minutes. 250MB/s is the max speed of the SSDs, so the setup achieves 80% of the cluster SSD bandwidth.

We use CephFS, the POSIX-compliant file system that uses Ceph's RADOS object store [11], as the underlying file system. This analysis focuses on the file system metadata RPCs between the client and metadata server and does not include the RPCs needed to write and read actual data. CephFS uses a cluster of metadata servers to service file system metadata requests [12] and to characterize the workload, we instrumented the metadata server to track the number of each request type[1].

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. To be specific, it easier for applications to write checkpoints to a single file with small, unaligned, writes of varying length varying write (N-1) but general-purpose distributed file systems are designed for writes to different files (N-N).

[1]This code was merged into the Ceph project.

Conference'17, Washington, DC, USA
2018. 978-x-xxxx-xxxx-x/YY/MM...$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

The general problem is that the application understands the workload but it cannot communicate a solution to the storages system. The common solution is to add middleware (i.e. software that sits between the application and the storage system) to translate the data into a format the storage system performs well at.

The problem is that the underyling file system cannot keep up with the metadata load imposed by PLFS. PLFS creates an index entry for every write, which results in large per-processes tables [5]. This makes reading or scanning a logical file slow because PLFS must construct a global index by reading each process's local index. This process incurs a `readdir` and, if the file is open by another process, an additional `stat()` because metadata cannot be cached in the container [1].

2.1.1 System Architecture. PLFS [1] 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. The key insight of PLFS is that general purpose file systems perform well for applications that use N-N checkpoints and that the N-1 strided checkpoint style can be transformed with a thin interposition layer. To map offsets in the logical file to physical files each process maintains an index of {logical offset, physical offset, length, physical block id}.

PLFS maps an application's preferred data layout into one that the file system performs well on. Each process appends writes to a different data file in the hierarchical file system and records an offset and length are recorded in an index file. Reads aggregate per-process index files into a global index file, which it uses as lookup table for logical file.

2.1.2 Namespace Description. When PLFS maps a logical file to many physical files, it deterministically creates the file system namespace in the backend file system. For n processes on m servers:

`# of dirs` = $m \times$ `mkdir()`
`# of file` = $2n \times$ `create()`$[+2n \times$ `lookup()`$]$
`# of file per dir` = n/m

For metadata writes, the number of directories is dependent on the number of PLFS servers. When a file is created in a PLFS mount, a directory called a container is created in the underlying file system. Inside the container, directories are created per server resulting in m requests.

The namespace structure for 3 processes writing to variable offsets in a single PLFS file is 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.

2.1.3 Overhead: Processing RPCs. For writing, the number of files is dependendent on the number of PLFS processes. Processes write data and index files to the directory assigned to their server. The number of requests, which can range from $2n$ to $4n$, is a function of files per directory. If each server only has one process, then the number of file create requests will be $2n$ because each process can cache the directory inode and create files in one request. With multiple processes per server, clients write to the same directory and the number of requests doubles because processes cannot cache the directory inode.

For metadata reads, transforming write IO in this way has space and read overheads. In PLFS, this is a problem because index files need to be coalesced on reads. Patterned PLFS [6] reduces the space overheads by storing formulas, instead of index files, to represent write behavior. Diddlings [5] transfers index files after each write to absorb the transfer overheads up front. While these approaches help alleviate read overheads, they do not reduce the file system metadata load, which is the real problem. Reading the index file still requires a file system metadata operation.

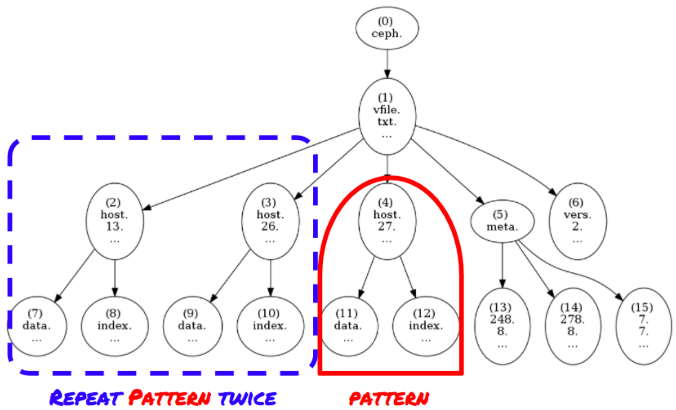


Figure 2: The PLFS file system namespace is structured and predictable; the pattern (solid line) is repeated for each hosts. In this case, there are three hosts so the pattern is repeated two more times.

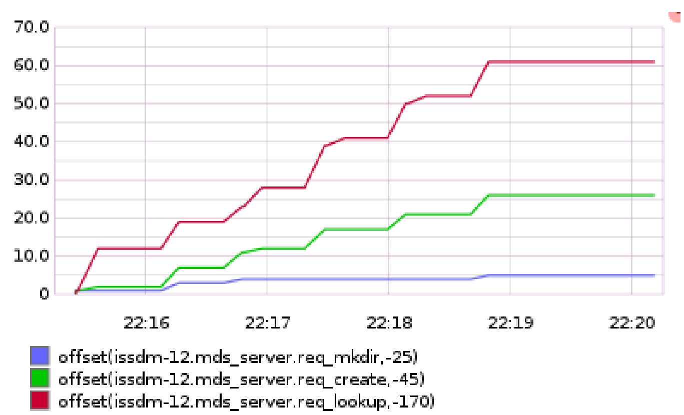


Figure 3: The requests scale linearly with the number of clients. `lookup()` requests dominate because clients share the root and must do path traversal.

- read file: $2n \times$ `open()`

Takeaway. The scalability of PLFS is limited by the request throughput of the metadata server in the underlying distributed file system. At large scale, the metadata server will need to service many requests.

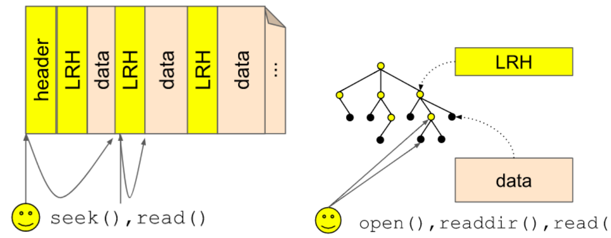
2.2 High Energy Physics: ROOT

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 re-simulate phenomena of interest for analysis and there are different types of reconstructions each with various granularities. The data is organized as nested and 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 stored in ROOT files.

2.2.1 System Architecture. A ROOT file is a list of objects and 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. Other objects, such as "Keys", "Directories", and "Leaves", contain HEP-specific metadata. 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. In summary, ROOT files are self-describing files containing data located with metadata and serialized/de-serialized with the ROOT framework.

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. The disadvantage is that the ROOT framework has no formal specification. Much of the development was done at CERN in parallel with other HPC ventures. As a result, the strategies are similar to techniques used in HDF5, Parquet, and Avro.

2.2.2 Namespace Description. Currently, the ROOT file can be viewed as a namespace of data. At the top of the namespace are Keys, each containing pointers to groups of Branches. For example, "MetaData" has data about the run and "Events" has all the proton-proton activity. 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.



(a) file approach **(b) namespace approach**

Figure 4: (a) shows how the file approach for handling HEP data stores everything in a single ROOT file, where the client reads the header and seeks to metadata (LRH). For ROOT files stored in distributed file systems reads will have amplification because the system's striping strategies are not aligned to Baskets. (b) shows how the namespace approach stores Baskets as files in the file system namespace so clients read only the data they need.

Unfortunately, the HEP community is running into scalability problems. The current effort is to integrate the ROOT framework with Ceph. Storing ROOT files as objects in an object store has overhead when users pull the entire GB-sized blob to their laptop. This model is a mismatch to the workload, which reads metadata and Branches a-la-cart. A new effort, which we call the namespace approach, is attempting to partition the ROOT file onto a file system namespace. As shown in Figure 4b, file system directories hold Branch metadata and files contain Baskets. With this approach clients only pull baskets they care about, which prevents read amplification (*i.e.* reading more than is necessary).

2.2.3 Overheads. We benchmark the write and read overhead of storing HEP data with the following approaches:

(1) file approach: one object in an object store
(2) file approach: one file
(3) namespace approach: many files

The file approaches are deployed without any changes to the ROOT framework. For the namespace approach, HEP-specific metadata is mapped onto the file system namespace. In CephFS, Baskets are stored in Ceph objects and the Branch hierarchy is managed by the metadata server. Clients contact the metadata server with a Branch request, receive back the Branch hierarchy necessary to name the Ceph object containing the Basket as well as the deserialization metadata necessary to read the object. The workload is a list of Branch accesses from a trace of the NPTupleMaker high energy physics application. Each Branch access is:

`Branch0/Branch1,3,1740718587,5847,97,136`

where the tuple is the full Branch name, Basket number, offset into the ROOT file, size of the Basket, start entry of the Basket, and end entry of the Basket. For the file approach, we use the offset into the ROOT file and the size of the Basket. In setup 1, the ROOT file is pulled locally and the Branches are read from the file. In setup 2, the offset and size of the

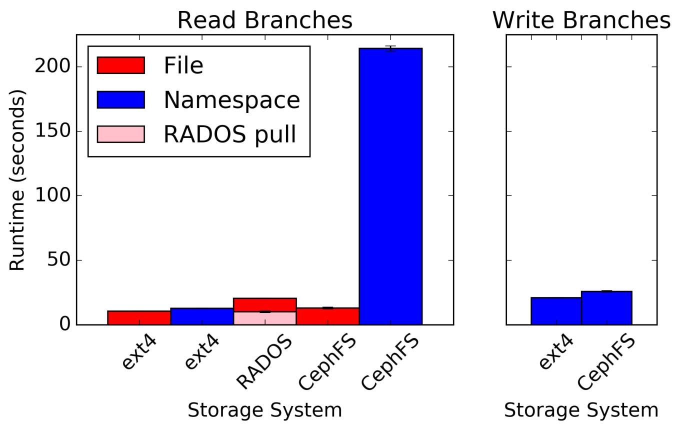


Figure 5: The "File" bars show the runtime of storing HEP data as a single ROOT file and the "Namespace" bars show the runtime of storing a file per Basket. The baseline is the runtime of "ext4" on a single node, "RADOS" is reading HEP data as a Ceph object, and "CephFS" is reading HEP data as files in a CephFS mount. "CephFS" is far slower because of the file system metadata load.

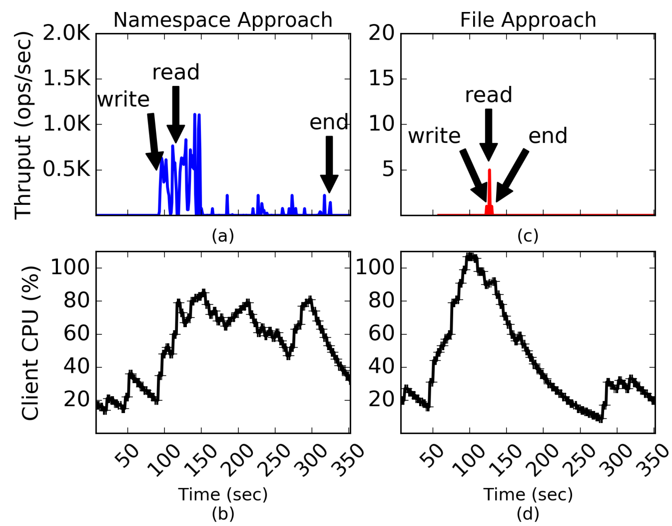


Figure 6: Reading and writing high-energy physics (HEP) data as many files allows physicists to read just the data they care about. But using this namespace approach sends many RPCs to the metadata server (a), resulting in worse performance and lower CPU utilization at the client (b). Alternatively, using the traditional file approach has IO amplification because all data moves over the network but less RPCs (c), better performance, and higher client CPU utilization (d).

read are sent to the CephFS metadata server. For setup 3, the full Branch name and Basket number are used to traverse the file system namespace.

The read and write performance for the different approaches are shown in Figure 5, where the x-axis is different storage backends, the y-axis is runtime, and the error bars are the standard deviations for three runs. We compare against ext4 on a single node as a baseline. The reason that the namespace approach on CephFS is slow is shown in Figure 6. The file system metadata accesses, characterized by many `open()` requests, incur many RPCs. This causes worse performance and lower client CPU utilization compared to reading a single ROOT file. So the cost of read amplification in the file approach is offset by the cost of doing namespace operations. For this experiment, the ROOT file is 1.7GB and 65% of the file is accessed so the namespace approach might be more scalable for different workloads.

Takeaway. The scalability of the ROOT framework is limited by the overhead of contacting the remote metadata server in the underlying distributed file system. At large scale, the metadata server will need to have a network with high bandwidth and capacity.

2.3 Large Scale Simulations: SIRIUS

SIRIUS [7] is the Exascale storage system being designed for the Storage System and I/O (SSIO) initiative. The core tenant of the SIRIUS project is application hints that allow the storage to reconfigure itself for higher performance. Techniques include tiering, management policies, data layout, quality of service, and load balancing.

2.3.1 System Architecture. SIRIUS includes a metadata service called Empress [8], which is a query-able SQLite instance that stores metadata for bounding boxes (*i.e.* a 3-dimensional coordinate space). Figure 7 shows how metadata is stored in and retrieved from Empress; the entire simulation space, represented as a 3D torus, is partitioned into bounding boxes, whose coordinates are stored in SQLite and passed into a objecter function, $F(x)$. The objecter function translates the bounding box coordinates into a list of object IDs, which are stored in the database and used to write/read data to/from the scalable object store.

SIRIUS aims to satisfy queries structured like the "Six Patterns in HPC" [9]. So clients reading from SIRIUS will first contact Empress with the queries for all data, all of 1 variable, all of a few variables, a plane in each dimension, an arbitrary rectangular subset, or an arbitrary area on an orthogonal plane, and Empress will return a list of objects. Armed with this list, the client contacts the object storage system (in our case this is RADOS, Ceph's object store) and reads relevant data.

2.3.2 Namespace Description. The seven columns in the database are:

- six integers representing coordinates and offsets
- 1 string representing object IDs

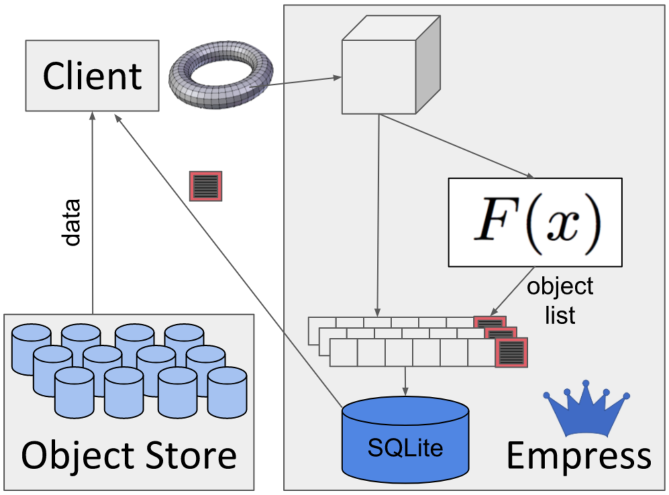


Figure 7: The SIRIUS project uses Empress to store metadata for bounding boxes in a 3D torus. Bounding box coordinates and a list of object names are stored in SQLite. The object names are calculated using an objecter function, labeled $F(x)$ above. For reads, clients query Empress for the object list before reading from the object store.

Each row in the database is duplicated for each variable in the simulation. Variables are domain-specific values that scientists care about, such as temperature and pressure. Figure 7 only has three variables, represented by the rows of coordinates/object list values, but most simulations will have a minimum of 10 variables.

2.3.3 Overheads. Empress lists all objects names for a bounding box and clients filter the list for data of interest. This is a problem because of the size of the object name list. Listing large numbers of items in file systems is notoriously slow and studies on `ls` have shown the operation to be especially heavy-weight [2, 3]. As currently designed, Empress would store too many object names and the queries only use a subset of the resulting list.

# of object names: a back-of-the-envelope calculation for the number of object names in the system is:

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

where reasonable values would be one million processes, writing 8GB of data for 100 timesteps for 10 variables. Using an 8MB stripe size (the optimal object size in RADOS), the object name list size is:

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

Queries use a subset of object names. When running queries characterized by the "Six Patterns in HPC", Empress must exhaustively list objects and filter the coordinates of interest. In addition to the time it would take to search 1 trillion objects in SQLite, another problem is the actual storage footprint of storing this many items.

Takeaway. The scalability of the SIRIUS storage system is limited by the performance of scanning lists. At large scale, the metadata server will need to service many requests to a large namespace.

3 NAMESPACE SCHEMAS

For three domain-specific applications and use-cases, we have identified different scalability bottlenecks:

(1) namespaces with many requests
(2) namespaces managed by remote servers
(3) namespaces that are too large

We propose namespace schemas to support lazy generation of metadata at different nodes in the storage system. A centralized, globally consistent metadata service (either a single metadata server or a cluster of active-active metadata servers) provides clients with (1) the root of the subtree of interest and (2) a namespace schema. Using the namespace schema, clients and servers do not need to materialize the namespace up front, thus avoiding the costs of making the tree with metadata RPCs.

The namespace schema is stored in the directory inode of the root of the subtree of that the client cares about. We use the "file type" interface from the Malacology [10] project to facilitate this domain-specific functionality. This is similar to push-down predicates in databases, where the application is providing domain-specific knowledge that the storage system knows how to leverage. We have defined three types of namespace schemas: a formula, a binary, and a pointer.

3.1 Formula Schema

3.2 Binary Schema

3.3 Pointer Schema

A Fresh, Unorthodox, Unexpected, Controversial, and Counterintuitive Idea

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] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: A Checkpoint Filesystem for Parallel Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*.

[2] Philip Carns, Sam Lang, Robert Ross, Murali Vilayannur, Julian Kunkel, and Thomas Ludwig. Small-file Access in Parallel File

For n processes on m servers:

`# of dirs` = $m \times$ `mkdir()`

`# of file` = $2 \times n \times m$

`# of file per dir` = n/m

(a) Function for PLFS subtrees.

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

(b) Binary for HEP subtrees.

(c) PLFS

Figure 8: Namespaces generated by 3 motivating examples.

Systems. In *Proceedings of the Symposium on Parallel and Distributed Processing (IPDPS '09)*.

[3] Marc Eshel, Roger Haskin, Dean Hildebrand, Manoj Naik, Frank Schmuck, and Renu Tewari. Panache: A Parallel File System Cache for Global File Access. In *Proceedings of the Conference on File and Storage Technologies (FAST '10)*.

[4] Shel Finkelstein. FIXME!. Personal Communication. (FIXME!).

[5] Gary Grider. FIXME!. 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. (FIXME!).

[6] Jun He. FIXME!. IO Acceleration with Pattern Detection. (FIXME!).

[7] S A Klasky, H Abbasi, M Ainsworth, J Choi, M Curry, T Kurc, Q Liu, J Lofstead, C Maltzahn, M Parashar, N Podhorszki, E Suchyta, F Wang, M Wolf, C S Chang, M Churchill, and S Ethier. Exascale Storage Systems the SIRIUS Way. *Journal of Physics: Conference Series* (????).

[8] Margaret Lawson, Craig Ulmer, Shyamali Mukherjee, Gary Templet, Jay F. Lofstead, Scott Levy, Patrick M. Widener, and Todd Kordenbrock. 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*.

[9] Jay Lofstead, Milo Polte, Garth Gibson, Scott Klasky, Karsten Schwan, Ron Oldfield, Matthew Wolf, and Qing Liu. Six Degrees of Scientific Data: Reading Patterns for Extreme Scale Science IO. In *In Proceedings of High Performance and Distributed Computing (HPDC '11)*.

[10] Michael A. Sevilla, Noah Watkins, Ivo Jimenez, Peter Alvaro, Shel Finkelstein, Jeff LeFevre, and Carlos Maltzahn. 2017. Malacology: A Programmable Storage System. In *Proceedings of the European Conference on Computer Systems (EuroSys '17)*.

[11] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI '06)*.

[12] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan 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)*.

[13] Qing Zheng, Kai Ren, and Garth Gibson. BatchFS: Scaling the File System Control Plane with Client-funded Metadata Servers. In *Proceedings of the Workshop on Parallel Data Storage (PDSW '14)*.

[14] Qing Zheng, Kai Ren, Garth Gibson, Bradley W. Settlemyer, and Gary Grider. 2015. DeltaFS: Exascale File Systems Scale Better Without Dedicated Servers. In *Proceedings of the Workshop on Parallel Data Storage (PDSW '15)*.