2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

Scalable Huge Directories through OSD+ Devices

Ana Avilés-González, Juan Piernas, Pilar González-Férez
Dept. de Ingeniería y Tecnología de Computadores
Universidad de Murcia
Murcia, Spain
Email: {ana.aviles, piernas, pilar}@ditec.um.es

***Abstract*—Management of directories with millions of files, accessed by thousands of clients at the same time, is a problem recently identified in HPC environments. This paper introduces an OSD+-based technique to deal with those directories. We use directory objects in OSD+ devices for dynamically distributing a huge directory among several servers. Directory objects work independently, achieving good performance and scalability. Experiments show that, by using just 8 OSD+s and Ext4, FPFS is able to create, stat and delete more than 70,000, 120,000 and 37,000 files per second, respectively. With ReiserFS, these numbers are 118,000, 97,000 and 67,000. Experiments, however, have produced unforeseen results too. While distribution is beneficial when a huge directory is accessed by many clients, it can also downgrade the performance when several huge directories are concurrently accessed by a few clients.**

***Keywords*-Distributed huge directories; metadata cluster; OSD+; FPFS.**

I. Introduction

The average size of files in many storage systems, that store petabytes of data, is decreasing, and this, in turn, increases the number of files that a modern distributed storage system has to deal with [1], [2], [3]. The increase in the number of files strains the metadata services provided by storage systems because metadata operations are CPU consuming and sensitive to storage devices' performances.

A growing number of files is not the only problem file systems need to face. Another issue is the increasing use of huge directories with millions or billions of entries accessed by thousands of clients at the same time [2], [4], [5], [1]. This scenario appears, for instance, for data-intensive parallel applications that create a file per thread/process, instead of creating a single large file for all the threads, because design and implementation is easier [6], [7].

To deal with a large number of files, some parallel file systems use a small cluster of metadata servers [8], [9], [10]. Other parallel file systems expect to provide a similar service shortly [11], [12]. Only a few provide (or plan to provide) some support for huge directories [10], [11], [13].

The Fusion Parallel File System (FPFS) builds such a metadata cluster by means of *OSD+* devices [14]. OSD+s support *directory objects*. Unlike objects found in a traditional OSD (referred here as *data objects*), directory objects store file names and attributes, and support metadata-related operations. Through OSD+s, an FPFS metadata cluster is as large as its corresponding data cluster. OSD+s have already proved they can achieve a high performance and scalability for metadata operations [14].

In this paper, we show how to integrate the management of huge directories with OSD+s. Our approach leverages the existing directory objects in OSD+s. We propose to dynamically spread a huge directory across several directory objects in different OSD+s, where the original directory object will act as the *primary directory object* while the rest as *secondary directory objects*. We also describe the implications that this technique has for other FPFS elements.

We have evaluated our metadata cluster's performance and scalability for huge directories. Experiments show that, using Ext4 or Reiserfs as backend files systems, dozens and even hundreds of thousands of operations per second are performed with just a few OSD+s. Also, scalability achieved is very good, being superlinear in some cases. These results show that FPFS can fulfill the tough requirements of many HPC environments for huge directories such as a billion files per directory, and more than 40,000 files created per second [6].

Experiments, however, have also produced unexpected results. When a huge directory is accessed by hundreds or thousands of clients at the same time, distribution is beneficial. But, when several huge directories are concurrently accessed by a few clients, distribution can downgrade the performance of the metadata service. Moreover, the dynamic redistribution of several huge directories at the same time can also be expensive if it is not properly implemented. This paper analyzes these problems along with some possible solutions.

II. Related Work

GPFS [15] distributes hugedirs through extendible hashing. However, working at a disk-block level basis, and using several locking mechanisms, limit the performance achieved by GPFS for some directory-related operations, including those on large directories [4]. Boxwood [16] also supports hugedirs by means of B-link trees, each one distributed among several servers. Nevertheless, since Boxwood relies on a global lock service for synchronized metadata accesses, it lacks the ability to effectively deal with a concurrently-accessed directory.

1066-6192/12 $26.00 © 2012 IEEE
DOI 10.1109/PDP.2013.11

Patil and Gibson [2] introduce GIGA+, a POSIX-compliant scalable directory design. GIGA+ incrementally hashes a directory into a growing number of partitions, that are migrated among metadata servers for load balancing. Migrations are individually done by the servers, without a system-wide serialization, synchronization or notification.

By merging ideas from extendible hashing and GIGA+, Yang *et al.* [13] present an implementation of distributed directories for OrangeFS. When a directory is created, an array of *dirdata objects* is allocated, with each object on one metadata server. Directory entries are then spread across the different dirdata objects. Unlike GIGA+, the initial number of active dirdata objects is configurable. They claim that the splitting process is expensive and, for a directory that is expected to be large, it is better to use all the dirdata objects to get better scalability from the start. A similar approach has been proposed for Lustre [11], where directories are statically striped over several MDTs.

Ceph [10] writes a directory's content to the object storage devices using the same striping and distribution strategy as file data, although metadata operations are carried out by a small cluster of metadata servers. Each metadata server keeps a record of the popularity of metadata within a directory, and adaptively hashes individual directories when they get too big or experience too many accesses.

Finally, Shvachko [17] proposes the use of HBase for maintaining the HadoopFS namespace, making HBase a scalable replacement for the HadoopFS's NameNode. To partition the namespace, Shvachko analyzes several existing approaches such as hashing of file paths, Ceph-like partitioning, and fixed-height tiles. In theory, some of these approaches (e.g., hashing of file paths) can also be used for splitting a big directory into several servers.

Our proposal for managing hugedirs in FPFS is similar to that of Lustre and OrangeFS, although, unlike them, it is dynamic since new directories are not initially distributed, and only directories which grow too large get distributed. Our focus, however, is not on proposing new mechanisms for hugedirs but on showing that distributed directories can be efficiently implemented in an OSD+-based metadata cluster. Thanks to the use of OSD+s devices, FPFS can provide better performance for hugedirs accessed by thousands of clients than other parallel file systems.

III. Architecture of FPFS

Generally, current parallel file systems have three main components: clients, metadata servers and data servers. Data servers are usually OSD [18] devices that export an object interface, and manage the disk data layout.

FPFS [14], however, merges data and metadata servers into a single type of server (see Figure 1) by using a new enhanced OSD that we call OSD+. OSD+s are capable not only of managing data as a common OSD does, but also of handling metadata. By using OSD+s, we increase the metadata cluster capacity (becoming as large as the data cluster), performance and scalability. The new devices simplify the storage system since no difference between two kind of servers is made.

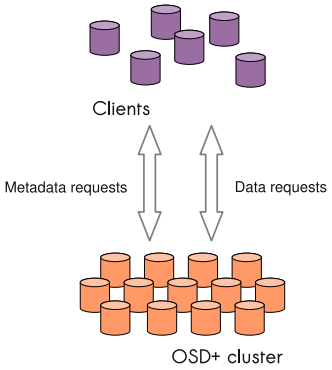

Figure 1. FPFS architecture.

A. OSD+

Besides the low-level block allocation functions, traditional OSDs can take advantage of their device intelligence by implementing more complex tasks [10], [19]. OSD+s take this approach a step further by also delegating metadata management to storage devices.

Traditional OSDs are only able to deal with *data objects*, that support operations like creating and removing objects, and reading from and writting to a specific position in an object. Our design, however, extends this interface to define *directory objects* capable of managing directories. They support metadata operations like creating and removing directories and files, or getting directory entries.

Currently, there exist no commodity OSD-based disks, so mainstream computers, exporting an OSD-based interface, are deployed. Internally, a local file system is used for storing objects. We take advantage of this fact by *directly mapping* directory operations in FPFS to directory operations in the local file system. Thus, we export many features of the local to the parallel file system, such as concurrency and atomicity, when a metadata operation involves one directory (one OSD+). When it involves more than one OSD+ (e.g, a rename), the participating OSD+s deal with concurrency and atomicity by themselves, without client involvement.

Note that OSD+s are substantially different from the modified OSD devices proposed by Ali *et al.* [20]. These authors explore the use of OSD devices for storing and partially managing directories. Their proposal saves directory entries as attributes of empty objects, and introduces a new OSD operation to make attribute changes atomic. However, despite this operation, OSDs are basically passive with respect to metadata operations, which are performed by a small cluster of dedicated metadata servers. Unlike this approach, in FPFS, all the OSD+s actively participate in the

storage and management of a complete directory hierarchy by means of the directory objects.

B. Metadata Distribution

FPFS distributes the directory objects (and, therefore, the file-system namespace) across the metadata cluster to make metadata operations scalable with the number of OSD+s, and to provide a high performance metadata service. In the current implementation, distribution is based on CRUSH [10], a deterministic pseudo-random function that guarantees a probabilistically balanced distribution.

Hash partition strategies present different scalability problems during cluster resizing, renames and permission changes. In FPFS, cluster resizing problems are addressed by CRUSH, which minimizes metadata migrations and imbalances due to the addition and removal of devices. Renames and permission changes are managed in FPFS by means of lazy techniques [21]. Note that, in our case, renames and permission changes only affect directories.

C. Directory Objects

Internally, a directory object is implemented as a regular directory whose pathname is its directory pathname in FPFS. Therefore, the directory hierarchy of the parallel file system is imported within the OSD+s by partially replicating the global namespace. Thus, the directory hierarchy is maintained, allowing FPFS to provide standard directory semantics, and to obtain file and directory access permissions.

To preserve the hierarchy, directory objects maintain an entry for every file and subdirectory they have by creating empty files and directories in the local file system.

D. Clients

In order to obtain the layout of a file, an FPFS client first contacts the OSD+ that contains the object of the parent directory of the target file. Once the file layout information is found, the FPFS client communicates with the OSD+s storing the data objects, similarly to other parallel file systems.

IV. Huge Directories

So far, we have assumed that a directory object is managed by a single OSD+. This is probably the most efficient approach for small directories; striping across multiple servers would lead to an inefficient resource utilization, particularly for directory scans that would incur disk-seek latencies on all servers only to read tiny portions [2].

However, huge directories (or *hugedirs* for short) are also common for some HPC applications, and new mechanisms are necessary to deal with them, specially when thousands of clients work concurrently on the same hugedir.

A. Design

FPFS dynamically distributes hugedirs among several OSD+s, considering a directory is huge when it stores more than a given number of files. Once the threshold is exceeded, the directory is distributed along a subset of OSD+s. In doing so, the directory's workload is shared out among several OSD+s, avoiding an OSD+ to become a bottleneck.

There are a *primary* and several *secondary* OSD+s supporting a hugedir. The former has the *primary directory object*, which is the object that a client would usually contact if the directory was not distributed (i.e., directory objects for small directories are also their primary directory objects). The latter store *secondary directory objects*, which are contacted by clients aware of the directory's distribution.

A client unaware of the distribution of a hugedir contacts its primary object as any other regular directory object, obtaining the OSD+ *id* (*oid*) through the following function:

$$oid = CRUSH(hash(dirname))\%\ osd\_count. \quad (1)$$

As reply, the client gets a *distribution list* containing the primary and secondary OSD+ *ids*. The client stores this list in memory, and uses it for calculating the location of the hugedir's entries. Once aware, the client changes from the directory-level to a "local" file-level distribution, used for allocating new entries, and looking up existing ones, in the hugedir:

$$oid = osd\_set[(hash(filename)\%osd\_set\_count)]. \quad (2)$$

osd_set is the *distribution list*; as index, the value returned by a hash function applied on a file name is used.

Once distributed, new files in the hugedir are created following Equation 2. Note, however, that files created before the directory was identified as huge, should be redistributed to follow Equation 2 too. Due to this redistribution, a directory can be found in three different states: "non-distributed", "distributing", and "distributed". The first is the most common state, where all directories storing less entries than the threshold belong to. The second lasts as long as the redistribution of existing entries does. During this state no requests can be performed on the hugedir, and clients are informed of the new distributed state. Finally, the "distributed" state is set once redistribution finishes.

B. Huge directory renames

In the aforementioned Equation 2, we get rid of the directory's name and include the distribution list. With this approach, we avoid migrating all the directory objects of a hugedir on a rename and, usually, only the location of its primary object will change.

Two rename scenarios arise. When the OSD+ storing the new primary object is already in the distribution list, no migration is done, only a change of roles between two

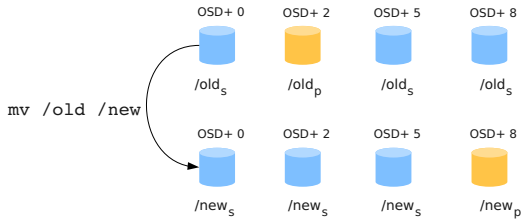

Figure 2. Hugedir rename where the new primary OSD+ is already in the distribution list.

Figure 3. Hugedir rename where the new primary OSD+ was not in the distribution list.

OSD+s takes place (see Figure 2). However, when the OSD+ storing the new primary object is different from those in the distribution list (see Figure 3), apart from changing the primary object location, a migration is also needed. All entries in the previous primary object are moved to the new primary object, just like a small directory rename works [14].

C. Clients

OSD+s decide when a directory is distributed, but clients are not aware of that distribution until they perform a regular operation on the directory. At that moment, OSD+s reply informing it is distributed and attach its distribution list. Then, clients cache this list and apply the corresponding function (see Equation 2) to find the directory's entries.

Eventually, this cached information may be out of date due to a rename or deletion of a hugedir. Hence, some inconsistent scenarios can happen given that the clients' side is only updated when they send a request to a directory.

To keep coherency, each directory object stores a *timestamp* with its creation time, and includes it in any message sent by the object. In turn, clients cache that information to later check whether a hugedir has changed.

Two coherency problems can happen. Firstly, after a hugedir rename, a client sends a request to an old secondary OSD+. Secondly, a hugedir is removed and created again, and the client sends a request to an OSD+ storing the new directory object. In the former case, the OSD+ does not store that hugedir anymore, so replies that the directory does not exist. In the later, the OSD+ replies affirmatively, but the client will note timestamps differ. In both cases, the client cleans up the cached information for the directory, and retries the operation following Equation 1. After retrying, the client receives a new distribution list if the directory is huge.

D. Implementation

In the current implementation, FPFS distributes a directory when its size is bigger than a given size, because `stat` does not return the number of files in a directory. The distribution is dynamic: initially, a directory is managed by a single OSD+ which stores the directory object. As soon as the directory is identified as huge, a distribution process reallocates directory entries. This process is led by the primary OSD+, which sends the reallocated entries to the secondary OSD+s in parallel, following Equation 2.

To assure clients do not get an inconsistent view of the directory, every OSD+ records its directories activity through an AVL tree. If a directory is non-distributed, clients accessing it are monitored until the state changes into distributing. Then, the tree is used for blocking any new request and for controlling when outstanding requests on the directory finish. Once the latter requests are done, distribution takes place. All requests received during the process are returned to the clients together with the distribution list, so they can forward those requests to the appropriate OSD+s.

Relocated entries are not erased from the primary objects during distribution due to the remove operation's poor performance on local file systems (see Section V). However, this may result in inconsistent scenarios, for instance, during hugedirs scans, where each OSD+ returns the entries contained in its primary or secondary object (depending on its role). Not having those entries removed from the primary object can cause incoherent results (e.g., duplicated entries). To avoid that, the primary object checks whether it is responsible for an entry before sending it to a client. Checking is performed on-the-fly by using Equation 2. This is far quicker than deleting files or marking them as "deleted" with extended attributes. Thanks to this approach, we have sped up the distribution process by one order of magnitude.

Hugedir's directory objects are internally marked as either primary or secondary through extended attributes, as well as directory states and distribution lists. To get a faster access, a directory's state is also recorded in the AVL tree the first time is accessed.

V. Experimental Results

This section analyzes performance and scalability achieved by FPFS for hugedirs. We have built a skeleton file system for FPFS where both OSD+s and clients are completely implemented in user-space [14]. Since we are focused on metadata, the version used only supports directories and some related operations. Metadata logs are committed to disk every 5 seconds, as in other file systems [22].

The tested system is a cluster made up of 12 compute and 1 frontend nodes. Each compute node has two 2.50 GHz Intel Xeon E5420 Quad-core CPUs, 4 GB of RAM, and two Seagate ST3250310NS disks of 250 GB. On each node, the system disk has a 64-bit Fedora Core 11, and the test disk is

exported as an FPFS OSD+. The interconnect is a Gigabit network with a D-Link DGS-1248T switch.

For clients, 1 to 4 nodes are used depending on the test. For FPFS, configurations varying the number of OSD+s (1, 2, 4 and 8), and backend (Ext4 or ReiserFS) are set up.

Metadata performance depends on the options used for formatting and mounting a file system. Specifically, an Ext4 file system is formatted with options `-J size=400 -i 4096 -I 512 -O dir_index, extents, uninit_groups`, and a ReiserFS one with `--journal-size 32749`. Mount options are, for Ext4, `user_xattr`, `noatime`, `nodiratime` and `data=writeback`, while, for ReiserFS, `user_xattr`, `notail`, `noatime` and `nodiratime`.

We have analyzed two different aspects of the FPFS support for hugedirs: (a) throughput and scalability for a single shared directory, and (b) performance when several shared and non-shared hugedirs are accessed at the same time.

Three benchmarks (*create*, *stat* and *unlink*) have been used for creating, stating, and deleting empty files[1]. In all the benchmarks, operations on files in a hugedir are evenly distributed among the clients accessing the directory.

Results are the average of, at least, five runs of each test. Confidence intervals are shown as error bars, for a 95% confidence level. Test disks are formatted between runs for *create*, and unmounted/remounted between tests for the rest.

A. Baseline Performance

We start with a baseline for the performance of various file systems with the *create* benchmark. We compare results obtained by running this test locally on Ext4 and ReiserFS to those obtained by running the same test on a separate single client and single server instance of FPFS, OrangeFS [13], HadoopFS [17], and NFSv3 (we also tried Ceph [10], but several bugs prevented us from succeeding). Except in FPFS, that uses Ext4 and ReiserFS, the other cases use Ext4 as backend.

For FPFS, the create benchmark is linked with the FPFS library that implements POSIX-equivalent file operations. A similar approach is used for OrangeFS and HadoopFS. Local file systems, and the NFSv3 client, perform file operations through system calls.

Table I shows the baseline performance. As we can see, local file systems deliver high directory insert rates, with ReiserFS surpassing Ext4 by 24%. Any networked file system achieves a modest performance, being FPFS the file system which obtains the best results among them.

A reason of the low performance achieved by networked file systems, compared with that reached by local ones, is network overhead, specially for FPFS. For instance, FPFS takes 109.64s to create 400,000 files on Ext4 (this time yields 3,648 creates per second, as shown in Table I). Exchanging 1,600,000 messages of 64 bytes each between two cluster nodes (this is roughly the same network traffic produced by FPFS in this test) takes 80.66s. Considering that Ext4 needs 22.19s to locally create 400,000 files, we can conclude that FPFS's overhead is small ($109.64 - 80.66 - 22.19 = 6.79$s), and its limiting factor is the interconnect.

[1]Note that similar tests can be performed by means of well-known benchmarks such as metarates and mdtest.

Table I
FILE CREATE RATE IN A SINGLE DIRECTORY ON A SINGLE SERVER AND 400000 FILES IN TOTAL.

	File system	File creates (ops/s) in one directory
FPFS (library API)	Ext4	3,648
	ReiserFS	3,859
Local file systems	Ext4	18,023
	ReiserFS	22,324
Networked file systems	NFSv3 filer	1,479
	HadoopFS	262
	OrangeFS	253

B. Single shared directory

This section shows results obtained when a single shared directory is accessed by hundreds of processes at the same time to create, get the status of and delete files. There are 256 clients, spread across 4 cluster nodes, that work on equally-sized disjoint subsets of the files. In total, the shared directory contains $200,000 \times N$, $400,000 \times N$, or $800,000 \times N$ files, where N is the number of OSD+s. Since files are uniformly distributed among the OSD+s, directory objects, containing the files of the hugedir, are of equal size.

Figure 4 depicts performance obtained by FPFS, in transactions per second, when a hugedir is distributed. Results achieved by Ext4 and ReiserFS are compared. While ReiserFS gets better results during create and unlink operations, Ext4 is better during stats.

As we can see, directory size determines the performance obtained by FPFS with Ext4 to a large extent, achieving better results for smaller directory objects. Note that clusters of hundreds or thousands of OSD+s are expected, so a hugedir distributed among many OSD+s will typically use small directory objects. Performance hardly depends on the directory objects' size with ReiserFS as backend system.

These good results have been obtained with a dynamic distribution. The time taken by this distribution is usually small (< 2 s). A directory is identified as huge when its size is greater than 244 kB. Given the lengths of the file names, this is equivalent to distribute a directory when it has more than 8,000 files. This threshold is based on the observation that 99.99% of directories contain less than 8,000 files [2].

Note that the threshold's size affects only the create benchmark, because the distribution of a directory only takes place when its size increases. Neither the stat nor the unlink

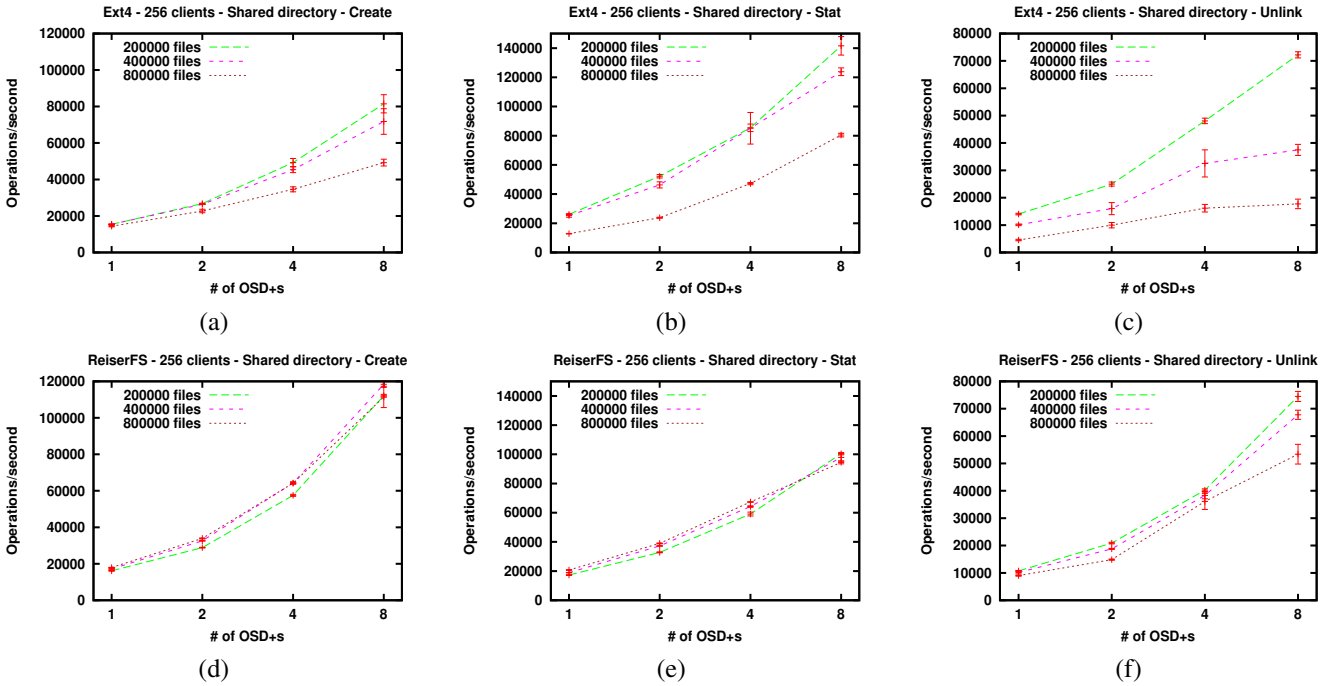

Figure 4. Operations per second obtained by FPFS with Ext4 (three first graphs) and ReiserFS (three second graphs) for the create, stat and unlink benchmarks, when using one shared directory.

benchmarks are affected by the threshold, since the directory is already distributed when the test is run.

Impact of directory size on performance also determines the scalability reached by Ext4 and ReiserFS (Figure 5 shows results for Ext4; results for ReiserFS have been omitted due to space restrictions). Speedup is computed by comparing the performance obtained when not distributing and when distributing the files in a single shared directory. As the number of OSD+s increases, we get outstanding results, specially for Ext4, achieving a superlinear scalability in all the tests. Note that scalability cannot be calculated from Figure 4 since the number of files varies with the number of OSD+s, and we have to compare directories of equal size.

Superlinear scalability when using Ext4 as backend is mainly due to the fact that Ext4 performance gets worse as the number of entries in a directory grows, as already explained. Hence, by distributing the management of a directory, we are not only sharing out the workload among various servers, but also creating smaller secondary directory objects on the local file systems. With ReiserFS, performance also decreases with the directory size, but the downgrade is much softer. Only when a directory is really huge (millions of files), the downgrade is noticeable and ReiserFS can achieve a superlinear scalability too. Some experiments carried out (also omitted due to space constraints) have showed that performances of Ext4 and ReiserFS vary over the course of the time for the create test. But, while Ext4 performance decreases from roughly 95,000 ops/s at the beginning to 45,000 ops/s at the end of the *create* test (with 800,000 files per directory object and 8 OSD+s), ReiserFS gets a quite sustained rate of ops/s, and its performance only decreases from roughly 145,000 ops/s to 115,000 ops/s.

C. Multiple huge directories

As previous section shows, distribution is beneficial for a hugedir accessed by hundreds or thousands of clients at the same time. However, results can be rather different when several hugedirs are concurrently accessed by a few clients.

Table II shows performance obtained by FPFS on Ext4 when the number of clients per hugedir varies. There are 8 directories, each containing 320,000 files (2,560,000 files in total). Directories are distributed: never, dynamically (when a directory roughly exceeds 8,000 files), and always (i.e., when threshold is 0). Note that 1 client per directory is an example for non-shared directories, and that, with 32 clients per directory, there are 256 clients altogether.

Given a benchmark and number of processes per directory, since workload is always balanced, we should expect roughly the same throughput for the three distribution configurations at least. However, results say that distribution downgrades performance for non-shared hugedirs (1 client per directory). When there are 16 or 32 clients per directory, distribution slightly improves performance in some cases, but downgrades it in other cases, for instance, in *create*.

The main reason is that distribution makes disk accesses

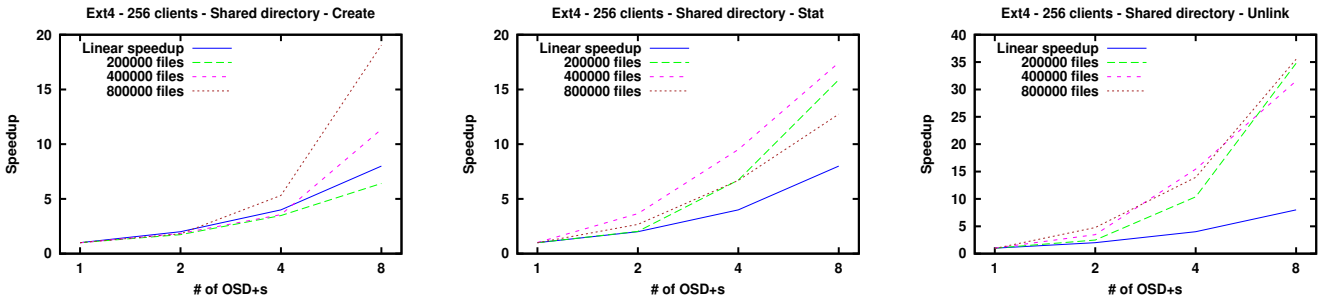

Figure 5. Scalability obtained by FPFS with Ext4 and a single directory: (a) create; (b) stat; (c) unlink.

Table II
PERFORMANCE OBTAINED BY FPFS ON EXT4 WHEN THE NUMBER OF CLIENTS PER HUGEDIR VARIES. THE 8 EXISTING DIRECTORIES ARE DISTRIBUTED: NEVER, DYNAMICALLY (WHEN A DIRECTORY ROUGHLY EXCEEDS 8,000 FILES), AND ALWAYS (I.E., WHEN THRESHOLD IS 0). BENCHMARK NAMES APPEAR BOLD-FACED. CONFIDENCE INTERVALS (NOT SHOWED) ARE SMALLER THAN 10% OF THE MEAN.

create	1 client/directory			16 clients/directory			32 clients/directory		
# OSD+	Never	Dynamic	Always	Never	Dynamic	Always	Never	Dynamic	Always
1	215.27	217.66	212.75	152.48	152.21	152.20	154.98	154.14	154.28
2	150.67	198.90	197.40	60.48	86.03	86.55	63.06	84.23	83.44
4	106.87	164.34	160.30	33.53	52.49	45.61	34.27	49.74	43.06
8	92.75	140.15	137.74	21.29	38.05	36.09	21.59	31.70	30.34

stat	1 client/directory			16 clients/directory			32 clients/directory		
# OSD+	Never	Dynamic	Always	Never	Dynamic	Always	Never	Dynamic	Always
1	243.18	240.60	243.37	203.00	203.79	207.18	207.79	206.79	206.75
2	126.56	136.51	134.37	77.01	80.36	77.17	77.62	79.61	76.90
4	76.60	85.14	83.29	24.06	26.76	25.38	23.73	26.97	24.97
8	68.98	77.87	77.09	12.63	14.68	14.09	11.91	15.26	14.64

unlink	1 client/directory			16 clients/directory			32 clients/directory		
# OSD+	Never	Dynamic	Always	Never	Dynamic	Always	Never	Dynamic	Always
1	512.57	511.49	507.72	1099.27	1217.70	1118.20	1522.21	1366.19	1438.73
2	163.31	230.08	226.24	258.33	398.13	381.33	280.84	455.29	448.77
4	85.97	143.52	137.06	80.67	164.19	103.50	95.90	202.02	147.50
8	63.67	114.10	111.41	21.49	79.17	77.79	22.02	81.55	51.02

less efficient because there are always 8 directory objects per server that are accessed almost at the same time. Since directory objects are spread over the disk, this increases head seeks and trashes disk caches. It also limits the scalability achieved. Without distribution, however, directories and requests are evenly distributed across the servers, making the number of directory objects per server smaller ($8/N$, where N is the number of OSD+s). Less directory objects means less head seeks and, hence, better performance.

We have carried out the same tests using ReiserFS as backend. Results obtained (not shown here due to space restrictions) are quite similar, although there exist more cases where distribution improves performance over non-distributed directories. The reason is again related to the head-seek overhead. Since ReiserFS does not divide a disk into block groups, it puts directory objects close on disk. This produces shorter seeks, and reduces seek times.

The above results question directory distributions purely based on directory sizes. More important than the size are the number of processes accessing a directory, and the availability of resources in the servers. A problem is that both things can vary quickly, and continuously changing the servers a directory is split into seems inefficient. However, distribution is important for hugedirs, as we have also seen. Therefore, we need to find better ways to split directories.

Since a problem seems to be the relative poor performance of the hard disks for random accesses, the use of SSD devices for metadata could be a solution. These devices does not suffer the head-seek overhead problem, so we assume that an increase in the number of directory objects and requests per OSD+, caused by the distribution of hugedirs, should not downgrade servers' throughput.

VI. Conclusions and Future Work

An OSD+-based technique for FPFS to deal with huge directories accessed by thousands of clients at the same time has been presented. This proposal uses the directory objects provided by the OSD+ devices to dynamically distribute a huge directory among several servers. Directory objects work independently, achieving good performance and scalability. Moreover, we have optimized the redistribution of existing directory entries, and avoided massive metadata migrations when a huge directory is renamed.

Results show that FPFS exceeds today's requirements of HPC applications regarding huge directories (a billion files per directory, more than 40,000 files created per second, etc). By using just 8 OSD+s and ReiserFS, FPFS is able to create, stat and delete more than 118,000, 97,000 and 67,000 files per second, respectively. Scalability is also very good, being superlinear in some cases.

Experiments, however, have produced unexpected results too. While distribution is beneficial when a huge directory is accessed by many clients, it can also downgrade the performance when several huge directories are concurrently accessed by a few clients. A limiting factor here is the small number of IOPS supported by current hard drives. As future work, we plan to experiment with SSD devices to confirm our hypothesis.

Acknowledgment

Work supported by Spanish MICINN, and European Comission FEDER funds, under grants TIN2009-14475-C04 and TIN2012-38341-C04-03.

References

[1] R. Freitas, J. Slember, W. Sawdon, and L. Chiu, "GPFS scans 10 billion files in 43 minutes," IBM Almaden Research Center, Tech. Rep. RJ10484, 2011. [Online]. Available: http://www.almaden.ibm.com/storagesystems/resources/GPFS-Violin-white-paper.pdf

[2] S. Patil and G. Gibson, "Scale and concurrency of GIGA+: File system directories with millions of files," in *Proceedings of USENIX FAST'11*, 2011.

[3] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. L. Miller, D. D. E. Long, and T. T. McLarty, "File system workload analysis for large scale scientific computing applications," in *Proceedings of 21st MSST*, 2004.

[4] E. Artiaga and T. Cortes, "Using filesystem virtualization to avoid metadata bottlenecks," in *Proceedings of DATE Conference & Exhibition*, 2010.

[5] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: A checkpoint filesystem for parallel applications," in *Proceedings of SC'09*, 2009.

[6] H. Newman, "HPCS mission partner file I/O scenarios, revision 3," Nov. 2008. [Online]. Available: http://wiki.lustre.org/images/5/5a/Newman_May_Lustre_Workshop.pdf

[7] A. Fikes, "Storage architecture and challenges," in *Google Faculty Summit 2010*, June 2010. [Online]. Available: http://research.google.com/university/relations/facultysummit2010/storage_architecture_and_challenges.pdf

[8] R. Latham, N. Miller, R. Ross, and P. Carns., "A next-generation parallel file system for linux clusters," *LinuxWorld*, pp. 56–59, Jan. 2004.

[9] S. Sinnamohideen, R. R. Sambasivan, J. Hendricks, L. Liu, and G. R. Ganger, "A transparently-scalable metadata service for the Ursa Minor storage system," in *Proceedings of the USENIX ATC*, 2010.

[10] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn., "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th OSDI*, 2006.

[11] W. Di, "CMD code walk through," http://wiki.lustre.org/images/7/70/SC09-CMD-Code.pdf, 2009. [Online]. Available: http://wiki.lustre.org/images/7/70/SC09-CMD-Code.pdf

[12] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proceedings of the 26th MSST*, 2010.

[13] S. Yang, W. B. L. III, and E. C. Quarles, "Scalable distributed directory implementation on Orange File System," in *Proceedings of 7th SNAPI Workshop*, 2011.

[14] A. Avilés-González, J. Piernas-Cánovas, and P. González-Férez, "A metadata cluster based on OSD+ devices," 2011.

[15] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proceedings of USENIX FAST'02*, 2002.

[16] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou, "Boxwood: Abstractions as the foundation for storage infrastructure," in *Proceedings of the 6th OSDI*, 2004.

[17] K. V. Shvachko, "Apache Hadoop. The scalability update," *USENIX ;login Magazine*, vol. 36, pp. 7–13, Jun. 2011.

[18] M. Mesnier, G. R. Ganger, and E. Riedel, "Object-based storage," *IEEE Communications Magazine*, Aug. 2003.

[19] E. Polyakov., "The Elliptics network," http://www.ioremap.net/projects/elliptics, 2009. [Online]. Available: http://www.ioremap.net/projects/elliptics

[20] N. Ali, A. Devulapalli, D. Dalessandro, P. Wyckoff, and P. Sadayappan., "An OSD-based approach to managing directory operations in parallel file systems," in *Proceedings of SC Conference*, 2008.

[21] S. A. Brandt, E. L. Miller, D. D. E. Long, and L. Xue., "Efficient metadata management in large distributed storage systems," in *Proceedings of the 20th MSST*, 2003.

[22] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: current status and future plans," in *Linux Symposium*, 2007. [Online]. Available: http://ols.108.redhat.com/2007/Reprints/mathur-Reprint.pdf