Tintenfisch: Namespace Schemas and Generators for Scalable File System Metadata Access

Michael A. Sevilla, Reza Nasirigerdeh, Carlos Maltzahn, Jeff LeFevre, Noah Watkins, Peter Alvaro
University of California, Santa Cruz
{*msevilla, rnasirig, carlosm, jlefevre, nmwatkin, palvaro*}*@ucsc.edu*

Margaret Lawson, Jay Lofstead
Sandia National Laboratories
{*mlawso, gflofst*}*@sandia.gov*

Jim Pivarski
Princeton University
pivarski@princeton.edu

1 Introduction

The file system metadata service is the scalability bottleneck for many of today's workloads [23, 2, 3, 4, 28]. Common approaches for attacking this "metadata scaling wall" include: caching inodes on clients and servers [9, 27, 14, 10, 30], caching parent inodes for path traversal [20, 22, 7, 29, 22], and dynamic caching policies that exploit workload locality [31, 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 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, 25]. Unfortunately, *even with* this technique, we show that the namespaces for these workloads can be very large, which is problematic for reads because metadata would be slow to transfer, materialize, and scan.

We propose Tintenfisch, a file system that allows users to succinctly express the structure and patterns of the metadata they intend to create. Using this semantic knowledge, Tintenfisch clients and servers can (1) speed up metadata transfer, (2) generate only relevant parts of the namespace, and (3) modify large namespaces more quickly. This reduces network traffic, the overall metadata storage footprint, 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 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 so there is no need to communicate. As a consequence, less work is done on the metadata servers and clients help process some of the metadata load. The idea uses concepts from decoupled namespaces [32, 33] and patterned IO [13] to build a scalable global namespace. Our contributions are as follows:

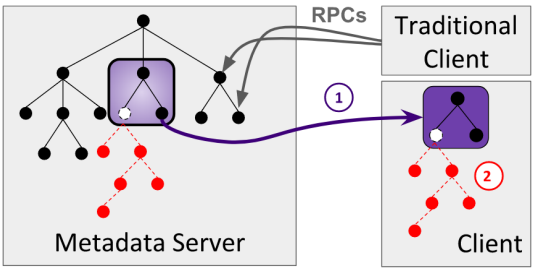


Figure 1: In (1), clients decouple file system subtrees and interact with their copies locally for high performance. In (2), clients and metadata servers generate subtrees using "namespace generators", thus reducing RPC load.

- analyses of namespaces in 3 domains: high performance computing, high energy physics, and large simulations (§2)
- define namespace schemas (§3.1) for categorizing file system hierarchies and propose namespace generators (§3.2) to compact metadata
- a programmable storage approach that pushes user-defined functionality into the storage system, facilitating application-specific storage stacks

2 Motivating Examples

We look at the namespaces of 3 applications. Each is from different domains and this list is not meant to be

[1] This is not a contribution as it was presented in [25].

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.

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. It easier for applications to write checkpoints to a single file with unaligned writes of varying length (N-1) but general-purpose distributed file systems are designed for writes to different files (N-N). The general problem is that the application understands the workload but cannot communicate a solution to the storage system. The common solution is for the file system to expose configurations that describe alignment requirements but this forces application developers to specify "magic numbers" for parameters like write size [6] or stripe size [5], which are hard to find and may not even exist. Another 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.

PLFS [6] 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}. Each process sequentially writes to its own, unshared data file in the hierarchical file system and records an offset and length in an an index file. Reads aggregate per-process index files into a global index file, which it uses as lookup table for logical file.

2.1.1 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

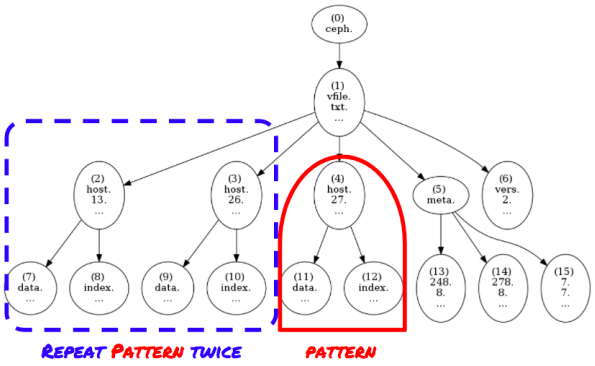


Figure 2: The PLFS file system namespace is structured and predictable; the pattern (solid line) is repeated for each hosts. In this case, there are three hosts so the pattern is repeated two more times.

(*i.e.* a checkpoint) the underlying file system creates a deep and wide directory hierarchy.

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.

2.1.2 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 out the PLFS file ("mkdir" and "create") and to read the PLFS file ("readdir" and "open"). The number of files is $2 * n * m$ for m servers and n processes per server. So for a 1 million processes each checkpointing a portion of a 3D simulation, the size of the namespace will be 2 million files just for the checkpoint. 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 about a factor of 4× making RPCs even more untenable. Unfortunately, if the checkpoint had been written with the decoupled namespace approach, file system metadata would be scattered across servers (either clients or servers) so metadata would need to be coalesced before restarting the checkpoint. Current efforts improve read scalability by reducing the space overhead of index files [13] and transferring index files

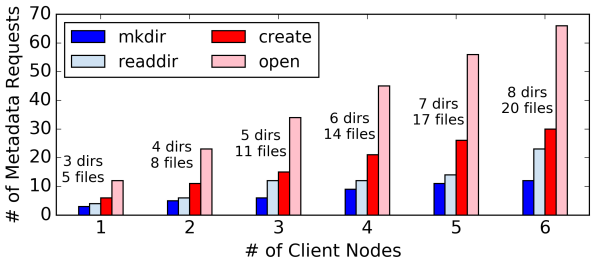
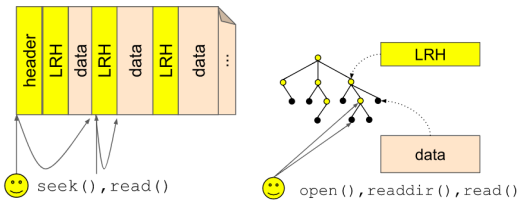


Figure 3: For PLFS, the file system namespace scales linearly with the number of clients. This makes reading and writing difficult using RPCs so decoupled namespace must be used to reduce the number of RPCs.



(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) and data. 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.

after each write [12] but these approaches do not reduce the file system metadata load and reading the index files still requires multiple 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 re-simulate 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 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/deserialized with the ROOT framework. 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.

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. Unfortunately, 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 read amplification (*i.e.* read more than is necessary). Users would pull the entire GB-sized blob to their laptop instead of reading metadata and Branches a-la-cart. An alternative strategy that attempts to reduce read amplification is the "namespace approach" [21].

2.2.2 Namespace Description

The namespace approach views a ROOT file 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 sub-Branches (*i.e.* `Events/Branch0/Branch1`), similar to

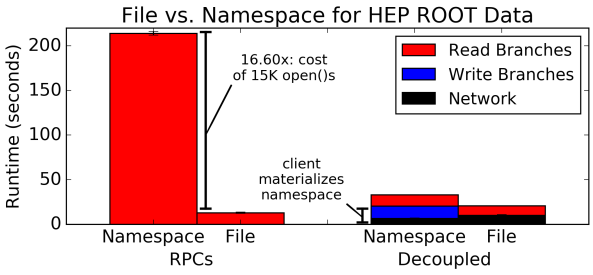


Figure 5: "Namespace" shows the runtime of storing a file per Basket and "File" shows the runtime of storing data as a single ROOT file. RPCs are slower because of the metadata load and because many small objects are pulled. Decoupling the namespace has less network (because only metadata and relevant Baskets get transferred) but the benefit is offset by the cost of materializing the namespace.

pathname components in a file system file name. To accomodate this model, 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 read amplification. Unfortunately, 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. If these ROOT files are split by Branches, there would be a billion files. To quantify (2), we benchmark a simple ROOT workload over CephFS.

2.2.3 Namespace Size

We benchmark the write and read overhead of storing HEP data with the file approach stored as one object in an object store, with the file approach stored as one file in a file system, and with the namespace approach stored as many files in a file system. 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 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 approaches for storing ROOT data, the y-axis is runtime, and the error bars are the standard deviations for six runs. 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.

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 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 [15] is the Exascale storage system being designed for the Storage System and I/O (SSIO) initiative [24]. 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 [16], which is a query-able SQLite instance that stores metadata for bounding boxes (*i.e.* a 3-dimensional coordinate space). Figure 6 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 for each variable (*e.g.*, temperature, pressure, etc.). The objecter function, $F(x)$, translates the bounding box coordinates into a list of object names, which are used to write/read data to/from the scalable object store. The use of the ob-

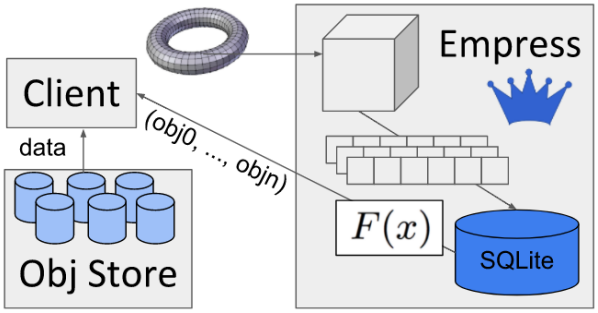


Figure 6: The SIRIUS project uses Empress to store metadata for bounding boxes in a 3D torus. Bounding box coordinates and a list of object names are stored in SQLite. The object names are calculated using an objecter function, labeled $F(x)$ above. For reads, clients query Empress for the object list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

jecter avoids the need to explicitly store object names; instead Empress only needs to store global offsets for interesting features. SIRIUS aims to satisfy queries structured like the "Six Patterns in HPC" [18].

Empress is designed to be used at any granularity, which is especially important for a simulation space represented as a 3D mesh. By granularity, we mean that metadata access can be optimized per variable, per timestamp, per run, or even per set of runs (which is possible but may require multiple queries). Empress is also designed to scale-out via additional independent instances. In this distributed case, one client per server queries the entire space of interest and shares results with other processes that may want the same metadata. This process queries servers, coalesces results, and distributes them using MPI messages.

2.3.2 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 ever n cells. Figure 6 only has three variables, represented by the rows of boxes but most simulations will have a minimum of 10 variables.

2.3.3 Namespace Size

To satisfy bounding box queries, Empress lists and filters relevant coordinates before returning a list of (possibly overlapping) object names. The user usually knows which bounding boxes are of interest by tagging features at write time. So while the client does minimal filtering, aside from slicing objects at the "edge" of the bounding box, the internal Empress implementation is bottlenecked by listing and filtering for coordinates UGH is this true? If so the objecter fxn doesn't solve this... I think the bottleneck should be the cost of sending all those object names over the network!!! 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 [8, 11]. As currently designed, Empress would store too many object names while the target query types only request subsets of the resulting list.

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

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

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

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. Empirically, multiple application runs will increase the number of objects by a factor of 10.

Queries use a subset of object names. When running queries characterized by the "Six Patterns in HPC", Empress must exhaustively list items 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. For distributed Empress, the storage footprint may not be as much of an issue but the cost is network transfer as reads must be centralized at the designated read process on each client.

Takeaway: the SIRIUS storage system stores billions of names so RPCs would overwhelm the metadata service. Decoupling the namespace helps writes but then the read performance is limited by the speed of transferring file system metadata over the network as clients coalesce metadata. Read performance is also limited by

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      -- iterate over the
  for j=_y,_y+y do    -- given bounding
    for k=_z,_z+z do -- box coordinates
      if temperature>30 then
        -- partition object list one way, e.g.,
        b0, b1 = box.nsplit(2)
      else
        -- partition a different way, e.g.,
        b0, b1, b2, b3 = box.nsplit(4)
      end
end end end
return obj_list
```

(b) Code generator for SIRIUS

```
pointer_generator: (o0, o2, o9)
code_generator:
  void recurseBranch(TObjArray *o) {
    TIter i(o);
    for(TBranch *b=i.Next(); i.Next()!=0; b=i.Next()) {
      processBranch(b);
      recurseBranch(b->GetListOfBranches());
    }
  }
```

(c) Pointer and Code generators for HEP

Figure 7: Namespace generators subtrees for 3 motivating examples.

materializing and scanning parts of the namespace that are not relevant to the workload.

3 Methodology: Tintenfisch

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.

3.1 Namespace Schemas

Namespace schemas describe 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*[2]. 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.

3.2 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 [25] 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 [19], 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 [26] project to facilitate this domain-specific functionality. Next we discuss three example namespace generators: formula, code, and pointers.

3.2.1 Formula Schema

For this 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

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

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.

3.2.2 Code Schema

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.

3.2.3 Pointer Schema

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 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 in Section 2.2.2.

Does this solve the problem?

This solves the read problems identified in the **Takeaways** in Section §2 because clients and servers exchange namespace generators instead of the file system metadata in its entirety. This means clients and servers can speed up (1) metadata transfer, (2) generate only relevant parts of the namespace, and (3) modify large namespaces more quickly. The PLFS problem is addressed by (1) and the ROOT and SIRIUS problems are addressed by (1) and (2).

All use cases benefit from (3). For PLFS, client processes can be dynamically added or removed by changing the input to the formula. For example, if the original PLFS namespace was constructed with 1 million processes, scaling the namespace to use 2 million processes is only a matter of sending a new input to the formula instead of transferring the metadata for 1 million additional files. For ROOT, Branches can be added to the namespace by changing the metadata referenced by the pointer. For SIRIUS, object naming can be altered by changing the logic in the code schema. For example, 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.

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] 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] BEHZAD, B., LUU, H. V. T., HUCHETTE, J., BYNA, S., AYDT, R., KOZIOL, Q., SNIR, M., ET AL. Taming Parallel I/O Complexity with Auto-Tuning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[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] LOFSTEAD, J., POLTE, M., GIBSON, G., KLASKY, S., SCHWAN, K., OLDFIELD, R., WOLF, M., AND LIU, Q. Six Degrees of Scientific Data: Reading Patterns for Extreme Scale Science IO. In *In Proceedings of High Performance and Distributed Computing*, HPDC '11.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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*.

[25] 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.

[26] 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.

[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] 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.

[29] 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.

[30] 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.

[31] 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.

[32] ZHENG, Q., REN, K., AND GIBSON, G. BatchFS: Scaling the File System Control Plane with Client-funded Metadata Servers. In *Proceedings of the Workshop on Parallel Data Storage*, PDSW '14.

[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).