Evolving Directory Model, Storage Structures, and Interface for Large Scale Small File Access

Can Ma, Jin Xiong, Dan Meng, Weiping Wang, Jizhong Han

`{macan,xj,md,wpwang}@ncic.ac.cn`

National Research Center for Intelligent Computing Systems,
Institute of Computing Technology, CAS, Beijing, China, 100190

Abstract

In emerging popular Web applications, e.g. online photo galleries, documents and social network services, large scale small files are concurrently created and read. These applications exhibit low latency requirements for accessing massive small files. Particularly, *N-Form* and *1-N Mapping* issues of distributed file systems, that affect I/O performance of small files, have not been addressed yet.

In this paper, we propose to use tabular directory model, structured storage and tabular query interface to address *N-Form* and *1-N Mapping* issues to boost up small file handling. We implement a prototype file system to manage billions of small files, and focus on improving metadata scalability elastically and exploiting locality and parallelism to improve small file I/O.

Our evaluation demonstrates that the new directory model, storage structures, and interface can not only save storage space, but also increase small file I/O performance. With 32 servers, we deliver the leading metadata and small file I/O performance: 180,000 file creates per seconds and 1GB/s write bandwidth.

1 Introduction

Emerging Web applications, e.g. online photo, audio, video, document and social network services, exhibit very different data access and storage requirements from HPC applications. Lots of small files are created, processed, and returned every second. For example, Facebook, maintained over 260 billions of photos and provided a peak access throughput of 550K photos per second in 2010[1]. Taobao, a famous e-commercial site in China, maintained over 20 billions of photos with average photo size at 15KB[2]. Renren, a popular social network in China, maintained billions of files with average file size at 32KB[3]. And geographic information services (GIS), e.g. Google Maps and OpenStreetMap, rely on billions of map tiles(a few KB to hundreds KB) to serve. With the evolution of Web 2.0, application requirements of storing and accessing billions of small files are increasing. The typical characteristics of small file access are: **a)** lots of small files, **b)** concurrent file creations and reads, and **c)** little file modifications. We can foresee that Web applications in the next few years require scalable distributed storage systems that can efficiently write, read and process on billions or even trillions of small files, and provide low latency query (or search) interface to them. However, as far as we know, most file systems can not handle small files as efficient as handle large files. Classic distributed file systems are designed to optimize I/O bandwidth for large files rather than to optimize latency for small files. Besides this, Web applications had exposed another two problems which make things even worse.

Problem A: (*N-Form*) Many Web applications need to store *N* forms of the original file. For example, online photo galleries need to store several different resolutions of an image for previews. HayStack, developed by Facebook to store user photos, keeps the original image and three different sized thumbnails[1]. Video services, e.g. Youtube, need to store many key frames of a video to serve user previews. Online document sites, e.g. Google Doc, need to keep summary page and preview pages for each document. For these files that derived from one file, saving them in independent files means that *N* copies of *almost-the-same* metadata are generated. It not only wastes space, but also tenses the metadata service. And data access performance is also penalized by random seeking among these files.

Problem B: (*1-N Mapping*) Most distributed file systems map a file in its namespace to *one* or *N* files in local file systems. For example, in Lustre[29], PVFS2[30] and Ceph[18], file data is stored into one (for small file) or several files (for large file) of storage node's local file system. It means that each file data access requires at least one metadata lookup in local file system. Although this out of core indexing has little impact on large file access, it greatly increases the I/O latency for small files.

In this paper, we evolve the classic directory model, storage structures, and interface to address the small file access issues. We propose a tabular directory model and query interface for *N-Form* problem, and directory based file aggregation and structured store for *1-N Mapping* problem. The tabular directory model group-

s *N* forms of a file together to reduce metadata footprint and mitigate service stress. And, directory based aggregation eliminates out of core indexing to reduce I/O latency. We focus on improving metadata scalability elastically, and exploiting locality and parallelism to accelerate small file I/O. We evaluate these methods in a prototype file system, *PomegranateFS*(abbr. PFS), to reveal the practical effects.

Our main contribution is that we evolve a new tabular directory model, storage structures, and query interface for large scale small file access. It solves *N-Form* and *1-N Mapping* problems for typical Web applications and gets best-in-class small file performance.

In the rest of the paper, we presents the motivations and related works in Section 2, design issues in Section 3, implementation details in Section 4, evaluation results in Section 5, and conclusion in Section 6.

2   Motivation and Background

Over the last two decades, research in distributed file system was mainly driven by application workloads that emphasized on large files. Few researches optimized small file access performance in specific file systems[17, 14, 20, 32, 31]. And most file system cannot handle large scale small files efficiently. This section summaries the technology trends calling for large scale small file access and how current file systems are ill-suited to satisfy this call.

2.1   Motivation

In today's social networks, photo gallery is an important I/O workload. For instance, massive users of QZone [1] upload tens of millions photos every day. For each photo, at least *three* different sized (small, medium, and original) images have to be saved. Typically, file sizes are under 1 MBs, from a few KBs to hundreds KBs. These different sized photos has special purposes. The small one is used as a thumbnail for gallery previews. The medium one is used in list previews. And the original one is used for downloading. Web site keeps gallery metadata and photo lists in relational database but image files in distributed file system (DFS), because saving these images in database is both inconvenient for management and inefficient for access. Generally, storing these images in current distributed file system means that for each uploaded photo we have kept *three* files' metadata in DFS's namespace and another *three* in underground local file systems. The first amplification is the *N-Form* issue: redundant metadata of DFS. Keeping redundant metadata means that DFS has to use more storage space to save, more I/O bandwidth to load, more memory to cache, and more requests to serve. The second amplification is the

1-N Mapping issue: redundant metadata of local file system. For each DFS file read, user has to firstly look for metadata in the local file system, and secondly read in the data content. It harms the I/O latency of small files.

Photo galleries, although important, might not be a sufficient reason for overhauling current file system implementations. Yet there are diverse applications, such as multimedia storages, online document workspaces, GIS applications and gene sequencing, that want to save large scale small files. Although these applications are often using the file system as a lightweight "key/value store", replacing file systems with NoSQL databases[6, 7] is an oft-rejected option because it is undesirable to port existing code to use new API (e.g. KV) and because many NoSQL databases do not provide sufficient small data read or write latency[8].

To boost up small file access in distributed file systems, we have to remove metadata redundance and eliminate unnecessary file mappings. Unfortunately, current file system directory model (key/value) is not capable to pack files with one file name easily, and the storage structure does not do aggregations aggressively. Although there are many researches on small file access, few research has focus on *N-Form* and *1-N Mapping* issues that introduced by emerging Web applications.

2.2   Related Works

Large scale small file access is non trivial, especially for file systems that want to support both small file and large file efficiently. For large file access, data content transfer latency dominates the total latency. While, for small file access, data content is short, and the transfer latency is tiny. The latency of metadata access and disk rotation for data have a huge impact on total latency.

File systems, which split metadata as `dentry` and `inode`, performs bad on small files. Each file access has to lookup at lease three disjoint positions, `dentry`, `inode`, and `block`, to get the data. As a result, numerous random accesses are generated with concurrent file access. And these random accesses further increase the rotation latency. Thus, C-FFS[17] and Ceph[18] embedded file `inode` in `dentry` to eliminate indirect access in file lookup stage. ReiserFS[14] tailed small file data with metadata to eliminate indirect access in data read stage. Although combining `dentry` and `inode` has complicated hard link implementation, it has improved small file performance a lot. We evolve this idea further to combine some file metadata, small file content, and attributes together to decrease latency.

Lots of small files also overwhelm directory indexing structures. Classic B+ tree is used in Ext3, Lustre and many other file systems. It is $O(log_k N)$ and terrible slow for large directories. Thus, GIGA+[20] and SkyFS[21] had proposed to use extendible hashing[16]

to manage large directories. These works are valuable for us to build a low latency $O(1)$ metadata service.

On the other hand, small file access means lots of random I/Os. When creating small files, blocks are allocated before doing I/O operations. And when doing I/Os, writes are somewhat randomized. It is fairly slow for hard disks. Thus, file systems, e.g. hFS[19], use log structured file format to aggregate small files and flush them as a whole to reduce disk rotations and utilize sequential I/O bandwidth. We carry on this idea and design a scatter and aggregation approach to set up many sequential data flows to make use of locality and parallelism.

For distributed file systems (abbr. DFS), things are even worse. DFS need another access stage on storage nodes' local file systems. For example, in Lustre[29], PVFS[30], and Ceph, file content is distributed on a set of nodes with a specific local file name. For each read, storage server has to lookup, open, read, and close the local file. It is inefficient for latency sensitive small file read. Thus, Michael et al.[31] had proposed to remove the metafile in PVFS, which is not needed for small file, to improve small file performance. Philip et al.[32] had integrated five approaches, e.g. batched metadata commit and new I/O flows, together to improve small file performance. However, these researches hasn't essentially solved the problem of file mappings between DFS and local file system. While, we propose a new directory based file aggregation approach to remove unnecessary mappings.

Researchers[15, 25] had proposed to use table interface to solve small files' problems. However, these works were fragmented and had not integrated and evaluated in one system. Besides, comparing with columnar databases[5, 6, 7], we have a similar column oriented table model. However, our directory table is schema-less and can be altered dynamically, and the storage structures are optimized for unstructured data access.

In all, although researches had addressed some issues of small files, they hadn't solved the *N-Form* and *1-N Mapping* problems fundamentally. Combining metadata and data or using log structured file format, has no effect on eliminating redundant metadata, and didn't solve the *N-Form* problem. Similarly, approaches proposed for PVFS have not eliminated file mappings, and didn't solve the *1-N Mapping* problem.

3   Design Issues

To address *N-Form* and *1-N Mapping* issues and build an efficient small file store, we focus on evolving directory model, storage structures, and interface. To minimize space consumption of DFS metadata, we propose a tabular directory model to group similar files together. To store small files efficiently, we propose a directory based

(a) Plain Model (N Copy)

(b) Plain Model w/ xattr (packing)

(c) Column Oriented Tabular Model (CODIR)

Figure 1: Directory Model for N-Form Problem. *Imagea-s and imagea-m are the small and medium forms of imagea. Plain model is either (a) metadata costing or (b) difficult to use. Column oriented tabular model resolves both these difficulties.*

storage structure to scatter and aggregate files based on columns. Finally, to access these columns of files more easily and quickly, we propose a new query interface and support searching.

3.1   Tabular Directory Model

Typically, current file system directory is a plain (key/value) table with three columns: *name*, *metadata*(including extended attributes, abbr. xattr), and *data*. Naturally, for a photo with several sized images, each image is saved in a standalone file, shown by Figure 1(a). Obviously, we have *N* copies of file metadata. Alternatively, we can group these *N* images together by xattr and self-packing to eliminate redundant metadata and boost up metadata accessing. However, it is non-trivial to store several image files with one file name in plain directory model. Firstly, it is impossible to save large data content in xattr. Linux VFS `xattr` only supports setting and getting at most 64KB buffer. Even worse, local file system, e.g. Ext3, sets up a smaller buffer limit at 4KB. Secondly, user has to pack multiple image files into one file's data content and record offset and length of each image in xattr, shown in Figure 1(b). Although it is possible to save *N* image files under one name, this approach is inconvenient for users. Other applications have to build their own packing and unpacking framework.

To address the *N-Form* issue, we propose a column-oriented tabular directory model (abbr. CODIR). Shown by Figure 1(c), each file has two basic columns: *name*, and *metadata*; and multiple *data columns*: different file forms, xattrs, or tags. Thus, several file forms derived from one file can be stored in different columns under one file name. For example, all file forms origin from *imagea* are saved with name *imagea*. Each form (e.g. *imagea-s*, the small sized *imagea*) is saved in one column (e.g. *Col 1*). By utilizing this tabular directory model, users can not only save space, but also store and retrieve column data much more easier. We have proposed an API to access file columns in Section 3.3.

3.1.1 Sparse Table

In CODIR's table, there might be unlimited columns. If columns' metadata are saved in dentries, then dentries are variably sized. Directory operations such as `seekdir` would be difficult to implement. Thus, we make dentries have a fixed size. Each dentry contains fixed number of *direct* columns and unlimited *indirect* columns. The frequently accessed data should be put in the direct columns, while the infrequently accessed ones should be put in the indirect columns.

As directory contains lots of columns, it might be sparse. Some dentries do not have valid values for some columns. Obviously, these blank cells do not need to be saved. For each column, valid data are sent to storage server and aggregated in a column file, shown by Figure 2. The data written offsets must be kept in dentries (as metadata) for read and write in the future. Data offsets of direct columns are stored in dentries, while data offsets of indirect columns are stored in an *indirect file*. Thus, accessing data of direct columns are faster than that of indirect columns. Besides, if the table is sparse, it is space consumptive for indirect columns to keep these offsets in a fixed sized array with holes. Thus, the indirect file is append-only with a tail index record. For each update, the update offset and an index record (indicates column ranges and internal offsets) are appending to tail of the file. On reading, the index record is read in firstly to determine which file position should be read next.

Although the tabular directory supports sparse table, we argue that sparse tables decrease directory performance and waste more space. To eliminate sparse tables in practice, users should only group files with similar properties in one directory. For example, do not mix user photos and documents together in a directory. Fortunately, this simple rule always keeps because of our habit to classify things before storing.

3.1.2 Advantages of CODIR

The tabular directory model can save DFS's metadata for *N-Form* problem. Other than metadata saving, this model has some other advantages. Firstly, it is more efficient to store larger data than extended attributes. In many implementations, `xattr` attaches to file metadata, and is not very efficient at storing large data content. While, CODIR permits to store several variable length of data in it. And these data are not required to store with metadata. Secondly, it is more efficient to do data compression on columns. Shown by Figure 2, we aggregate one or several columns' content (e.g. *Metadata*) to one storage file. Thus, for each directory, we have already grouped similar data together. It is efficient for compressor to get a better zip ratio. For example, we had compressed the *Metadata* file for about *25 times* smaller by using LZO[34] in PFS. Thirdly, it is more efficient to
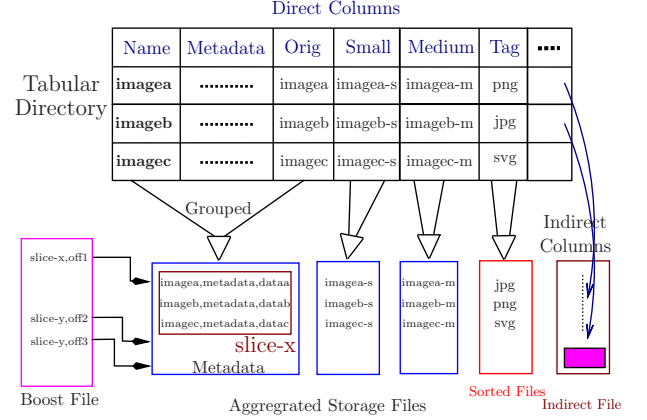

Figure 2: Storage File Mappings. *The first three columns are grouped and aggregated as one storage file, the next two columns are aggregated as independent storage files, and the Tag column is saved in a sorted file. Boost File accelerates metadata access. Indirect file contains indirect columns' metadata.*

do bulk load on specific columns. For example, file content of the same column are stored in the same storage file, thus, it is suit for bulk loading. This is valuable for Web applications that needs bulk file loads in the cache filling stage. For example, in photo galleries, loading small sized images (aka thumbnails) to cache are very fast with CODIR, because these are all sequential I/Os.

3.2 Structured Storage

Current distributed file systems mostly map their files to files of local file system on storage nodes. Thus, even you resolve the *N-Form* problem, you still have the *1-N Mapping* problem that increases small file access latency. Shown by Figure 2, to reduce this storage file metadata lookup latency, we propose a directory-based file aggregation approach. This aggregation reduces metadata, and is not only saving disk space but also saving memory (no need to cache lots of dentires of local file system). Small files in one directory are logically grouped in one big storage file to eliminate redundant metadata. Each data access is actually a read or write in this big file. Thus, ideally, for each directory, we only have to lookup and open one storage file. By caching this file's metadata, data I/O of small files does not need to do metadata lookups in local file system thereafter. Meanwhile, a file system directory is naturally a locality set. Files in one directory might be accessed contiguously. Combining files of the same directory together not only doesn't ruin, but also improve the spatial locality.

At the same time, in CODIR, data contents are aggregated by column by default. The rationale of splitting one file's columns into several storage files is to re-

duce unnecessary data loading in column fetching (and prefetching). For example, in a photo gallery, when you visit a list of previews, it is unnecessary and time consuming to read in every photo's original file. However, co-accessed columns could be grouped and stored together. For example, in Figure 2, we group file's metadata and data content together. Each file lookup, that read-in the metadata, would prefetch the data content to cache either. Subsequent data access would hit in the cache.

3.2.1 Aggregated Storage Files

We use log structure storage format for the aggregated file data. For small files, it is more space efficient than block based storage format because of no internal fragments[9]. New arriving data is appended to tail of the file. Removing files are only metadata operations and do not need to update data file. Every small file write must contain the full file content. Thus, file rewrite has to be transformed to read-modify-write (RMW).

After lots of file deletions or modifications, the aggregated data file might be filled with holes. These holes must be detected and garbage collected to save storage space. For each directory, metadata that needed in garbage collecting (GC) co-locates with the data. Thus, the GC process runs incrementally against directories on each storage node to not interfere with storage process.

The RMW issue of file rewritten might be a performance bottleneck. However, in Web applications, there are fewer file rewritten than file creations and reads. Further more, a) read small file (in stage R) means less time to stall, and b) lots of rewritten are actually issued after file reads (which matches the order of stages R and M). Thus, we argue that RMW issue would has more impacts on larger files than smaller ones and it is worth sacrificing rewrite performance to optimize file creation and read.

3.2.2 Structured Storage Files

Dentries are stored in an aggregated storage file: *Metadata*. Number of dentries grouped as a *slice*, which is the basic metadata read and write unit. Slice is different from directory partition in GIGA+[20] and SkyFS[21]. Slice uses hash table indexing, while directory partition uses out-of-core indexing. It reduces access latency for small files by eliminating out-of-core indexing.

It is slow to do slice lookups in the log structure *Metadata* file. Thus, an index file is built, *Boost File*, for the metadata file to speed up lookups. The index keeps a record <slice, offset> for each active slice. The *Boost File* is a supplement of the *Metadata* file. It is guaranteed to be updated on every slice write. Even if corrupted, it can be regenerated by scanning the *Metadata* file. Slice in *Metadata* file has a header, which contains magic
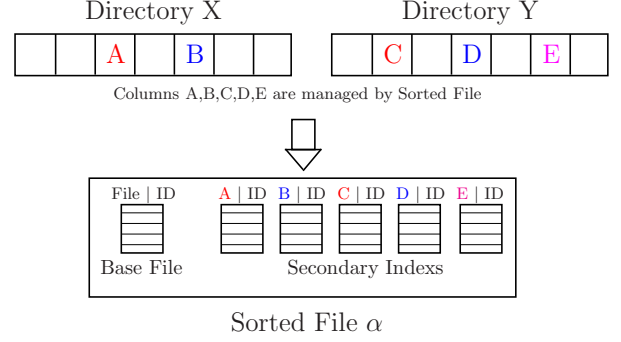

Figure 3: Sorted File Format. *Columns of directory X and Y are directed to Sorted File α. Each database contains one base file and several secondary indexes.*

numbers and special inequations. These can be used as border markers of slice in scaning.

Semantic searches, e.g. "*Top-K query: find the K hot jpeg files*", are widely used in many applications[27, 28]. In CODIR, columns could be used to store application specific tags or xattrs. These tags and xattrs are used to serve semantic searches. Researches[27, 28] have found that using an external database ruins the spatial query locality, and proposed many semantic search approaches on large scale file systems. However, most of these approaches depend on offline file scanning, and are not capable to handle mutable file systems (for Web2.0 applications). To support online indexing, we propose to **1)** use internal *sorted* file instead of external database, **2)** use Star schema to maintain relationships among columns, and **3)** only manage user-defined subset of directory columns to decrease file subset size that we handle. At the same time, sorted file is tightly coupled with file system to achieve lower latency. For example, shown by Figure 3, columns *A, B* from directory *X*, and columns *C, D, E* from directory *Y* are kept in one sorted file α. Each sorted file contains a base file to save basic mappings from file name to unique file id(File|ID), and several secondary indexes for each column(e.g. A|ID). In this way, it is easy to dynamically add new column or delete old column. In all, this approach not only maintains spatial locality, but also provides more flexibility. The sorted files could be key/value stores such as BerkeleyDB[33]. Thus, in Figure 2, query "*find all the jpeg files*" can be executed by searching the *sorted* file and locating file *imageb* directly, rather than scanning the whole directory to find the file.

3.3 Tabular Query Interface

Firstly, as we have extended the directory model to store multiple columns, we need an interface to get/set table cells. Secondly, in large scale file system, to find a file subset, searching is more meaningful than traversing. As

Category	Prototype
table	`tab_t` **opentab**`(char *dir)`
	closetab`(tab_t th)`
	tabschema`(tab_t th, colid, colname)`
	readtab`(tab_t th, file,`*colname*`,buf,len)`
	writetab`(tab_t th,file,`*colname*`,buf,len)`
	bulkload`(tab_t th, `*colname*`, buf, len)`
raw	**tread**`(filepath, `*colid*`, off, buf, len)`
	twrite`(filepath, `*colid*`, off, buf, len)`
sorted	`sorted_t` **opensorted**`(char *name)`
	closesorted`(sorted_t s)`
	search`(sorted_t s, char *expr) //+`**JOIN**
	set`(sorted_t s, filepath, char *kvlist)`
	delete`(sorted_t s, filepath, char *key)`
	update`(sorted_t s, filepath, char *kv)`
	test`(sorted_t s, filepath, char *key)`
getxattr	Wrapper Examples
	`(filepath, EMBEDDED_CMD_STRING, buf,len)`
opentab	`"pfs.table.opentab.DIR"`
bulkload	`"pfs.table.bulkload.TH.colname"`
search	`"pfs.sorted.search.S.R:`$k1 < 10\&k2 > 20$`"`
setxattr	Wrapper Examples
	`(filepath, EMBEDDED_CMD_STRING, buf,len)`
writetab	`"pfs.table.writetab.TH.file.colname"`
set	`"pfs.sorted.set.S.file.k1=v1,k2=v2"`

Table 1: User Level Interface for File and Directory

a result, we provide a user level tabular maintain and search interface, and also wrapped it to fit the existing `xattr` interface[2]: `getxattr` and `setxattr`.

As shown by Table 1, we defined three API groups for CODIR. The **table** group is for applications. When opening the directory table, user can read/write specific file column and do bulk load on columns. The **raw** group is for library developers. They can read/write specific file column by column ID instead of name directly. The **sorted** group is for tag operations. A list of key/value pairs can be attached with each file. KV pairs from specific columns of directories are grouped in one sorted file. These pairs are automatically indexed. By using sorted file, queries and joins on file tags are more quickly than brute-force scanning the whole directory tree.

The `xattr` wrappers encode tabular operations in strings. For example, string "pfs.sorted.set.S.file.k1=v1, k2=v2" represents a tuple ($<$file, $k1 = v1; k2 = v2>$) addition on sorted file `S`. String "pfs.sorted.search.S.R: $k1 < 10\&k2 > 20$" represents a range (`R:`) search on sorted file `S` that satisfies $k1 < 10\&k2 > 20$. The search operation returns an iterator, which allows you iterate the result set.

---

[2]Note that, it is impossible to do large data read/write with this wrapper because of the 64KB buffer limit in Linux VFS.
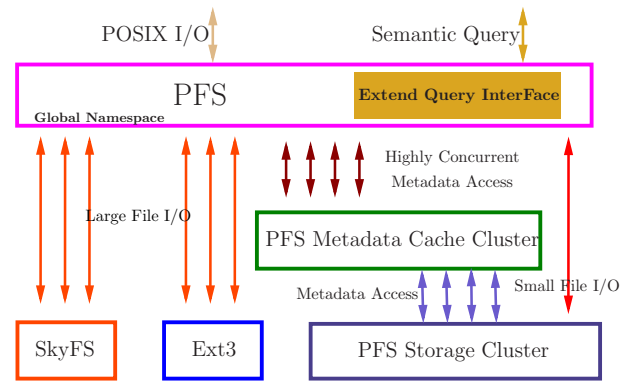

Figure 4: Architecture of PFS

4 Implementation Details

We have implemented the tabular directory model, storage structures, and interface in a prototype distributed file system PFS for Web applications that need to access large scale small files. Large files are handled by other DFSs. Shown by Figure 4, PFS contains a FUSE[23] client, a dedicated metadata and storage cluster to handle small files, and relies on Ext3, SkyFS (or Ceph, PVFS2) to handle large files. However, CODIR might be implemented in other DFSs to improve their small file performance.

The metadata cluster is a collaborative cache layer over storage servers to provide low latency metadata service. Metadata modifications are flushed back to disk asynchronously. The metadata service latency dominates the whole file access latency for extremely small files. As a result, caching metadata in memory decreases the total latency greatly.

The storage cluster is a shared storage pool. Metadata and data of small files are aggregated by directory and scattered to all the servers. Thus, for each directory, there are several sequential data write flows to exhaust large scale concurrent file creations.

4.1 Distinguish Small Files

Small files are typically less than 1MB[25]. In many photo galleries, we found that the average file size is at 32KB. Thus, for higher degree of confidence, we make 2MB as the transition point from small file to large file. But, at file `open` time, there is no hint on file length. There are two approaches to distinguish small files. The first one is a *static* approach. The parent directory is marked as *static-small* and all file creations in it automatically inherit the property. The second one is a *dynamic* approach. The parent directory is marked as *dynamic-small*. Static-small doesn't allow a small file to transform to a large one automatically, while dynamic-

small allows that.

At `open` time, we use parent directory's property to guide following I/O operations. When it is static-small, the followings are all small file operations. When it is none, the following are all large file operations. When it is dynamic-small, the PFS client probes on each file write and determine if the file is larger than 2MB to be able to change from small file to large file or vice verse.

4.2 Elastic Metadata Service

Metadata service is the bottleneck in many distributed file systems[29, 25], especially for small files. Even in scalable file systems[18, 20, 30], metadata performance has been slowed down by storage structures(e.g. out-of-core indexing). Thus, CODIR and file aggregation, that decrease and aggregate metadata, could increase metadata performance either. Meanwhile, for large scale small file creations, metadata service has to be able to scale out to large number of machines. But, in practice, it is difficult to estimate how many machines are enough for user requests before starting up. Thus, it requires *elastic* metadata service which supports adding or removing machines online.

To scale out to a large cluster and keep low latency, the metadata service of PFS uses extendible hashing[20] to manage slices of directory tables. With lots of file creations, slice splits to store more files. The new slice might be managed on the same metadata server (aka. local split) or other remote servers (aka. remote split). If the new slice is placed on remote servers, more server processing capability might be used. However, there always exists data transfer overhead in remote splits. Thus, in PFS, we make slices firstly split to remote servers to exploit more machines, and secondly split at the same server to eliminate data transfer overhead(named Two-Stage split).

To scale elastically, PFS uses consistent hashing to distributed table slices to servers. Thus, when adding or removing machines, slices that need to be transferred to remote machines are least[13]. At the same time, because of metadata service of PFS is a caching service, slices that need to be transferred could be simply dropped at original server and reload on demand by the new server. This approach mitigates data transfers on machine changes, and results in lower service interruption.

4.3 Parallel Small File I/O

In practice, applications are expected to use one directory to store one type of files as far as possible. Thus, PFS storage cluster aggregates small files based on directory. I/O operations for small files are first scattered to a list of servers, and then aggregated by directory. PFS exploits both spatial and temporal locality to boost up small file I/O performance.
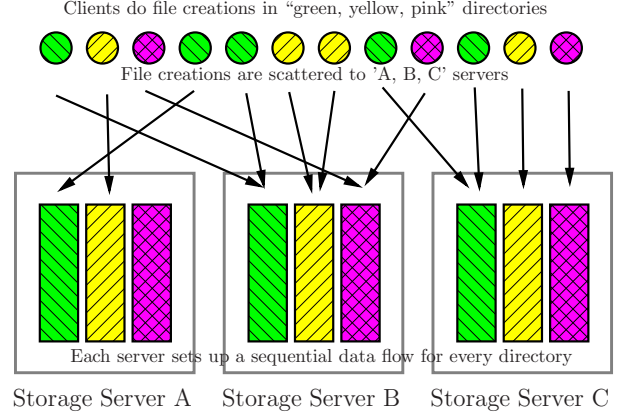

Figure 5: I/O Write Flow: Scatter and Aggregation

Each server opens one aggregated file for a directory. Instead of in-place modification, file creations and writes are appended to the file. If there are lots of file creations, the creation flow is directed to the file to exploit sequential I/O bandwidth. The temporal continuous data flows (split by column) are transformed to spatial continuous data streams. Further more, because of file creations in one directory are scattered to several servers, actually PFS sets up several sequential I/O flows to exhaust lots of concurrent file creations, shown by Figure 5. Note that, every file written must be an integrated writes.

At the same time, columns are stored in separated storage files by default. Thus, each file has already aggregated contents that has similar property. For example, in Figure 2, storage file for column 'Medium' contains all the files of medium size. For application requests that need to scan all medium images (violence detection), it is efficient to do sequential reads on the storage file. As columnar store in database, this exploits the vertical spatial locality in directory table.

5 Experimental Evaluation

Our experimental evaluations focus on the typical small file access pattern, and answer three questions: a) Does it save space by adopting CODIR for *NForm* problems? b) What are the metadata and small file performance tradeoffs for our design and implementation choices? c) What lessons we have learned from PFS?

5.1 Experimental Setup

We deploy PFS on two commodity clusters. For cluster A (50 nodes), the hardware configuration of a blade has one quad-core 1.6GHz Intel Xeon processors, 4GB memory, GigE NIC, and connected by 10GigE switches. For cluster B (80 nodes), each blade has three quad-core 1.6GHz Intel Xeon processors, 23GB memory, GigE
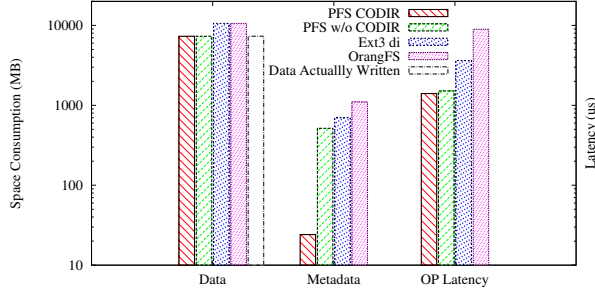
Figure 6: Space consumption of saving 1.5 million data contents in PFS, Ext3, OrangeFS. *By adopting CODIR, the metadata space consumption is only 1/20 comparing with PFS without CODIR. Meanwhile, CODIR uses less data and metadata space comparing with Ext3 and OrangeFS, and has the lowest latency for this test.*
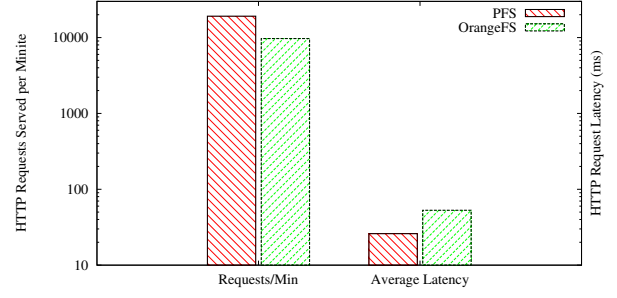
Figure 7: HTTP performance of a GIS Tile Server based on OrangeFS and PFS. *The served HTTP request rate of PFS with CODIR is twice of OrangeFS. With less metadata, file read is more faster.*

NIC and switches. Cluster A has a better network, while cluster B has more large memory machines. All nodes were running the Linux 2.6.16-60 kernel(Suse release) and PFS stores file system images on one 7200rpm SATA disk.

5.2 NForm Promblem

To find out whether CODIR can save metadata space for *NForm* problem, we run a micro benchmark to write 1,500,000 data contents (total data size is 7.9GB) on one blade of cluster A. For PFS with CODIR, data contents are saved as *three* columns in 500,000 files; for other configurations (PFS without CODIR, Ext3 with dir_index[3], and OrangeFS), data contents are saved in 1,500,000 files. Figure 6 plots the space consumption of metadata and data and the average write latency. By adopting CODIR, PFS performs best at saving metadata cost, and has the lowest average write latency. Meanwhile, with small file aggregation, the data content are saved continuously. It is more space saving than block based storages such as Ext3 and OrangeFS.

Second, we setup a GIS tile server[4] which exports a map tile service via HTTP to evaluate the small file read performance. The tile server keeps one million tile images in OrangeFS and PFS. In OrangeFS, the tiles are saved in 1000 subdirectories with 1000 tiles in each subdir. In PFS with CODIR, the tiles are saved in 200,000 files with 5 tiles (columns) in each file. We run *JMeter* to lookup tiles randomly with 10 threads over Giga Ethernet against OrangeFS and PFS based tile server and get the throughput and latency result in Figure 7. PFS with

---

[3] dir_index option enables hashed-tree indexing for Ext3, and it speeds up the metadata performace.

[4] Tiles are generated from *OpenStreetMap*'s database, and we use TileStach as the HTTP frontend.

CODIR is two times faster than OrangeFS.

Lesson #1: CODIR could not only save metadata space, but also increase small file I/O performance.

5.3 Structured Storage and Interface

We used synthetic benchmarks cluster postmark [24] to create, read, append, and delete files to evaluate the small file I/O performance, IOZone to investigate large file I/O, and mdtest to insert zero-byte files in shared directory to evaluate metadata performance.

5.3.1 File Aggregation

Firstly, we want to find out the scalability of small file aggregation. We ran cluster postmark in cluster A to investigate the scalability of small file(in range [1,1k]) I/O. In each run, we fixed the number of servers, and variate the number of clients to change loads. Figure 8(a) plots the file write rate varying with different number of clients and servers. When there were fewer servers, e.g. the #Server=1 line, system saturated quickly and the file write rate is not linearly increasing. When there are more servers, e.g. the #Server=32 line, file write rate keeps increasing linearly. At the same time, in cluster A, we find out that write bandwidth of small files is linear correlated with number of clients if the servers are not saturated(Figure 8(b)).

Lesson #2: Small file ($<$1KB) I/O is linearly scalable with adding more servers and clients.

Secondly, we want to know whether file aggregation is efficient for concurrent small file creates. We ran cluster postmark (47 processes) in cluster A with file size from 1KB to 512KB, and compared the file create rates and bandwidth between PFS and OrangeFS. Both PFS and OrangeFS were configured with 32 metadata and data servers. Figure 9 plots the aggregated create rates and write bandwidth. The file create rate and write bandwidth of PFS is more than 6000% of O-
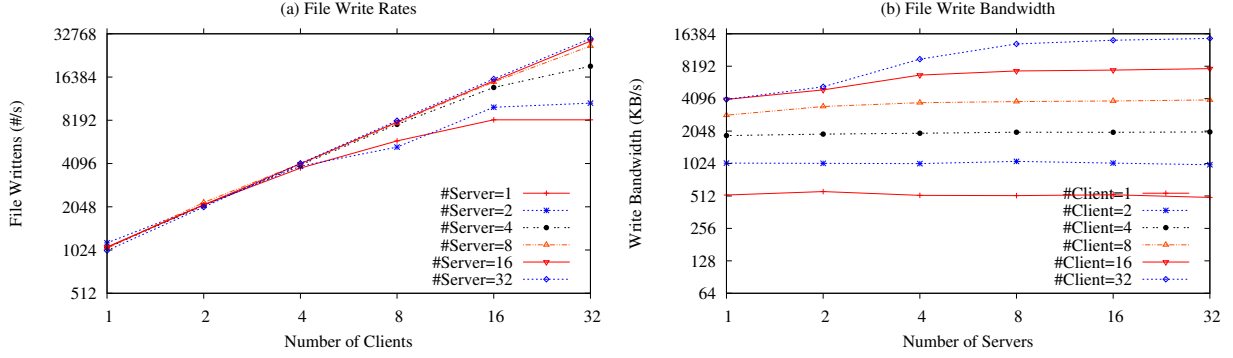
Figure 8: Small File I/O Scalability in Range [1,1K]. *Small file write rates increase linearly with more nodes until the servers are saturated. Double the number of servers almost doubles the aggregated file write bandwidth. However, write bandwidth doesn't increase with more servers because of the limited network message rate.*
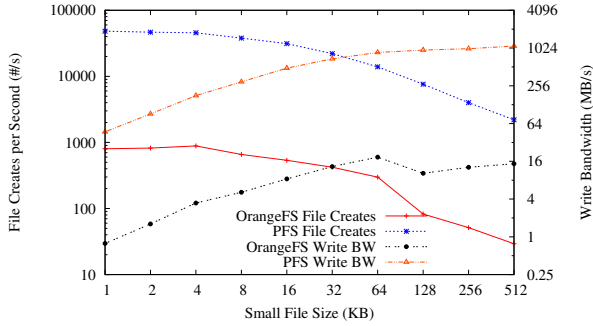

Figure 9: Small File Create Rates and Write Bandwidth. *For OrangeFS, clients totally creates 4,700,000 files; while for PFS, clients totally creates 47,000,000 files. With increasing the file size, file create rate decreases. The aggregated write bandwidth of PFS with 512KB file is more than 1GB per second.*
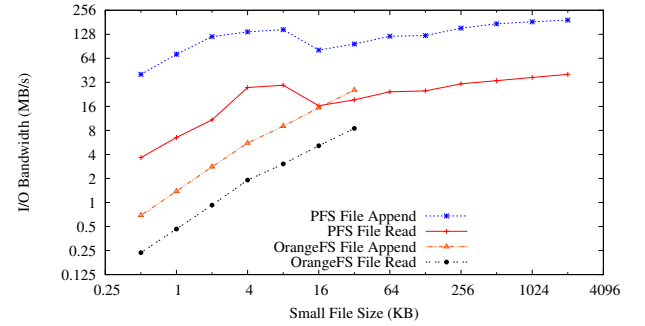
Figure 10: Small File Read/Append Bandwidth. *File read and append bandwidth of PFS is larger than that of O-rangeFS. The random read bandwidth of PFS drops at 8KB because the network MTU is only 5100B. Both read and append bandwidth drops at 16KB because PFS client uses 16KB buffer page. More pages means longer latency to read and write. The RMW issue in file append has little impaction on I/O bandwidth.*

rangeFS. By enlarging the file size, file create rates of both file systems are decreasing, while the write bandwidth are increasing. OrangeFS performs worse when small files are larger than 64KB. While, the aggregated write bandwidth of PFS with 512KB files is more than 1GB/s. This bandwidth promotion benefits from our scatter and aggregation small I/O flows in Figure 5.

Lesson #3: Directory based small file aggregation plus appropriate I/O scatter framework can greatly improve concurrent small file creation performance.

Thirdly, we want to know how RMW issue affect the small file I/O performance. We ran `cluster postmark` (75 processes) in cluster B to randomly read or append files (1:1) with file size from 512B to 2MB on PFS and OrangeFS. Both PFS and OrangeFS were configured with 75 metadata and data servers[5], and

−`noatime` option. Figure 10[6] plots the I/O bandwidth of file read and append. Both file read and append bandwidth of PFS almost increase with larger file size, and outperform that of OrangeFS. Although there are RMW issues in file append, the append bandwidth increases steadily to the maximum network bandwidth.

Lesson #4: RMW issue in file modifications has little impaction on small file I/O performance.

Finally, we run a real world application to untar 1,000,000 short gene sequence files (most less than 1K-B) on PFS, Ceph, and OrangeFS. Ceph is configured to

---

[5]OrangeFS failed to run cluster postmark when files are larger than

[6]The aggregated network bandwidth of cluster B is about 200MB/s, and it is limited by the top-level GigEther switch.

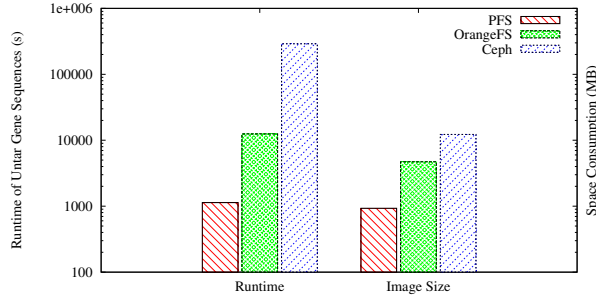32KB. This might be a bug of multi-metadata server implementation.

Figure 11: Runtime and Space Consumption of Untar 1 million Gene Sequences. *Untar short gene sequences in PFS is fastest and most space efficient with respect to OrangeFS and Ceph. This test runs on one machine.*
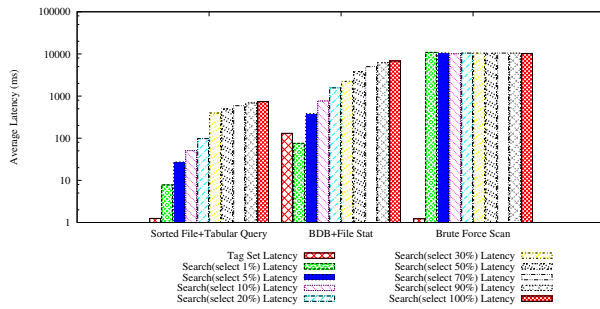

Figure 12: Tag Set and Search Latency. *Using sorted file + tabular query interface, tag set and search are faster than external database(BDB) and brute-force scanning search. Especially, external database approach is slow at importing tags to database. And for queries of lower selectivity, sorted file + tabular query performs much more better.*

store one data and metadata replica[7]. Figure 11 presents the total space consumption and runtime. Benefit from small file aggregation, PFS is much more space efficient and faster than Ceph and OrangeFS. It is 10 times faster than OrangeFS, and the image size is only 19.7% of OrangeFS and 7.6% of Ceph.

5.3.2 Sorted File and Tabular Query

In this test, we set up a set of files (1 million) and gave each file a specific tag. For CODIR model, file tags were saved in sorted file; for traditional directory model, file tags were saved in extended attributes; while for database model, file tags were saved in external BerkeleyDB[8]. For user queries, we test the tabular query interface, and compare it with brute-force scanning and

---

[7]We had tried to tune many different configuration variables, Ceph ran slowly still and finished the test in 3 days.

[8]The database is configured without transaction support, and in Concurrent Data Store mode(CDB).
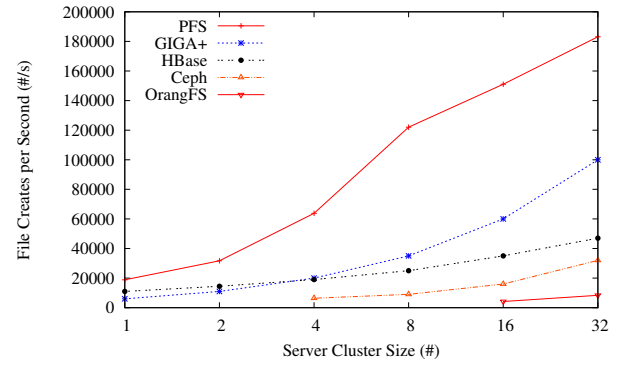

Figure 13: Metadata Scalability of PFS. *PFS delivers a peak throughput of roughly 180,000 file creates per second. The network message rate limits PFS's ability to match the ideal linear scalability. It is fast than GIGA+ because of Two-Stage split strategy and none out-of-core indexing.*

external database queries. The first two tests perform through `xattr` syscall interface. Figure 12 plots tag set and search latency for different selectivity. Comparing with external BDB approach, using tightly coupled sorted files decreases both tag set and search latency. Comparing with brute-force scanning, database based searches are sensitive to query selectivity. For queries with lower selectivity, it performs more better, because more files are ignored to stat.

Lesson #5: Sorted file with tabular query interface performs better than external database indexing and classic directory scanning.

5.4 Elastic Metadata Service

Firstly, we want to find out whether PFS could scale out. We used `mdtest` to evaluate metadata performance of PFS, GIGA+, OrangeFS, Ceph, and HBase in cluster A. Our configuration used 8 processes per client to simultaneously create files in a shared directory, and the number of files created is proportional to the number of servers: 12.8 million files on 32 servers. HBase is used to simulate the metadata service of Google's next generation file system[25]. PFS use FUSE client to complete the test, and each file `creat()` results in two RPC calls to server: `getattr()` to check if a file exists, and `creat()` to create and return file's attributes.

We reuse the experiment results of GIGA+, Ceph, and HBase performed on different clusters from the original paper[20, 18]. That cluster used quad-core 2.83GHz machines with 10GigE NICs and switches(faster than ours). Figure 13 compares the aggregated file create throughput, in file creates per second. PFS scales *linearly* up to the size of 32-servers configuration, and can sustain 180,000 file creates per second. PFS also demonstrated
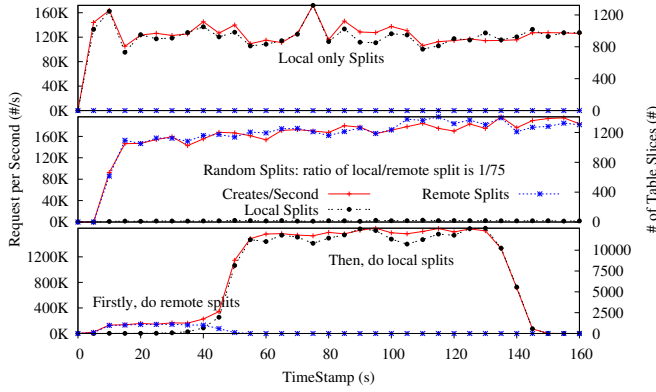
Figure 14: Local Only, Random, and Two-Stage Splits. *Local only splits couldn't exploit more nodes to improve service rate; Random splits introduces too many remote splits and network latency. Both are not efficient slice split strategy for large scale metadata service. Two-Stage splits performs much more better than them.*

scalable performance for the concurrent lookup workload delivering a throughput of more than 410,000 file lookups per seconds(not shown). It benefits from our Two-Stage slice split strategy and removing out-of-core indexing. Although OrangeFS (experimental version) had also implemented a GIGA+ like distributed directories, we found that it performs worse than GIGA+(the bottom line).

Lesson #6: By splitting directory and removing out-of-core index, metadata performance imporves greatly.

5.4.1 Slice Split

Next, we want to find out the metadata behavior of our Two-Stage slice split strategy(firstly split to remote servers, then split at the local servers). The test runs on cluster B with 75 metadata servers and clients. Clients issued millions of file creation requests. Figure 14 plots the immediate request rate and numbers of two types of splits in three runs with local only, random, and Two-Stage split strategy. It is obviously that Two-Stage splits are far more better than local only and random splits[9]. After entering into local split mode, serviced request rate increases for about *10 times* larger.

Lesson #7: By exploiting parallelism(more nodes) and locality(local only splits), a simple Two-Stage splits strategy increases the metadata performance greatly.

5.4.2 Elastic Speedup

Finally, we want to find out whether the metadata service of PFS could scale up and down elastically. To understand how server changes impacts on the latency of file operations, we run a test to do file creates, lookup-

---

[9]To plot the two stage clearly, we slow the transition from remote splits to local splits in Two-Stage strategy intentionally.
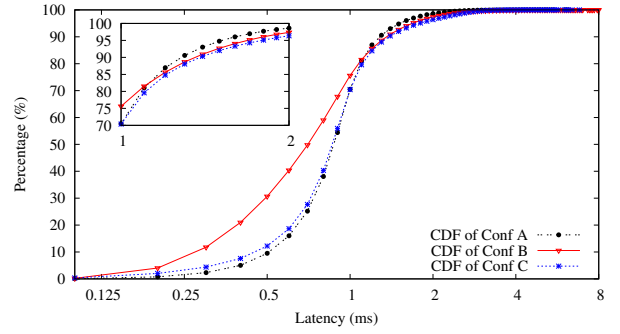

Figure 15: Cumulative Distribution Function of three Runs. *Online a server shifts the curve to lower latency(Conf A to B). Offline a server shifts the curve to higher latency(Conf B to C). It means that PFS scales up and down elastically.*

s, and removes in three configuration. `Conf A` ran the test on *one* server without any online/offline operation. In `Conf B`, we ran the test on one server, and online a new server in the test. In `Conf C`, we ran the test on two server, and offline an existing server in the test. Figure 15 plots the cumulative distribution function of request latency in three configurations. Adding a server made the average request latency smaller, while deleting a server made the latency larger. For `Conf A,B,C`, half of the requests' latency are under 1ms, and 99% of them are under 2.2, 2.5, and 2.7ms.

5.5 Relayed Large File I/O

We want to find out whether our optimizations for small files has affected the large file I/O performance. To understand the tradeoffs of our design choices on large file I/O, we ran `IOZone` on cluster A to investigate it. In this test, we used Ext3(one node) and SkyFS(30 nodes) as our large file store separately , and all the I/O requests are routed to Ext3 or SkyFS data servers. `IOZone` is configured with 1MB record size, and the total file size is *three* times larger than memory size. Shown by Figure 16, comparing with Ext3's sequential read, FUSE client of PFS performs bad that the overhead is 33%. However, comparing with SkyFS, the FUSE client performs better. Because SkyFS also uses FUSE client, we consider that this strange behavior might be some characters of FUSE.

Lesson #8: Even considering the FUSE overhead, large file I/O in PFS has not gone bad in large cluster.

6 Conclusion and Future Work

In this paper we address the emerging requirement for large scale small file access. We focus on two common problems: *N-Form* and *1-N Mapping* in distributed file systems, and evolve the directory model, interface and storage structures for small files. We propose to use the
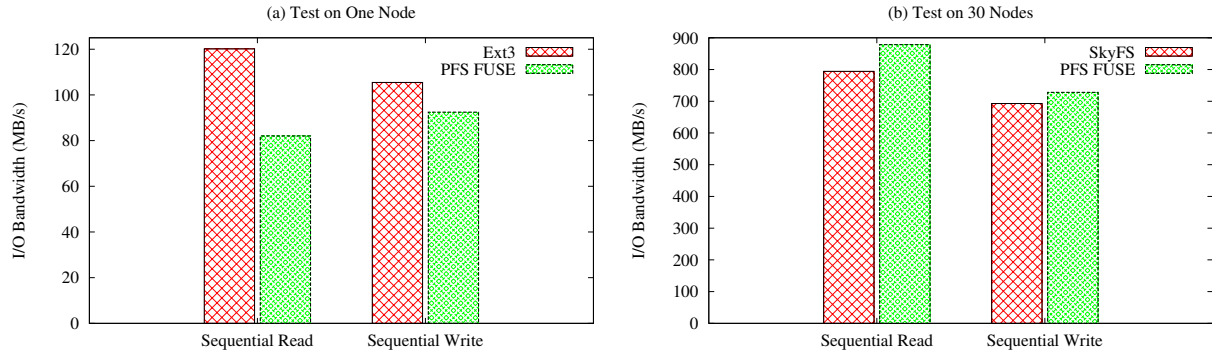
Figure 16: Large File I/O Bandwidth of PFS. *With respect to Ext3, PFS FUSE performs bad on both sequential read and write, and introduce 33% and 15% overhead. However, comparing with SkyFS in large cluster, PFS performs better.*

tabular directory model and interface. In this way, file system can not only save lots of spaces, but also increase the efficiency of metadata service. On the other hand, we propose to use directory based small file scatter and aggregation. Thus, file system can greatly improve small file I/O performance. Our evaluation shows that by using CODIR model, interface, and storage structures, file system achieves best-in-class performance on a commodity cluster. Especially, metadata and small file performance scale linearly with more nodes.

PFS is open source at `github.com` for researchers who want to test, tune or integrate it. In the future, we want to enhance and extend the POSIX interface for small files.

References

[1] Beaver, Doug and Kumar, Sanjeev and Li, Harry C. and Sobel, Jason and Vajgel, Peter, *inding a needle in Haystack: facebook's photo storage*. Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI 2010).

[2] Wensong Zhang, *Taobao File System*, `http://rdc.taobao.com/blog/cs/?p=128`, (2010).

[3] Rongfeng Tang, Private Communication, (2011).

[4] Chang, Fay and Dean, Jeffrey and Ghemawat, Sanjay and Hsieh, Wilson C. and Wallach, Deborah A. and Burrows, Mike, et. al., *Bigtable: a distributed storage system for structured data*, Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation, (OSDI 2006)

[5] Stonebraker, Mike and Abadi, Daniel J. and Batkin, Adam and Chen, Xuedong, et. al., *C-store: a column-oriented DBMS*, Proceedings of the 31st international conference on Very large data bases, (VLDB 2005)

[6] Lakshman, Avinash and Malik, Prashant, *Cassandra: a decentralized structured storage system*, SIGOPS Oper. Syst. Rev. vol 44, 35 - 40, (2010)

[7] HBase, `http://hbase.apache.org`, (2010)

[8] Cooper, Brian F. and Silberstein, Adam and Tam, Erwin and Ramakrishnan, Raghu and Sears, Russell, *Benchmarking cloud serving systems with YCSB*, Proceedings of the 1st ACM symposium on Cloud computing, (SoCC '10)

[9] Rosenblum, Mendel and Ousterhout, John K., *The design and implementation of a log-structured file system*, ACM Transactions on Computer Systems, February, (1992)

[10] *ZFS: the last word in file systems*, Sun Microsystems. September 14, (2004).

[11] *Network Appliance: File System Design for an NFS File Server Appliance*, NetApp Inc. (1995).

[12] Andr Allavena, Alan Demers and John Hopcroft, *Correctness of a Gossip-based Membership Protocol*, Proc. 24th ACM Symposium on Principles of Distributed Computing (PODC 2005).

[13] Karger, D.; Lehman, E.; Leighton, T.; Panigrahy, R.; Levine, M.; Lewin, D., *Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web*, Proceedings of the 29th Annual ACM Symposium on Theory of Computing, (1997)

[14] Hans Reiser. ReiserFS. `www.namesys.com`, (2004).

[15] Patil, S., G.A. Gibson, G.R. Ganger, J. Lopez, M. Polte, W. Tantisiriroj, L. Xiao, *In Search of an API for Scalable File Systems: Under the table or above it?*, USENIX Workshop on Hot Topics in Cloud Computing, (HotCloud 2009)

[16] Fagin, Ronald and Nievergelt, Jurg and Pippenger, Nicholas and Strong, H. Raymond, *Extendible hashing—a fast access method for dynamic files*, ACM Transaction Database System. (1979)

[17] Ganger, Gregory R. and Kaashoek, M. Frans, *Embedded inodes and explicit grouping: exploiting disk bandwidth for small files*, Proceedings of the annual conference on USENIX Annual Technical Conference, (USENIX ATC 1997)

[18] Weil, Sage A. and Brandt, Scott A. and Miller, Ethan L. and Long, Darrell D. E. and Maltzahn, Carlos, *Ceph: a scalable, high-performance distributed file system*, Proceedings of the 7th symposium on Operating systems design and implementation, (OSDI 2006)

[19] Zhang, Zhihui and Ghose, Kanad, *hFS: a hybrid file system prototype for improving small file and metadata performance*, Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, (EuroSys 2007)

[20] Swapnil Patil, Garth Gibson, *Scale and concurrency of GIGA+: file system directories with millions of files*, Proceedings of the 9th USENIX conference on File and stroage technologies 2011, (FAST 2011)

[21] Xing, Jing and Xiong, Jin and Sun, Ninghui and Ma, Jie, *Adaptive and scalable metadata management to support a trillion files*, Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, (SC 2009)

[22] Nisar, Arifa and Liao, Wei-keng and Choudhary, Alok, *Scaling parallel I/O performance through I/O delegate and caching system*, Proceedings of the 2008 ACM/IEEE conference on Supercomputing, (SC 2008)

[23] FUSE, Filesystem in Userspace, http://fuse.sf.net/.

[24] Lachaize, Renaud and Flouris, Michail, *Parallel Postmark Benchmark: cluster postmark*, http://michail.flouris.net/2008/06/parallel-postmark-benchmark/.

[25] McKusick, Marshall Kirk and Quinlan, Sean, *GFS: Evolution on Fast-forward*, ACM Queue, vol 7, 10-20, (2009)

[26] Stoica, Ion and Morris, Robert and Karger, David and Kaashoek, M. Frans and Balakrishnan, Hari, *Chord: A scalable peer-to-peer lookup service for internet applications*, Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, (SIGCOMM 2001)

[27] Andrew Leung, Minglong Shao, Timothy Bisson, Shankar Pasupathy, Ethan L. Miller, *Spyglass: Fast, Scalable Metadata Search for Large-Scale Storage Systems*, Proceedings of the 7th USENIX Conference on File and Storage Technologies, (FAST 2009)

[28] Yu Hua, Hong Jiang, Yifeng Zhu, Dan Feng and Lei Tian. *SmartStore: A New Metadata Organization Paradigm with Semantic-Awareness for Next-Generation File Systems*, Proceedings of the International Conference for High Performance Computing, Network, Storage and Analysis, ACM/IEEE Supercomputing Conference, (SC 2009)

[29] P. Schwan, *Lustre: building a file system for 1000-node clusters*, in Proceedings of the Linux Symposium, (2003).

[30] Ligon, III, W.B., and Ross, R. B., *PVFS: Parallel Virtual File System*, Beowulf Cluster Computing with Linux, Thomas Sterling, editor, pages 391-430, MIT Press, November, (2001).

[31] Michael Kuhn, Julian Kunkel, Thomas Ludwig, *Directory-Based Metadata Optimizations for Small Files in PVFS*, Proceedings of the 14th international Euro-Par conference on Parallel Processing, (EuroPar 2008)

[32] Philip Ross Carns, Sam Lang, Robert Ross Kunkel, Thomas Ludwig, *Small File Accesses in Parallel File Systems*, Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing, (IPDPS 2009)

[33] Oracle, BerkeleyDB, (2011)

[34] Markus Oberhumer, LZO Compression, www.oberhumer.com/opensource/lzo, (2011)

[35] Nick Murphy and Mark Tonkelowitz and Mike Vernal, *The Design and Implementation of the Database File System*, (2002)