Tintenfisch: Compacted and Mergeable Namespaces

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

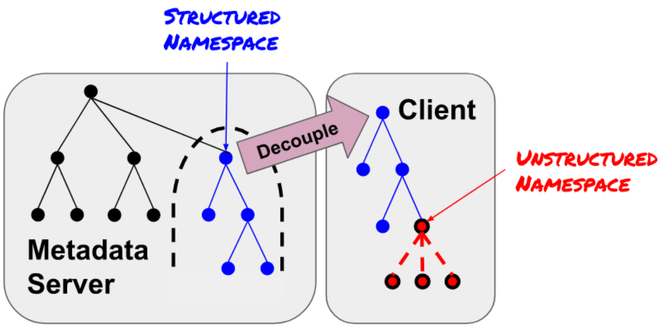


Figure 1: Clients decouple the file system subtrees and interact with their private copiese locally for high performance. They can specify the structure of the metadata they intend to create (structured namespace) or they can create ad-hoc metadata (unstructured namespace), which is merged later.

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.

ACM Reference format:
Michael A. Sevilla. 2017. Tintenfisch: Compacted and Mergeable Namespaces. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 5 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

We propose PaulOctoFS, 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, PaulOctoFS 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 [8, 9] 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 the PaulOctoFS 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

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).

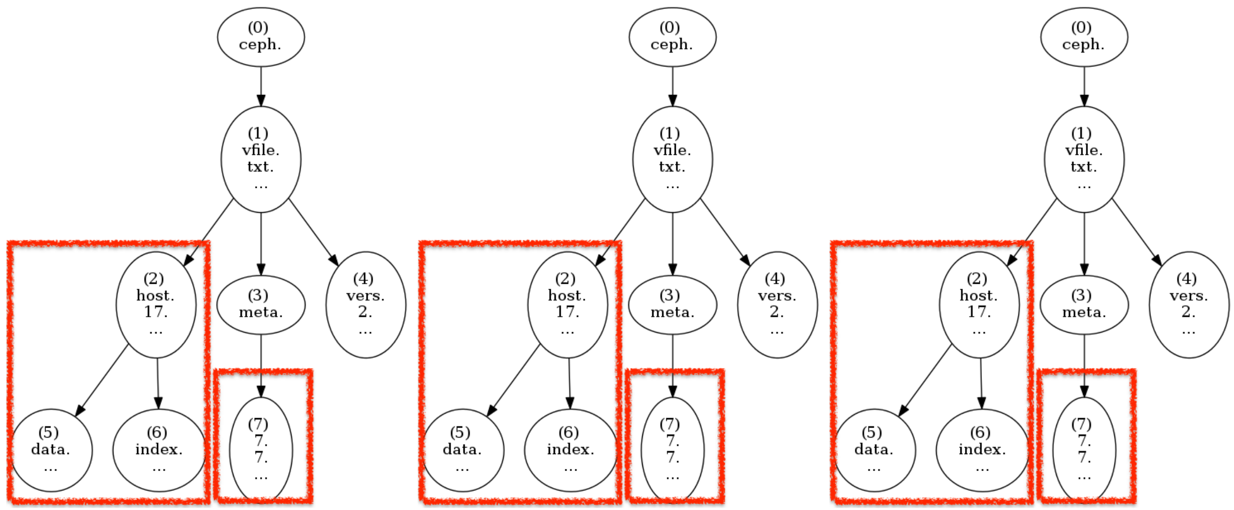
2.1 PLFS (HPC)

Checkpointing performs small writes to a single shared file but because filesystems are optimized for large writes, performance is poor. To be specific, it 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 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. In this section, we examine a motivating example (Section **??**) and a compression technique for that example use to communicate (Section 2.1.1) (Section 4.1).

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 [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 incurrs a `readdir` and, if the file is open by another process, an additional `stat()` because metadata cannot be cached in the container [1].

Conference'17, Washington, DC, USA
2017. 978-x-xxxx-xxxx-x/YY/MM...$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn



(a) PLFS **(b) PLFS** **(c) PLFS**

Figure 2: Namespaces generated by 3 motivating examples.

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 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 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.

2.1.1 Language: Pattern PLFS. I/O access patterns are studied extensively and results are integrated into existing systems. The common checkpointing technique, employed by ADIOS and PLFS, transform the concurrently written file into exclusively written file fragments.

Despite extensive studies on I/O access patterns, current systems do not dynamically recognize patterns at a fine granularity. Because the PLFS checkpoint technique makes many small writes, it is either slow (on disk) or consumes a large amount of space (memory).

The authors present algorithms to discover and replace PLFS metadata. The system is composed of:

- local per-process metadata: split based on pattern discovering engine (get tuples using sliding window)
- merge local indices into a single global one per PLFS (check if local neighbors abut each other)

The authors' algorithms rediscover information as data moves through POSIX. By dynamically pattern matching and compression, they are able to reduce latency and disk/memory usage on reads.

They tested with FS-TEST, MapReplayer, and real applications. In their experiments, metadata is reduced by several orders of magnitude, write performance is increased (by 40%), and reads are increased (by 480%).

Discovering structure in unstructured IO is useful for other systems, like pre-fetching and pre-allocation of blocks in file system or SciHadoop (metadata/data). This work that these algorithms (applied to compress metadata) can successfully optimize I/O.

- What PLFS structures allow us to to this?
- How dependent on workloads are these?
- Can this be extended to other file systems?

2.2 ROOT (HEP)

2.3 SIRIUS (HPC)

3 COST OF TRANSFORMATIVE WRITE IO

In this section, we show the performance implications of transformative IO by examining the metadata request load incurred by PLFS. We use CephFS, the POSIX-compliant file system that uses Ceph's RADOS object store [6], 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 trace[1] (macrobenchmark).

CephFS uses a cluster of metadata servers to service file system metadata requests [7] and to characterize the workload, we instrumented the metadata server to track (1) the number of each request type [2] 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 down by the metadata server.

3.1 Overhead of Metadata Writes

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:

- create directories: $m \times$ `mkdir()`
- create files: $2n \times$ `create()`$[+2n \times$ `lookup()`$]$
- files/directory: n/m

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 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.

[1] Traces found here: here

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

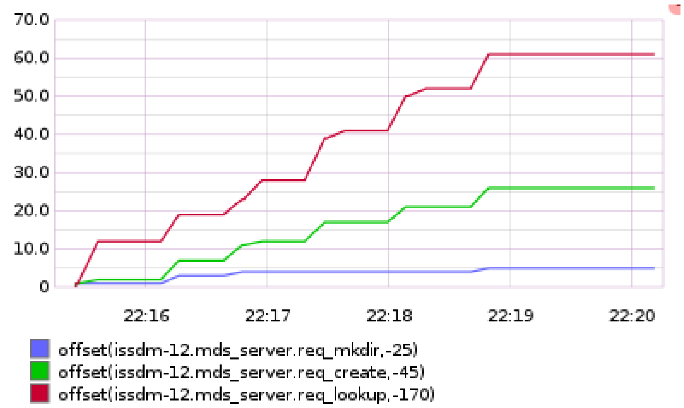


Figure 3: The requests scale linearly with the number of clients. `lookup()` requests dominate because clients share the root and must do path traversal.

3.1.1 Microbenchmark. TODO: multiple processes per node TODO: verify path traversal

3.1.2 Macrobenchmark.

3.2 Overhead of 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: $2n \times$ `open()`

3.2.1 Microbenchmark.

3.2.2 Macrobenchmark.

4 METHODOLOGY

In this section, we show how clients and metadata servers communicate using the Pattern PLFS language and present our storage system that adapts to the wokload (Section 4.1)). Other destructive solutions include changing the storage system and altering the application.

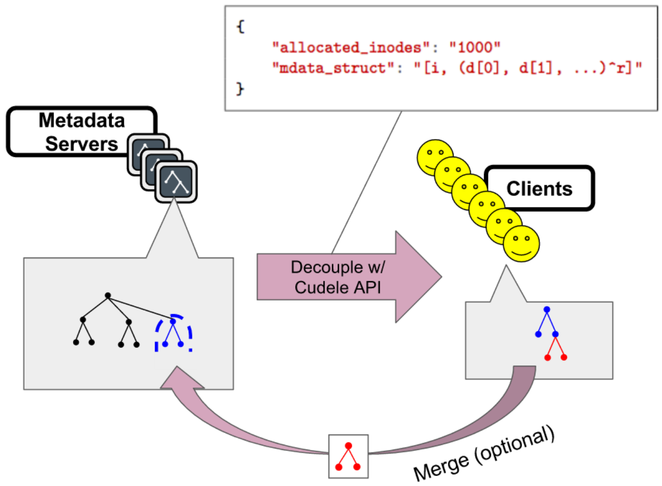


Figure 4: System XX lets clients optimize performance by telling the storage system about the workload. Clients can specify a Structured Namespace (blue subtrees and Section 4.1.1) or by merging file system metadata from an Unstructured Namespace (red subtree and Section 4.1.4).

4.1 Adapting to the Workload with Cudele

Cudele is a file system with programmable consistency and durability. Clients use an API to decouple existing subtrees from the global namespace; metadata operations from the other clients targeted at the decoupled subtree can be programmed to be blocked or marked as overwritable. With the decoupled subtree in hand, the client can do metadata operations locally. Upon completion, the client can merge the subtree back into the global namespace.

Cudele has the mechanisms for understanding the file system metadata language and adapting to the workload. Figure 4 shows how clients decouple the namespace with the Cudele API, specifying how many extra inodes they want and the structure for the namespace they intend to create. The metadata server and client both know about the metadata in the blue subtree, requiring no RPCs, and if the client creates more metadata (red subtree), it can merge it back into the global namespace. This model lets users enjoy the simplicity of global namespaces and the high performance of node-local operations. We extend the API to support the declaration of structured namespaces and leverage the existing API to merge unstructured namespaces.

4.1.1 Structured Namespaces. A structured namespace is created according to a pattern. If both the client and metadata server knows the pattern, they can create the metadata independently. This has two benefits: (1) it reduces RPCs which improves performance and reduces network traffic and (2) it allows the client and server to operate in parallel. The patterns that Cudele understands are shown in Listing 1 and the programmable interfaces are shown below. There are two parameters for unstructured namespaces: `pattern` and `trigger`.

4.1.2 Trigger: Start Namespace Construction. `trigger` specifies when to start the namespace construction on the metadata server. The metadata reconstruction can be asynchronous and saving this resource intense process for later can have better performance. To facilitate the exploration of different trigger policies, we make the value for the `trigger` parameter programmable. Administrators inject Lua code that specifies or calculates thresholds for when to start namespace construction. Although we make this programmable, we do not make any conclusions about the best trigger time and leave the exploration of this space as future work.

In Listing 1, the trigger is:

```
{
  if MDSs[whoami]["cpu"] > 30
}
```

which means that construction of the namespace will start if current MDS (`whoami`) has a CPU utilization (‘‘`cpu`") above 30%.

Triggering construction asynchronously can improve performance because the process can be deferred until the system has less load. However, this performance gain comes at the cost of consistency. Even if the construction is triggered immediately, the metadata is eventually consistent; other clients see outdated metadata because the namespace is sitting on the client. Delaying the trigger improves the liklihood that system finds a window of low load but also increases the latency of other clients.

Implementation: we re-use the polling and embedded Lua virtual machine in Mantle [5] to implement the trigger interface. By default, every 10 seconds the metadata server checks if the condition for triggering is satisfied by executing the Lua code. Mantle has variables exposed for administrators to explore load balancing policies; just like this work, some of these policies need to identify overloaded metadata servers so we re-use all those variables. Some of the more useful variables include:

- Memory Usage
- CPU Utilization
- Request Rate
- Queue Depth
- Server Tags: whoami, i

4.1.3 Pattern: Express Namespace. `pattern` describes the metadata layout of the Structured Namespaces. It is the same language used in [4]. When the metadata server starts a namespace construction, it creates all the file system metadata generated by this formula. As a refresher, the pattern in Listing 1:

$$[i, (d[0], d[1], ...)^r]$$

means that there are r entries in the PLFS index file, where the first entry has a physical offset of i and lengths of d, where the pattern in d repeats.

Implementation: Another big fat TODO.

```
{
  <!-- Structured Namespace Pattern !-->
  "S_pattern": "[i, (d[0], d[1], ...)^r]",

  <!-- Structured Namespace Trigger !-->
  "S_trigger": "if MDSs[whoami]["cpu"] > 30",

  <!-- Untructured Namespace Allocated Inos !-->
  "US_alloci": "1000",
}
```

Listing 1: Using the Cudele API to express metadata structure, which is understood by both the server and client.

4.1.4 Unstructured Namespaces.

4.1.5 Migrating Metadata Construction.

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* (2009) *(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 Nothingdnabling Strategy for Optimizing Reads and Active Analysis. (FIXME!).

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

[5] Michael A. Sevilla, Noah Watkins, Carlos Maltzahn, Ike Nassi, Scott A. Brandt, Sage A. Weil, Greg Farnum, and Sam Fineberg. 2015. Mantle: A Programmable Metadata Load Balancer for the Ceph File System. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*.

[6] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design & Implementation (OSDI'06)*.

[7] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. 2004. Dynamic Metadata Management for Petabyte-Scale File Systems. In *Proceedings of the 17th ACM/IEEE Conference on Supercomputing (SC'04)*.

[8] Qing Zheng, Kai Ren, and Garth Gibson. 2014. BatchFS: Scaling the File System Control Plane with Client-funded Metadata Servers. In *Proceedings of the 9th Workshop on Parallel Data Storage (PDSW' 14)*.

[9] 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 10th Workshop on Parallel Data Storage (PDSW' 15)*.