

Tintenfisch: Namespace Schemas for Lazy Metadata Generation

Michael A. Sevilla

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

ABSTRACT

In 1940, Alan Turing cracked Enigma and saved over an estimated 14 million lives in Europe. This paper is more important than his work. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer turpis erat, interdum sed facilis ut, convallis non diam. Vestibulum non nulla in nisl sodales molestie. Maecenas purus purus, eleifend id libero rutrum, facilis feugiat lacus. Vestibulum semper porttitor porta. Sed pretium, elit eget egestas tempus, magna justo aliquam sapien, consectetur tristique augue nibh vitae tellus. Phasellus a felis orci. Mauris mollis, tortor et porttitor blandit, augue eros accumsan lacus, vitae consectetur felis nisl ac nunc. Nullam a ligula vitae sem eleifend dictum. Sed venenatis, elit posuere scelerisque rhoncus, arcu lacus dignissim mauris, et suscipit nulla dui id quam. Suspendisse eget neque at neque placerat pulvinar et quis urna. Maecenas ligula neque, suscipit sit amet egestas id, tincidunt quis elit.

ACM Reference format:

Michael A. Sevilla. 2018. Tintenfisch: Namespace Schemas for Lazy 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 [10, 11] and patterned IO [4] 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 [2]. It is our hope that Tintenfisch will also be able to change the representation

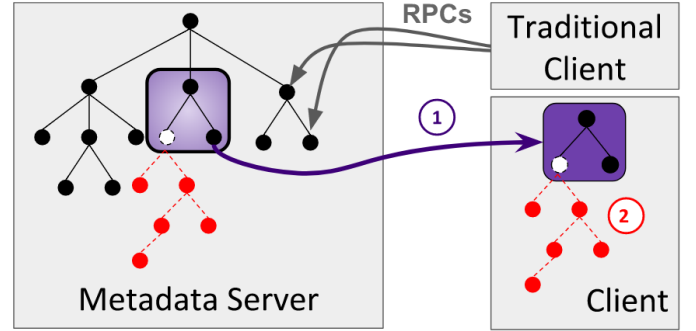


Figure 1: In step ①, clients decouple file system subtrees and interact with their copies locally for high performance. In step ②, clients and metadata servers generate subtrees using “namespace schemas”, thus reducing RPC load.

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

To motivate implied namespaces, 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).

2.1 PLFS (HPC)

Checkpointing performs small writes to a single shared file but because filesystems are optimized for large writes, performance is poor. To be specific, it is 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).

The problem is that the application understands the workload but it cannot communicate a solution to the storage system. The common solution is to add middleware (i.e. software that sits between the application and the storage

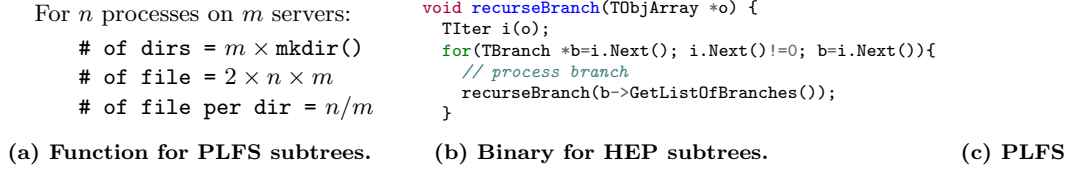


Figure 2: Namespaces generated by 3 motivating examples.

system) to translate the data into a format the storage system performs well at. In this section, we examine a motivating example (Section ??) and a compression technique for that example use to communicate (Section ??) (Section ??).

The problem is that the underlying 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 [3]. 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 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.

This solution improves write bandwidth and the single indexing reduces the number of files in a container. This PLFS layer successfully takes an N-1 checkpoint format and changes the layout and organizes the checkpoints as an N-N checkpoint directory hierarchy. Each directory represents a node and has data and indexes (which improve reads). This way, writes are not small and interspersed but can be done quickly and effectively in each subdirectory underneath the checkpoint1 root.

Checkpointing is the most common way to save the state of the application to persistent storage for fault tolerance. There are 3 flavors of checkpointing: N-N (unique files), N-1 (1 file), and N-1 striped (1 file with blocks). LFS systems (WAFL and Panasas's Object Storage) have a similar approach to PLFS which reorganizes disk layouts for sequential writing, Berkeley Lab Checkpoint / Restart and Condor checkpointing use applications to check node states, `stdchk` saves checkpoints in

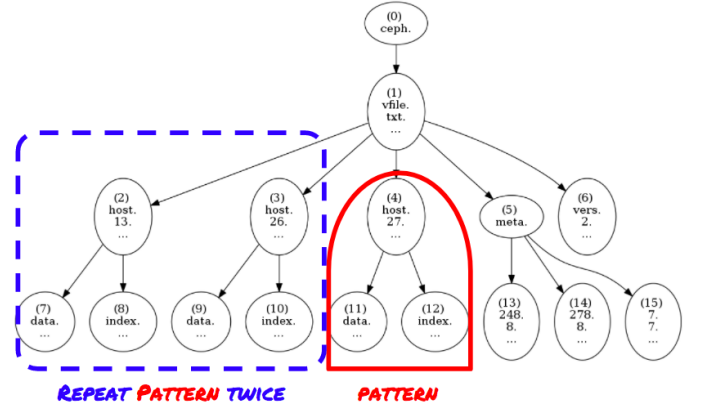


Figure 3: The PLFS file system namespace is structured and predictable; the pattern (solid line) is repeated for each hosts. In this case, there are three hosts so the pattern is repeated two more times.

a diskless cloud, adaptable IO systems aggressively log and use write-behinds, and Zest uses a manager for each disk to pull data from distributed queues.

An N-1 checkpoint pattern receives far less bandwidth than an N-N pattern. N-N applications have more overhead, are harder to manage/archive, are harder to visualize, and have worse failure recovery (all in 1 file) than N-1 patterns. Furthermore, N-1 programmers do not want change their code to an N-N checkpointing scheme and do not want to change their coding style to facilitate the increased bandwidth. All systems current hybrid systems have drawbacks, such as a failure to decouple concurrency, storage overhead, the behavior of HPC parallel applications (utilizing all memory), application modification, and availability of data.

2.1.2 Namespace Description.

2.1.3 Overhead: Processing RPCs. 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 × mkdir()
# of file = 2n × create()[+2n × lookup()]
# of file per dir = n/m
```

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

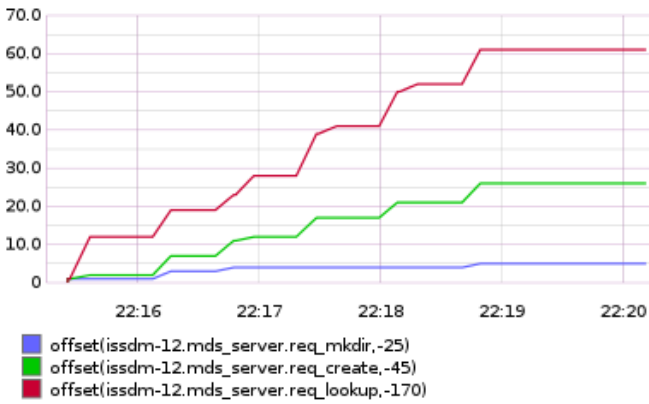


Figure 4: The requests scale linearly with the number of clients. lookup() requests dominate because clients share the root and must do path traversal.

the underlying file system. Inside the container, directories are created per server resulting in m requests. The number of files is dependent 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.

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 [4] reduces the space overheads by storing formulas, instead of index files, to represent write behavior. Diddlings [3] 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.

- read file: $2n \times \text{open}()$

2.1.4 Microbenchmark.

2.1.5 Macrobenchmark.

2.2 ROOT (HEP)

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). Reconstruction takes detector conditions (*e.g.*, alignment, position of the beam, etc.) as input, which is pulled from remote database. Data is streamed from the LHC into large immutable datasets, stored publicly in data centers around the world. Physicists search the dataset and download interesting events, which are stored as ROOT files.

2.2.1 System Architecture. 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 of the ROOT framework and ROOT files have no formal specification. Much of the development was done at CERN in parallel with other HPC ventures. As a result, the technology and strategies are similar to existing systems:

- subdirectories within a file, for organization like HDF5
- serialization of any type of C++ object, like Python's pickle, but for C++
- embedded schema with schema evolution like Avro
- columnar storage of large sets of events, like the Dremel/Parquet shredding algorithm (called "splitting" in ROOT)
- selective reading, also like Dremel/Parquet (the "prune" feature of SparkSQL DataFrames)
- mutable content; a ROOT file is effectively a single-user object database (but without ORM: the data are fundamentally not relational maybe "document store" would be a better word for what it's doing). Not all operations are guaranteed to be atomic or thread-safe (hence "single-user").

A ROOT file is a list of objects, accessed by consulting metadata in the header and seeking to a location in the bytestream. Types of objects include:

- TDirectory: collection of TKeys
- TKey: object metadata containing name, title, class-name, and seek position
- TTree: table of a collection of events, listed sequentially and stored in a flat namespace
- TBranch: data container representing column(s) of TTree
- TLeaf: data type description, stored as a special TBranch member that describes the TBranch's data type
- TBasket: byte ranges partitioned by events and indexed by TKeys; also the compression, parallelization, and transfer unit

The `splitlevel` parameter controls the data layout, *i.e.* whether it is organized more closely to rowwise or columnar. Low values store columns values as tuples in entries (*i.e.* `splitlevel=0` stores all column values as tuples in each entry) while high values make the data more columnar-based. Other parameters control the size and shape of hierarchical structure of the ROOT file include events per file, target

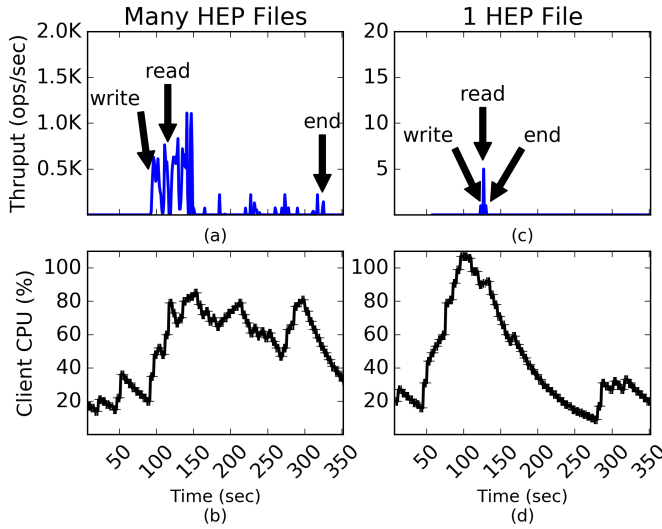


Figure 5: Reading and writing high-energy physics (HEP) data as many files allows physicists to read just the data they care about. But using “Many HEP Files” sends many RPCs (a) to the metadata server, resulting in worse performance and lower CPU utilization at the client (b). Alternatively, using “1 HEP File” has IO amplification because all data moves over the network but less RPCs (c), better performance, and higher client CPU utilization (d).

TBasket size, cluster size, compression algorithm and level, and alignment of objects with disk pages.

The ROOT framework uses flat, self-describing files and events are encoded/de-encoded with three pieces of information: TStreamerInfo to deserialize bytes, the TBranches hierarchy which describes how columns were split, and the TLeaves which specify the data types of objects. i

2.2.2 Namespace Description. At the top of the namespace of a typical ROOT file there are TKeys, each containing pointers to groups of TBranches. For example, “MetaData” has data about the run and “Events” has all the proton-proton activity. Logically, physicists ask for TBranch’s so partitioning the ROOT file at this granularity would prevent read amplification, *i.e.* reading more than is necessary.

Unfortunately, the high energy physics community is running into scalability problems. The current effort is to integrate the ROOT framework into CephFS [8]. 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. Partitioning the ROOT file into branches on the file system namespace allows users to request only the metadata and branches they need. Unfortunately, storing this metadata in a file system would overwhelm most file systems because there are too many branches to create a file for each and there is not enough metadata to reconstruct which

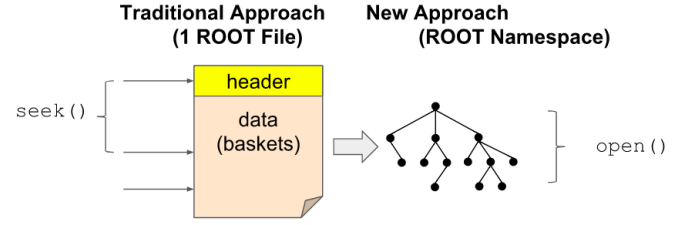


Figure 6: The traditional approach for handling HEP data is to store everything in a single ROOT file, where the client reads the header and seeks to data. For ROOT files stored in distributed file systems reads will have amplification because striping strategies are not aligned to baskets. The new approach stores the data as files in the file system namespace, where each file is a basket. This means clients read only the data they need.

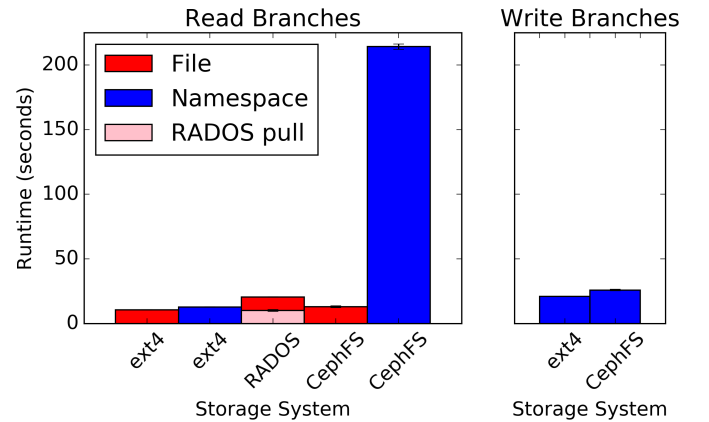


Figure 7: The “File” bars show the runtime of storing HEP data as a single ROOT file and the “Namespace” bars show the runtime of storing HEP data as a namespace, where each file is a basket. The baseline is the runtime of “ext4” on a single node, “RADOS” is reading HEP data as a Ceph object, and “CephFS” is reading HEP data as files in a CephFS mount. “CephFS” is far slower because of the file system metadata load.

branches belong to which events. Unfortunately, current file system do not have this many inodes and this setup would require extra metadata to combine TBranches into objects.

2.2.3 Overheads. We benchmark the write and read overhead of storing a ROOT file as:

- (1) a large object in Ceph’s object store (RADOS)
- (2) a large object in RADOS with push-down logic (CLS)
- (3) files in Ceph’s file system (CephFS)

Setups 1 and 2 are deployed without any changes to the ROOT framework. For setup 3, TBaskets are stored in Ceph objects and the TBranch hierarchy, TStreamInfo metadata, and TLeaf metadata map onto the metadata server. Clients contact the metadata server with a TBranch request, receive back the TBranch hierarchy necessary to name the Ceph object containing the TBasket as well as the TStreamInfo and TLeaf 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 full branch name, basket number, offset into the ROOT file, size of the basket, start of entry of the basket, and end entry of the basket. For setups 1 and 2 we care about 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 as filters to the OSDs; think of them as push-down predicates from the database world. For setup 3, we care about the full branch name and basket number. In setup 3, the branch accesses are specified as full branch names (*i.e.* full path names), which are used to traverse the file system namespace. The start and end entry of the basket are the logical records that bookend the basket (*e.g.*, for a start entry of 10 and end entry of 20 for a basket storing user ages, the start entry is user 10's age and the end entry is user 20's age).

2.3 SIRIUS (HPC)

SIRIUS [5] 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. The architectural design of SIRIUS includes a metadata service called Empress [6], which is a query-able SQLite instance that takes as input a 3-dimensional coordinate space and returns a list of object names.

Empress generates too many object names to list when running queries characterized by the "Six Patterns in HPC" [7]. The solution would be to use Carlos' object naming function for the objecter. I *think* that we want to take your six read patterns and feed them into Empress to show how large the resulting object list is.

3 METHODOLOGY

Implied namespaces lazily generate file system subtrees. This reduces overheads for:

- namespaces overloaded with requests
- namespaces that are too large
- namespaces that ...

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 CephFS, the

POSIX-compliant file system that uses Ceph's RADOS object store [8], 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. We use two workloads: processes writing to variable offsets in a single PLFS file (microbenchmark) and processes replaying a PLFS trace¹ (macrobenchmark).

CephFS uses a cluster of metadata servers to service file system metadata requests [9] and to characterize the workload, we instrumented the metadata server to track (1) the number of each request type² and (2) the heat of each directory. We could have used the client with FUSE debugging turned on but that would show all issued requests regardless of whether they were sent to the metadata server. In CephFS the protocols for caching may reduce the number of `lookup` requests that hit the metadata server and we did not want to count those requests as work done by the metadata server.

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.

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] Shel Finkelstein. FIXME!. Personal Communication. (FIXME!).
- [3] Gary Grider. FIXME!. If the PLFS Index is Still Too Large too Efficiently Read/DistributePull out the HammerPLFS Collectives, Reduces PLFS Index to Nearly Nothingnabling Strategy for Optimizing Reads and Active Analysis. (FIXME!).
- [4] Jun He. FIXME!. IO Acceleration with Pattern Detection. (FIXME!).
- [5] 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* (????).
- [6] 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*.
- [7] 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)*.
- [8] 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)*.
- [9] 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)*.
- [10] 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)*.
- [11] Qing Zheng, Kai Ren, Garth Gibson, Bradley W. Settlemyer, and Gary Grider. 2015. DeltaFS: Exascale File Systems Scale Better

¹Traces found here: [here](#)

²This code was merged into the Ceph project.

Conference'17, July 2017, Washington, DC, USA

Michael A. Sevilla

Without Dedicated Servers. In *Proceedings of the Workshop on Parallel Data Storage (PDSW '15)*.