International Journal of Parallel Programming manuscript No.
(will be inserted by the editor)

Scalable Metadata Management through OSD+ Devices

Ana Avilés-González · Juan Piernas · Pilar González-Férez

Received: date / Accepted: date

Abstract We present the design and implementation of both an enhanced new type of OSD device, the *OSD+ device*, and a metadata cluster based on it. The new OSD+ devices support *data objects* and *directory objects*. Unlike "data" objects, present in a traditional OSD, directory objects store file names and attributes, and support metadata–related operations. By using OSD+ devices, we show how the metadata cluster of the Fusion Parallel File System (FPFS) can effectively be managed by all the servers in a system, improving the performance, scalability and availability of the metadata service. We also describe how a directory with millions of files, and accessed by thousands of clients at the same time, is efficiently distributed across several servers to provide high IOPS rates. The performance of our metadata cluster based on OSD+s has been evaluated and compared with that achieved by Lustre. The results show that our proposal obtains a better throughput than Lustre when both use a single metadata server, easily getting improvements of more than 60–80%, and that the performance scales with the number of OSD+s. They also show that FPFS is able to provide a sustained throughput of more than 70,000 creates per second, and more than 120,000 stats per second, for huge directories on a cluster with just 8 OSD+s.

Keywords Metadata cluster · OSD+ · Management of huge directories · Fusion Parallel File System · Metadata service scalability.

CR Subject Classification D.4.2 [**Operating Systems**]: Storage Management – *Secondary Storage* · D.4.3 [**Operating Systems**]: File Systems Management – *Distributed File Systems*.

This work was supported by the Spanish MEC and MICINN, as well as European Commission FEDER funds, under Grants CSD2006–00046 and TIN2009–14475–C04.

Ana Avilés-González, Juan Piernas, Pilar González-Férez
Facultad de Informática, Campus de Espinardo, 30100 Murcia (Spain)
Tel.: +34-868-887657
Fax: +34-868-884151
E-mail: {ana.aviles, piernas, pilar}@ditec.um.es

1 Introduction

The avoidance of bottlenecks is critical in modern distributed storage systems to achieve the desired features of high performance and scalability. Considering that these systems have to deal not only with a large volume of data but also with an increasing number of files, an efficient metadata management becomes a fundamental aspect of a system's storage architecture to prevent such bottlenecks [1,2].

Although metadata is usually less than 10% of the overall storage capacity of a file system, its operations represent 50%–80% of all the requests [3]. Metadata operations are also very CPU consuming, and a single metadata server can easily be overloaded by a few clients in a parallel file system. Hence, to improve the performance and scalability of metadata operations, a cluster of servers is needed. PVFS [4] and Ceph [5], for instance, use a small set of dedicated servers as metadata cluster, and Lustre expects to provide a similar production–ready service for version 2.2 [6].

With respect to data, many modern cluster file systems [4,5,7] use hundreds or thousands of Object Storage Device (OSD) [8], or conceptually equivalent devices, to store information and to achieve a high performance. Because there are no commodity OSD-based disks available yet, these devices are implemented by mainstream computers which export an OSD-based interface, and internally use a regular local file system to store objects.

By taking into account the design and implementation of the current OSD devices, this paper explores the use of such devices as metadata servers to implement the metadata cluster. This way, both data and metadata will be distributed over and managed by thousands of nodes. In order to deal with metadata, we propose to extend the type of objects and operations an OSD supports. Specifically, our new devices, that we call *OSD+*, support *directory objects*. Unlike objects found in a traditional OSD device (referred here as *data objects*), directory objects store file names and attributes, and support metadata–related operations, such as the creation and deletion of regular files and directories.

Although our OSD+s are basically independent and operate in an autonomous manner, they should be able to collaborate to provide a full-fledged metadata service. For instance, there exist metadata operations (e.g., a directory creation) that involve two or more directory objects, which can be managed by different OSD+s.

Since our software-based OSD+ devices also use an internal local file system to store data objects, we propose to take advantage of this fact by *directly mapping* directory–object operations to operations in the underlying file system. This approach produces several benefits: (a) many features (atomicity, POSIX semantics, etc.) and optimizations present in the local file system are directly exported to the parallel file system for metadata operations which involve a single directory; (b) resource utilization in the storage nodes (CPU, memory, secondary storage, etc.) is increased; and (c), since we leverage the directory implementation in the local file system, the software layer which creates the OSD+ interface is thin and simple, producing a small overhead and hopefully improving the overall metadata service performance.

Our OSD+ devices allow us to design a new parallel file system, called *Fusion Parallel File System* (FPFS), which combines data and metadata servers into a single type of server capable of processing all I/O operations. An FPFS metadata cluster will

be as large as its corresponding data cluster, effectively distributing metadata requests among as many nodes as OSD+s comprising the system, and improving the metadata performance and scalability. Metadata availability will also be increased because the temporal failure of a node will only affect a small portion of the directory hierarchy. Note that OSD+s reduce administration costs too, due to the deployment of a single type of server.

Although OSD+s manage any file system operation, they can be seen as members of two separate clusters: a data and a metadata cluster. Since modern file systems already have a good data performance and failure recovery, FPFS's data cluster works as and borrows ideas from them. For the metadata cluster, however, our goal is to provide a better service than that produced by existing file systems. Therefore, this paper only focuses on the design and implementation of the FPFS metadata cluster.

Note that, in order to build a high–performance and scalable metadata service based on OSD+ devices, two main problems must be addressed: (a) distribution of the directory objects among the storage devices to balance workload, and (b) atomicity of operations which involve more than one storage device to ensure file system consistency. Both problems will be addressed in the present work.

Huge directories, with millions of entries, which are accessed and modified at the same time by thousands of clients, are also common for some HPC applications [2, 9, 10]. FPFS also handles this scenario by *dynamically* distributing huge directories among several OSD+s. In doing so, a huge directory's workload is shared out among several OSD+s, avoiding an OSD+ to become a bottleneck.

We have evaluated our metadata cluster and compared its performance with Lustre's [7]. The experimental results show that a single OSD+ can easily improve the throughput of a Lustre metadata server by more than 60–80%. They also show that the performance of our metadata cluster scales with the number of OSD+s, proving that even a small metadata cluster can provide a good performance. They also show FPFS achieves a high performance and super–linear scalability when managing huge directories, making FPFS fulfill the tough requirements of many HPC environments.

To sum up, the main contributions of this paper are: (a) the design and implementation of the OSD+ devices (focused on the support of directory objects), (b) the design and implementation of a metadata cluster using OSD+s, (c) the proposal of a new parallel file system based on OSD+s, (d) the design and implementation of mechanisms to manage huge directories by means of OSD+s, and (e) the evaluation of the performance achieved by the resulting metadata cluster.

The rest of the paper is organized as follows. Section 2 shows the architecture of FPFS, a parallel file system based on OSD+s. Section 3 describes the design of a metadata cluster which uses OSD+s as metadata servers. The OSD+ implementation details are discussed in Section 4. Experimental results are presented in Section 5. The related work is described in Section 6. Finally, Section 7 concludes the paper.

2 Architecture of FPFS: an OSD+–Based Parallel File System

Generally, parallel file systems have three main components: clients, metadata servers and data servers. This architecture, based on two types of servers, improves perfor-

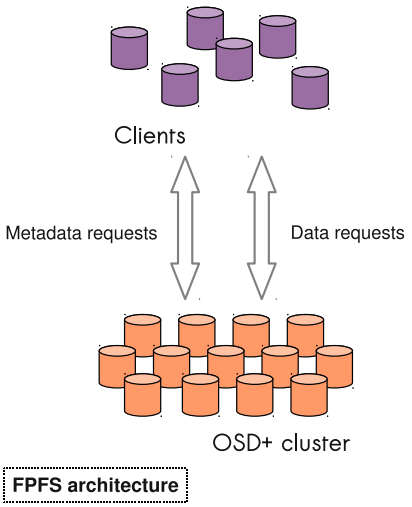

Fig. 1 FPFS architecture.

mance by allowing metadata and data requests to be attended in parallel. Usually, data servers are OSD [8] devices which export an object interface, and intelligently manage the disk data layout (that has traditionally been responsibility of the metadata servers). OSDs also improve data sharing and permit to set object–specific security policies.

Unlike current parallel file systems, OSD+–based Fusion Parallel File System (FPFS) merges data and metadata servers into a single type of server by using a new enhanced OSD device that we call OSD+. The new OSD+ devices are capable not only of managing data as a common OSD does, but also of handling metadata requests. Thus, the metadata cluster increases its capacity becoming as large as the data cluster. System's scalability and capacity are increased too. The new devices simplify the complexity of the system because no difference between two kind of servers is made. Figure 1 shows the two basic components of FPFS (clients and the OSD+ cluster), which are briefly described below.

2.1 Clients

Clients of a parallel file system such as Lustre [7], PVFS [4], PansFS [11] or pNFS [12] behave in the same way when accessing files. First, in order to locate the data objects of a file in the OSD cluster, they contact a known metadata server that provides the location information. Then, they communicate with the given OSDs to access the data objects.

Similarly, in order to obtain the layout of a file, an FPFS client contacts an OSD+ whose *id* is obtained by applying a hash function on the pathname of the parent directory of the target file. Once the file layout information is found, the FPFS client will be able to communicate with the OSD+ storing the corresponding data objects.

2.2 OSD+

Besides the low–level block allocation functions, traditional OSDs can take advantage of their device intelligence by implementing more complex tasks, such as RADOS in Ceph [5], or Elliptics [13] in POHMELFS [14]. RADOS, for example, facilitates serialization, replication, and detection and recovery from failures in a semi–autonomous manner for the OSDs in the data cluster. OSD+s also leverage the intelligence present in the OSD devices, but take the approach a step further, delegating the metadata management to the storage devices as well.

Traditional OSDs are only able of dealing with *data objects*. Typical operations on these objects are: creating and removing objects, and reading from and writing to a specific position in an object. Our design, however, extends this object interface in order to define *directory objects* capable of managing directories. They support metadata–related operations such as creating and removing directories and files, getting directory entries, retrieving file attributes, etc. Besides the usual operations on directories, the OSD+ implementation also provides functions to deal with metadata operations which may involve the collaboration of several OSD+s, such as directory renames, directory permission changes, and links.

It is important to realize that, currently, there are no commodity OSD–based disks available, so mainstream computers exporting an OSD–based interface by running emulators [15], or other software applications, are usually used. Internally, an ordinary local file system stores objects. We do take advantage of this fact by *directly mapping* directory operations in FPFS to directory operations in the local file system. In this way, we export many features of the local to the parallel file system, such as concurrency and atomicity, when a metadata operation involves only one directory (and, hence, only one OSD+). When a metadata operation 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 a client issuing a metadata operation only contacts with the OSD+ containing the object of the parent directory of the target file).

Similarly to Ceph [5], every client and every OSD+ has a copy of a cluster map which describes the underlying physical organization of the cluster, and its current state. As Section 3 explains, this map, along with a companion hash function, is used for distributing data and directory objects across the cluster.

Apart from the OSD+ cluster, there is also a small cluster of monitors (not shown) which stores the master copy of the cluster map, and whose principal task is to keep a consistent and coherent view of the cluster. Each time a client joins the system, it receives the current cluster map from the monitors. Since monitors interact with clients and OSD+s only when changes in the cluster occur, their overhead is small, and can also be run in any of the OSD+s.

3 The Metadata Cluster: Design

The metadata cluster of FPFS uses OSD+ devices to provide a high performance and scalable metadata service. It also profits the enhanced intelligence of the OSD+s

to tackle with directory renames, links and permission changes, in a consistent and atomic manner.

3.1 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. The distribution is based on CRUSH [5], a deterministic pseudo–random function that guarantees a probabilistically balanced distribution of objects through the system. Nevertheless, any other distribution function could be used, such as RUSH [16] (a family of algorithms from which CRUSH is based on), to distribute the objects along the cluster.

Given a directory, CRUSH outputs its *placement group* (PG), a list of devices made up of a primary node and a set of replicas. These devices are chosen according to weights and placement rules that restrict the replica selection across failure domains, avoiding, in this way, potential sources of failures and load imbalance. As input, CRUSH receives an integer which, in our metadata cluster, results from hashing the directory's full pathname. Since CRUSH only needs a *cluster map* to compute the result, and given that this map is available for all the nodes in the cluster, any party in the system is able to independently calculate the location of any directory object.

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 [17]. Nevertheless, it is important to note that, in our case, renames and permission changes only affect directories. The experimental results will show that these operations are infrequent for directories (similar results have recently been obtained by other authors [17, 18]). This fact, along with the mentioned lazy techniques and CRUSH, will further minimize the impact of these operations on the metadata cluster performance.

Although directory objects are scattered across the cluster, the directory hierarchy of the parallel file system is maintained to provide standard directory semantics (e.g., when listing a directory), and to determine file and directory access permissions (note they are determined from the root directory). The directory hierarchy is maintained by storing, in each directory object, an entry for every subdirectory which it contains, if any (see Section 4 for details).

3.2 Huge directories

Although every directory object is managed by a single OSD+, this is probably the most efficient approach for small directories. Studies of large file systems have found that 99.99% of the directories contain less than 8,000 files [2]. Striping small directories 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.

However, huge directories are also common for some HPC applications, and new mechanisms are necessary to deal with them, specially when thousands of clients work on the same huge directory at the same time.

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

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

Clients do not know which directories are distributed in advance. When a client sends a request to a distributed directory, it receives a response indicating that the directory is managed by several objects. Then, the client marks the directory as distributed and sends any subsequent request to the appropriate primary or secondary OSD+. The latter is done by changing the distribution function from a directory–level hash function:

$$osd\_id = CRUSH(hash(parent\_dirname)) \tag{1}$$

to a file–level hash function, so as to allocate the files in different OSD+s (% is the module operation and *osd_count* is the total number of OSD+ in the cluster):

$$osd\_id = (CRUSH(hash(parent\_dirname)) + hash(filename))\%\ osd\_count. \tag{2}$$

Note that, thanks to this function, if a directory is renamed, its primary and secondary directory objects are migrated as a whole (i.e., files do not have to be redistributed among the directory objects).

3.3 Directory Renames

If a directory name changes, so does its location and the location of any directory underneath in the hierarchy. This can incur a massive migration of metadata. To minimize this problem, lazy policies, similar to those used by LH [17], are applied to move the relocated metadata. Unlike LH, file renames do not produce metadata migrations in FPFS because their locations do not depend on their pathnames.

Rename requests are sent to the parent directories of the corresponding target directories. When the rename of a directory occurs, the OSD+ of its parent directory broadcasts the rename to inform the other OSD+s in the cluster, which maintain a *metadata log* for renames and other metadata operations (note that, as an optimization, the rename message could be sent only to those nodes affected by the renamed path). Thanks to the broadcast, when an OSD+ receives an operation on a directory

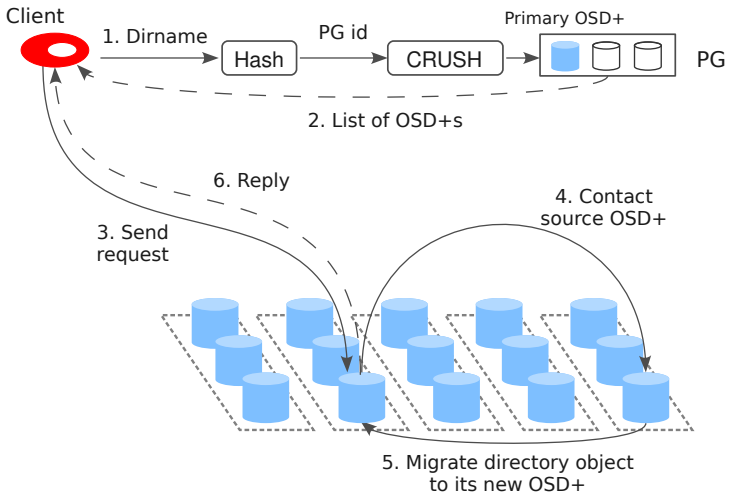

Fig. 2 Directory object migration.

whose object is missing but that, according to the new directory's pathname, should not be, the OSD+ migrates the object to carry out the operation instead of returning an error which an nonexistent object would normally produce.

Figure 2 shows this migration process. First, after obtaining the list of servers which contain the directory object (steps 1 and 2), the client contacts the primary OSD+ (step 3) which is supposed to have the directory object. Then, the failed request forces the OSD+ to migrate the object by looking for the corresponding rename in the log, and contacting the source OSD+ (step 4). Once the migration is done (step 5), the initial operation is carried out and the result is returned to the client (step 6). Due to a previous rename, the source OSD+ may not contain the directory object either. The process is then repeated recursively, moving backwards until the directory object is found and migrated.

3.4 Permission Changes

In order to directly determine access permissions and avoid directory traversals, dual–entry ACLs are used [17]. Given a directory, one entry of its ACL contains its permissions, whereas the other one represents its *path permissions* (these are the intersection of the directory's own permissions and its parent's path permissions). Unlike LH, only directories have dual–entry ACLs in FPFS. A file's permissions are derived from its ACL, and its parent directory's dual–entry ACL.

When checking permissions, the OSD+ containing the target directory object searches in its metadata log for invalidations along the requested object's path. If they exist, the parent directory is accessed (applying the placement function over the parent's path) to get its dual–entry ACL. Once path permissions of the target directory are updated, the requested ACL is calculated and returned. Since parent's permissions might also be out-of-date, this process is repeated recursively until the

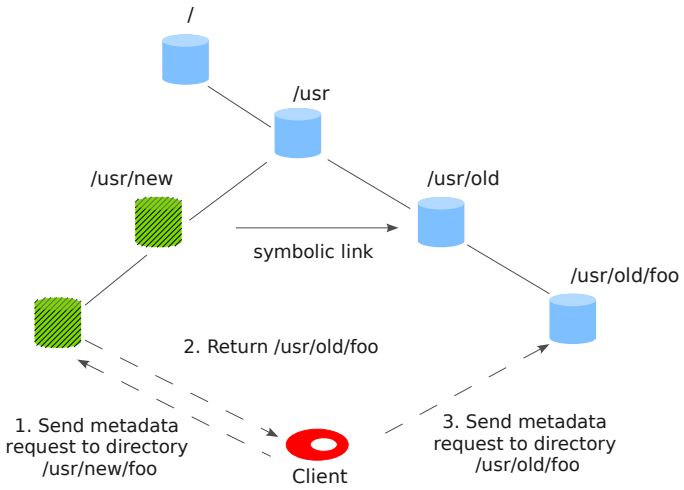

Fig. 3 Access to a directory containing a symbolic link. Striped cylinders represent OSD+s which "have" directories objects whose pathnames contain symbolic links (note that objects are actually stored in other OSD+s).

directory whose permissions have changed, or an updated directory, is reached. In this way, permissions are updated in a lazy fashion, minimizing the part of the hierarchy which is traversed.

3.5 Links

Our design makes the implementation of hard links (which are only possible between regular files) straightforward. The directory entry of a regular file comprises the name of the file and the *id* of the object containing the file's permissions. The object's *id* is a sort of i–node number which makes the creation of hard links direct: the directory entry for a hard link merely stores the new file name and the same object *id* of the source file.

However, more complex scenarios appear with symbolic links, since they can happen between directories. Any access to the subtree underneath a linked directory will fail, like an access to a renamed directory whose object has not been migrated yet. LH [17] proposes the creation of shortcuts to deal with files whose pathnames contain symbolic links. A shortcut to one of these files is created the first time the file is accessed by traversing the directory hierarchy. Any subsequent access to the the same file will use the shortcut, avoiding the hierarchy traversal. This approach, however, presents two problems: (a) shortcuts take up space and, (b) when a file access fails, there is no way to know if the failure is due to a missing file or a symbolic link in the name. The ambiguity in (b) produces the traversal of the directory hierarchy up to the root directory when accessing to any missing file.

Our proposal to tackle with symbolic links is simpler, and does not suffer the missing file problem of LH. In FPFS, a symbolic link is treated as a directory rename. The differences are: any access to a directory containing a symbolic link never

produces the directory's migration, and a client accessing one of these directories receives the resolved path to contact with the original OSD+ (see Figure 3).

3.6 Atomicity

An important aspect is that all the metadata operations must be atomic to provide a consistent parallel file system. When a metadata operation is performed by a single OSD+ (e.g., `create`, `unlink`, etc.), the backend file system itself guarantees the atomicity and POSIX semantics of the operation. However, operations such as `rename`, `mkdir` or `rmdir`, usually involve two OSD+s. Now, atomicity is jointly guaranteed by means of the backend file system, and a *three–phase commit protocol* (3PC) [19], where one node acts as the *coordinator* directing the remaining nodes or *participants*.

The three-phase commit protocol proceeds as follows. In the first phase, the coordinator checks whether the operation can be performed or not by asking the participants. Next, the coordinator verifies all the nodes are ready to commit the operation. In case all are prepared, the coordinator finalizes the protocol sending a commit message. After this last step, the transaction will not be aborted. Although the participants should acknowledge the commit message, the operation will be committed anyway.

4 The Metadata Cluster: Implementation

A prototype of the metadata cluster has been built on Linux. Each OSD+ is a user–space multithreaded process, running on a mainstream computer, which uses a conventional file system as backend (see Figure 4). The Linux syscall interface is used to access the local file system, which must be POSIX–compliant (remember that we want to export some characteristics of the underlying system to the parallel file system), and support extended attributes (used by our implementation; see Section 4.4).

For every new established connection from a client or another OSD+, a thread is launched in the target OSD+. The thread lasts as long as the communication channel remains open; hence, performance is improved due to the absence of connection establishments and termination handshakes per message. In the current implementation, TCP/IP and UDP/IP protocols are used.

In order to evaluate FPFS on metadata workloads, we have built a skeleton file system; that is, we have not yet implemented data operations, data striping, fault detection, recovery and other amenities, which are adequately implemented in many cluster file systems and can be borrowed from them.

4.1 Directory Objects

Internally, a directory object is implemented as a regular directory whose pathname is its directory pathname in the parallel file system. Thus, the directory hierarchy is imported within each OSD+ by replicating a partial namespace of the global hierarchy.

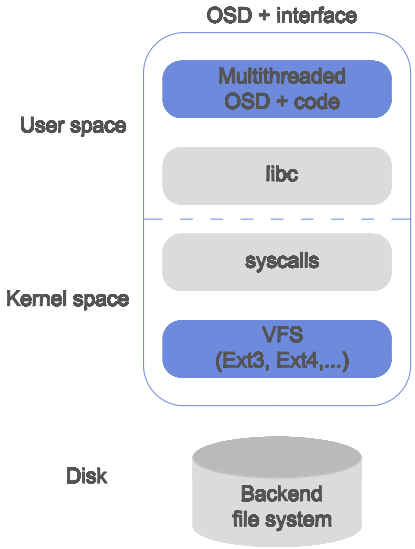

Fig. 4 Layers composing the OSD+.

In order to preserve the hierarchy, directory objects maintain an entry for every file and subdirectory they contain. These entries are implemented as empty files and directories, respectively, in the local file system. As we can see, an OSD+ can internally use several directories in the local file system for different purposes. Altogether, there are four types of directories, differentiated through extended attributes: a first type to implement directory objects; a second one to maintain the hierarchy (e.g., the subdirectories); a third to internally construct the paths of the directory objects; and finally, temporary directories to keep renamed metadata which has not been migrated yet.

Figure 5 shows how an FPFS's directory hierarchy is mapped to a four–OSD+ cluster. There are one regular file (`info.pdf`) and six directories: `/`, `home`, `usr1`, `usr2`, `usr3` and `docs`. Directory objects (marked with **o**) are stored in OSD+s 0, 1, 1, 3, 0 and 2, respectively. Realize that a directory object and its corresponding parent's directory object are usually placed in different OSD+s, except for `/home/usr1`, where both meet, by chance, in the same OSD+. Directories used for maintaining the hierarchy are identified by **h**. Their names will appear as subdirectories during a directory object's scan, along with the names of the regular files in the directory object. Finally, directories used internally for constructing the paths of directory objects, do not possess any extended attribute (for instance, `/home` and `/home/usr2` in OSD+ 2 are used for supporting the directory object of `/home/usr2/docs`). Note that the figure does not show any temporary directory (marked with **t**), because no rename has occurred.

As depicted in Figure 5, some directory can have extended attributes **h** and **o** set at the same time. This is the case for `/home/usr1`, since its directory object is in OSD+ 1 (hence the **o**), and because it is a subdirectory of `/home`, whose directory object is also in OSD+ 1 (hence the **h**).

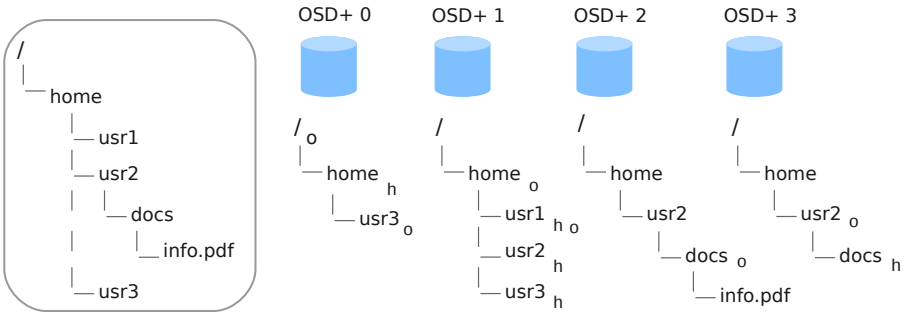

Fig. 5 Implementation of the parallel file system hierarchy in the OSD+ devices.

4.2 Huge directories

In the current implementation, FPFS distributes a directory when its size is bigger than 244 kB. The distribution is dynamic: initially, a directory is managed by a single OSD+ and directory object; as soon as a directory is identified as huge, a distribution process moves its directory entries from the primary OSD+ to the secondary OSD+s. All the directory entries are moved except those that belong to the primary OSD+, according to the file–level hash function (see Equation 2).

During a directory redistribution, no requests on the affected directory are attended in the primary OSD+ until the redistribution finishes. Requests received during the process will be returned to the clients informing them that the directory is now distributed, and that they should send their requests again taking into account the file–level hash function.

A directory can be in three different states (internally marked with extended attributes): *not distributed*, *migrating*, and *distributed*. The first state is the most common; all the directories smaller than 244 kB belong to this state. The second state lasts as long as the redistribution process does, and indicates that no requests can be performed on the directory. The third state is set once the redistribution process finishes; it also says that clients should be informed of the new distributed state of the directory.

Finally, realize that all the secondary directory objects, that store entries of huge directories, are also marked as “secondary” inside every OSD+.

4.3 Client–OSD+ Interaction

Communications between clients and OSD+s are established via TCP/IP connections and request/reply messages. Each OSD+ will launch one thread for attending the requests of a client, and for performing the operations on the local disk on behalf of the client. This way, the workload generated by the clients is reflected on the servers, which can be configured accordingly. Note that file systems like Lustre limit the number of threads on the metadata server [20], and that this number is usually much smaller than the number of clients; this decision distorts the clients’ workload that the server sees.

As requests, FPFS supports the most frequently used metadata operations (see Table 1): `mkdir`, `rmdir`, `opendir`, `readdir`, `create`, `unlink`, `open`, `close`, `lookup`, `stat`, `utime` and `rename`.

A client request is usually sent to the OSD+ storing the parent's directory object. For instance, if a client opens `/home/usr2/docs/info.pdf`, it sends a message to the OSD+ containing the directory object of `/home/usr2/docs`. One exception to this rule is `opendir`, where the target OSD+ is that corresponding to the pathname given as argument to the operation. Another particular case is `rename`, which receives two pathnames (an old path and a new one). In this case, the request is sent to the OSD+ containing the object of the new path.

When an operation involves several OSD+s, the OSD+ contacted by a client carries out the operation collaborating with other OSD+s. For example, to create `/home/usr2` (see Figure 5), OSD+ 1, which contains $\texttt{/home}_{\texttt{o}}$, initially creates the directory entry $\texttt{/home/usr2}_{\texttt{h}}$. If the creation is successful, OSD+ 2 completes the request creating the directory object $\texttt{/home/usr2}_{\texttt{o}}$.

4.4 Files and Data Objects

An FPFS file's metadata is initially stored as metadata of an empty file (basically, an i–node) in its parent directory. This improves operations like `stat`, since the directory entry and its metadata are in the same OSD+. This also exports file attributes in the local file system to the parallel file systems, so clients are able to see all the usual attributes (timestamps, mode, etc.) and extended attributes stored in the empty file.

Despite that we are only interested in metadata operations, to make a fair comparison with Lustre (see Section 5), FPFS also creates data objects for files. They are implemented as regular files in the OSD+s. Each data object has an *id*, called *OID*, which is stored as an extended attribute in its corresponding FPFS file's metadata. An OID is made up of two values: an object name (a random number; also used for the name of the internal regular file storing the data), and an OSD+ number (where the data object is stored).

Two different policies can be used to select the OSD+ where a data object is created: *same OSD+* and *random OSD+*. The former (used for obtaining the experimental results showed in Section 5) stores a data object in the OSD+ of its file's directory, thus reducing the network traffic during file creations because no other OSD+s participate in the operation. The latter chooses a random OSD+ instead. This second approach achieves a better use of resources, by keeping a more balanced workload, although it increases the network traffic during creations.

Although data operations have not been implemented yet, realize that clients will be able to access data objects through OIDs stored in files' metadata. An OID indicates the destination OSD+ and the assigned object name of a file's data object, so clients can directly send data requests to the given OSD+.

4.5 Logs

Lazy techniques require each OSD+ to store a *metadata log* with permission changes and directory renames. All the incoming requests are first checked against this log to provide a coherent and consistent reply to clients accessing metadata that may not have been updated yet.

Aside from the metadata log, the three–phase commit protocol (see Section 3.6) employs another log to rollback in case of failure. Both logs are sync'ed to disk every 5 seconds, which is the time usually used by file systems like Ext3 [21]/Ext4 [22] to commit their metadata.

4.6 Security

File systems must also prevent clients from doing malicious operations on the system. The current implementation entirely runs in user–space for fast prototyping and evaluation. However in a production system, the client side of the file system should be implemented inside the kernel, and applications should access the cluster file system through the VFS interface.

Authentication of clients against servers should occur at mount time. To this end, mechanisms as Kerberos [23], or that described in the OSD standard [24], can be used.

5 Experimental Results

The performance of the FPFS's metadata cluster has been evaluated and compared with Lustre's. We have also analyzed its scalability, and its throughput for huge directories. This second section describes the system under test, the benchmarks run, and the experimental results achieved.

5.1 System under Test

The testbed system is a cluster made up of 16 compute and 1 frontend nodes. Each compute node has two Intel Xeon E5420 Quad-core CPUs at 2.50 GHz, 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 distribution which also supports Lustre 1.8.2. The other disk, used as test disk, is exported as either an FPFS OSD+ or a Lustre MDS–MGS/OST server. The interconnect is a Gigabit network with a D-Link DGS-1248T switch.

Experiments use up to 12 out of the 16 nodes. For FPFS, several configurations are set up, each with a different number of OSD+s or backend file system. For Lustre, only one configuration is set up with just one node running all the Lustre services (MGS/MDS, and one OST), which is equivalent to an FPFS configuration with only one OSD+.

For clients, 1 to 4 nodes are used depending on the test. Since we have not detected either a CPU or network bottleneck in the clients during the experiments, several processes are run per CPU and core (up to 256 altogether) to analyze the servers' performance under heavy workloads.

Since the ldiskfs file system, used by Lustre, can be considered as something between Ext3 and Ext4 [22], FPFS has been evaluated using both file systems as backend to make a fair comparison with Lustre. This way, we will show that the improvements achieved by FPFS over Lustre are due, in many cases, to the smaller overhead and better performance provided by OSD+s and the metadata cluster implementation in FPFS. The use of different file systems will also show that each file system works better for different workloads, and that FPFS can easily be configured to use the proper file system for a given workload.

Metadata performance also depends on the options used for formatting a file system. Therefore, for the sake of comparison, the Ext3 and Ext4 file systems, used by FPFS, have been formatted with the same options as Lustre uses in ldiskfs. Other configuration issues, that may affect the performance of Lustre, have been considered too, following the recommendations in the Lustre operations manual [20].

Several issues regarding Lustre should be remarked. Lustre 2.0 includes new functionality to support a metadata cluster, but a production–ready service will not be available until version 2.2 [6]. We have not found information to set up the service either. Lustre 2.0 has also been modified to support several file systems as backend, although, to date, only a customized Ext3 file system ("ldiskfs") is fully supported. Finally, we run the tests on version 2.0.0.1, but the results were generally worse than on version 1.8.2. Therefore, we have discarded this version of Lustre and its results are not presented here.

Finally, we have also tried to evaluate the latest version of the Ceph's metadata cluster [5], but different problems have prevented us from succeeding: an excessive memory use which produces swapping for some workloads, frequent kernel panics, and a poor performance in many cases.

5.2 Benchmarks

Through the experiments, we have analyzed three different aspects. First, we have evaluated and compared the performance of FPFS and Lustre. This has been done by using the *HP Trace*, *creation/traversal of directories*, and *metarates* benchmarks. Second, we have evaluated FPFS's scalability by using those same benchmarks. And third, we have also tested FPFS's throughput for huge directories, by using the *hugedirs* micro-benchmark. The benchmarks used are the following:

- *HP Trace*: this benchmarks replays a 21-hour trace collected in 2002 which is, in turn, a subset of a 10-day trace of all file system accesses done by a medium-sized workgroup using a 4-way HP-UX time-sharing server attached to several disk arrays and a total of 500 GB of storage space [25].
  The selected period, one of the most active, covers from 6am on the fifth trace day to 3am on the next day. Table 1 shows an overview of the metadata requests

Table 1 Overview of the 21-hour HP trace.

Operation type	Count	Operation type	Count
Lookup	13908189	File rename	7683
Stat	2827387	Mkdir	7389
Open	2572124	Rmdir	6973
Unlink	67883	Directory rename	5
Create	41755		

in the trace. Since we are only interested in metadata operations, data operations (mainly, `read`, `write` and `mmap`), present in the trace, are omitted.
The trace is replayed by a multithreaded program that simulates a system with concurrent metadata operations. The program takes into account dependencies between operations (e.g., a file can not be created before the directory containing the file is created).

- *Creation/traversal of directories*: this benchmark is made up of two tests: the first one creates directory hierarchies with empty regular files, and the second one traverses those hierarchies. Each directory hierarchy is created by a single process by uncompressing a Linux kernel 2.6.32.9 source tree whose files have been truncated to zero bytes. Each process also traverses its own Linux source tree.
- *Metarates*: this program [26] evaluates the rate at which metadata transactions are performed. It measures aggregate transaction rates when multiple processes (coordinated by MPI) read or write metadata concurrently. We use 640,000 files in total, distributed into as many directories as processes. The program tests the performance achieved by each system for three types of metadata transactions[1]: create–close, stat, and utime calls, which basically generate a write–only, read–only and read–write metadata workload, respectively.
- *Hugedirs*: it is a microbenchmark made up of three different tests. The first one creates a fixed number of files on a single directory. That number depends on the number of OSD+s. For each OSD+, 400,000 files are created, so there are 400,000 files with 1 OSD+, 800,000 files with 2 OSD+s, and so on. The second test performs a `stat` operation on each of those files. Lastly, the third test performs an `unlink` operation on the directory's files. Operations are performed distributing and not distributing the directory in order to compare the distribution's performance.

5.3 Results

The results shown for every system configuration are the average of five runs of each benchmark. Confidence intervals are also shown as error bars, for a 95% confidence level. Test disks are formatted between runs for the *HP Trace* and *metarates* benchmarks, and unmounted/remounted between the directory tree creation and traversal

[1] Note that the same create–close and stat metadata workloads can be generated by more up–to–date benchmarks like mdtest [27]. However, unlike mdtest, metarates also supports utime operations, which read and write the same metadata element (an i–node in our case) in each transaction.

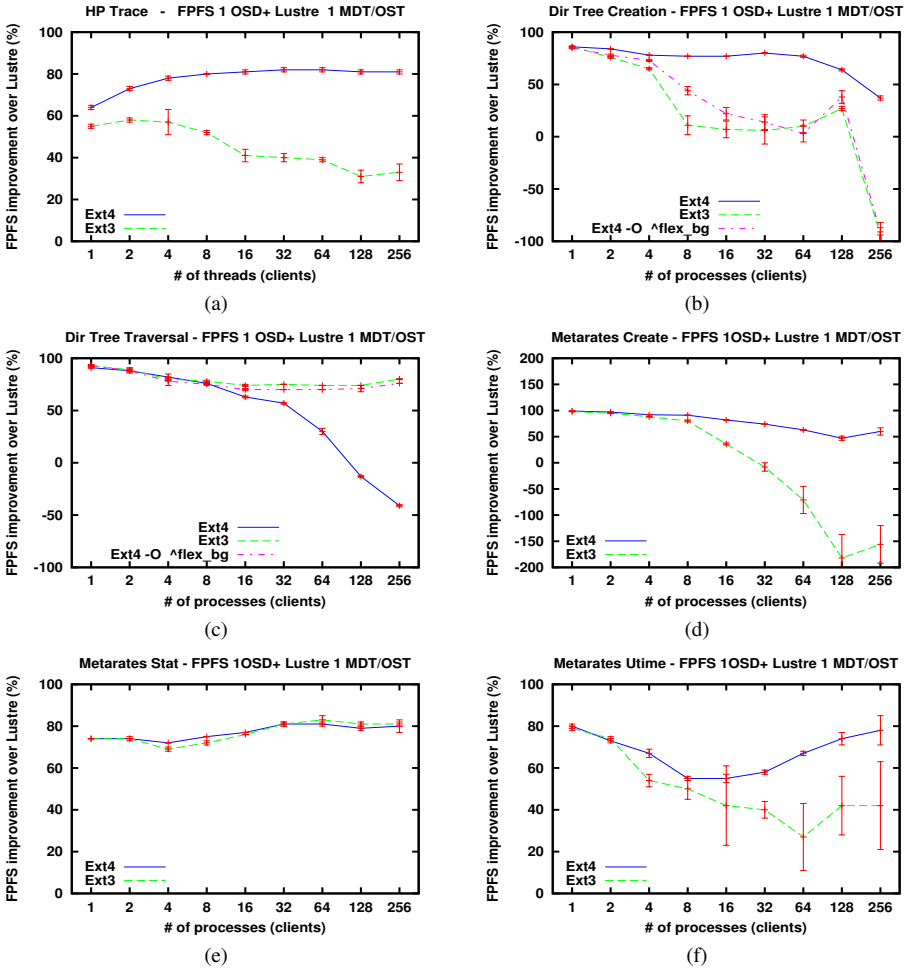

Fig. 6 Improvement obtained by FPFS 1OSD+ over Lustre: (a) HP Trace; (b) Creation of directories; (c) Traversal of directories; (d) Metarates: create-close transactions; (e) Metarates: stat transactions; (f) Metarates: utime transactions. Note the different Y–axis scales.

tests, and between the create, stat and unlink tests of *hugedirs*. The number of client processes per benchmark varies from 1 to 256 processes, in powers of two, except for the *hugedirs* benchmark that is set to 256 clients.

The results in the *hugedirs* benchmark are calculated by distributing and not distributing the huge directory, and for 1, 2, 4 and 8 OSD+s. For the remaining benchmarks and FPFS, results are obtained by using 1 and 4 OSD+s. Finally, FPFS's and Lustre's performances are compared by using only one node. which acts either as an FPFS OSD+ device or as a Lustre server, containing both the MDS/MGS service and one OST.

In Figure 6 we present FPFS's improvement over Lustre for *HP Trace*, *creation/traversal of directories*, and *metarates* benchmarks. Figure 7 shows, for the

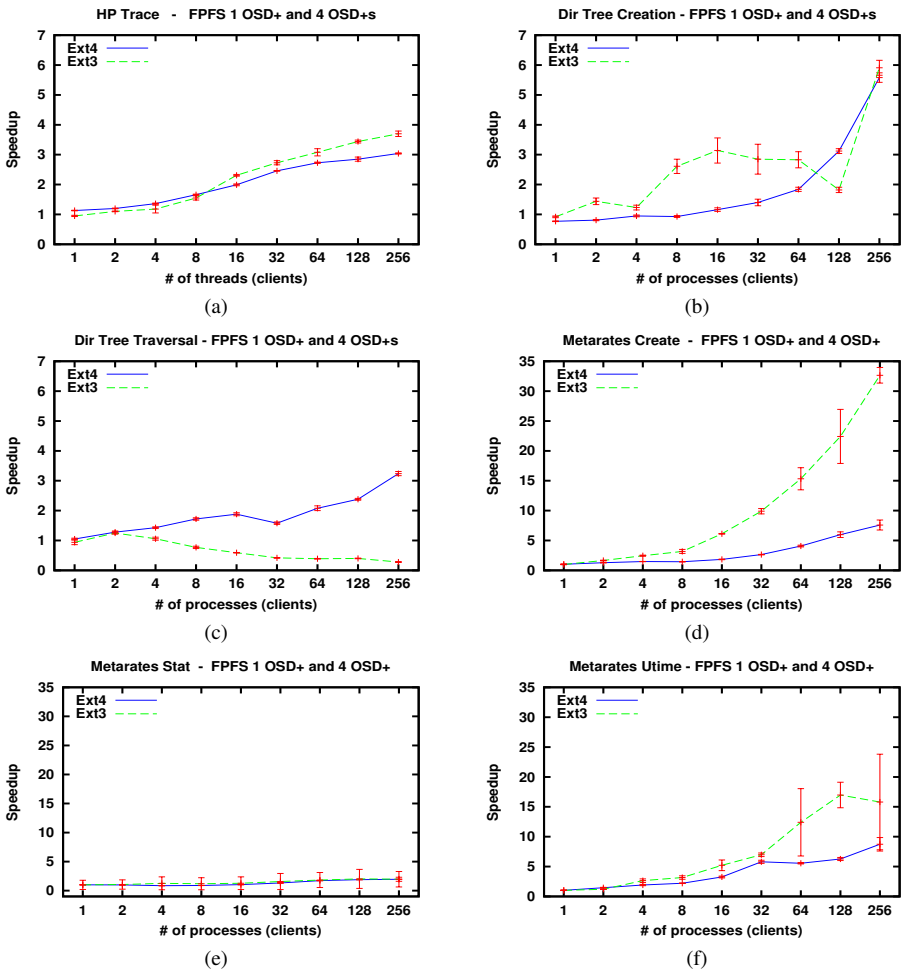

Fig. 7 Scalability for FPFS 1 OSD+ and 4 OSD+s configurations: (a) HP Trace; (b) Creation of directories; (c) Traversal of directories; (d) Metarates: create-close transactions; (e) Metarates: stat transactions; (f) Metarates: utime transactions. Note the different Y–axis scales.

same benchmarks, the scalability for FPFS with 1 OSD+ and 4 OSD+s. Finally, Figure 8 depicts the throughput and scalability for the *hugedirs* benchmark.

5.3.1 HP Trace

Figure 6.(a) compares the performance obtained by FPFS and Lustre in the *HP Trace* benchmark, when the number of threads varies from 1 to 256 in the program which replays the trace. For Ext4 and one thread, our proposal is 64% better than Lustre, and the percentage increases with the number of threads, reaching an improvement of 82% from 16 threads on.

It is worth remarking that, although Lustre is a full-fledged parallel file system and FPFS only implements an incomplete metadata service, both file systems roughly perform the same operations. This fact, along with the large performance differences in this test, ensures that FPFS represents a significant improvement over Lustre in time-sharing environments. Moreover, FPFS outperforms Lustre regardless of the backend file system used, mainly due to the thin layer FPFS adds on top of the backend file system, which directly translates FPFS requests into backend file system requests. Instead, Lustre adds several abstraction layers which increase the service time.

FPFS's scalability, shown in Figure 7.(a), reaches 3.04 for Ext4, and 3.70 for Ext3, when there are 256 threads. This value is smaller than the ideal 4, due to the dependencies between the operations in the trace, which limit the parallel execution of operations. However, as the number of threads increases, so does the number of possible ongoing metadata operations. Accordingly, scalability is better for a large number of threads, showing that FPFS can properly deal with large time-sharing systems.

5.3.2 Creation/Traversal of Directories

Like HP Trace, this benchmark creates/traverses thousands of files and directories (the Linux source tree used by each process has around 14,000 directories and 50,000 regular files). However, unlike the previous one, there are not dependencies between metadata operations carried out by different processes. Another difference is that not all the file system operations are needed (specifically, only the `create`, `opendir`, `close`, `mkdir` and `getdents` metadata operations are used).

Figures 6.(b) and 6.(c) show, respectively, that FPFS's improvement over Lustre can reach 86% during directory tree creations, and more than 90% for directory tree traversals, although the results greatly depend on the file system used and the number of processes.

The different behavior of Ext3 and Ext4 is due to an exclusive Ext4's option, *flex_bg*, used by default when the file system is created. This flag improves the creation of directories, but downgrades directory traversals for more than 64 processes. However, when *flex_bg* is unset (lines labeled "Ext4 -O ^flex_bg" in the figures), Ext4 roughly behaves as Ext3. Note that file–system configurations that provide good performance for directory tree creation and traversal are different, and you cannot get good results (with respect to Lustre) on both tests at the same time. Hence, in these tests, the underlying file system and formatting options can be decisive. Due to its flexibility, FPFS can easily be set up to get the best performance depending on the workload.

Figures 7.(b) and 7.(c) plot the scalability for directory tree creation and traversal for Ext4 and Ext3, each using its default value for *flex_bg*. In both tests, the scalability achieved for Ext4 increases with the number of clients. For the creation test, it is even larger than 4, which can be explained by looking at Figure 6.(b): with 256 clients and 4 OSD+s, every OSD+s is serving around 64 clients, and the performance of 1 OSD+ for 64 clients is much better than for 256 clients.

In the case of Ext3, the scalability is also quite good for the directory tree creation, but results for the traversal test are rather bad. We have not found a plausible explanation yet.

5.3.3 Metarates

Performance results of metarates transactions are shown in Figures 6.(d)-(f). FPFS over Ext4 noticeably outperforms Lustre for all types of transactions and number of processes. FPFS over Ext3 has the same behavior as over Ext4 in the stat benchmark, and also achieves good results in the utime test, but it performs badly in the create–close benchmark, where the results are consistent with those obtained by Ext3 in the creation of a directory tree (see Figure 6.(b)).

The good results obtained by FPFS in the stat and utime tests are because these tests first create the files, and then perform the corresponding operations. Accordingly, thousands of i–nodes and directory entries are already in the operating system's caches when the stat and utime operations start. Hence, the performance is limited by CPU and network bandwidth, and not by hard disks or directory sizes. Lustre's abstraction layers, however, introduce a larger overhead which downgrade its performance.

Figures 7.(d) and 7.(f) show the scalability achieved by FPFS in the create–close and utime tests. The super–linear scalability achieved can be explained by the use of the operating system's caches in the OSD+s, and the number of processes itself. Since the operating system uses write-back caches, metadata writes are delayed a few seconds (typically, 5 seconds in Linux) in main memory before being written to disk. When having more processes, the increase of OSD+s reduces the application time, and this, in turn, reduces the number of write operations to disk during the run of the tests. Also, as the total cache size grows, so does the amount of metadata in main memory that has to be written to disk after the test has finished. The completion of these last pending metadata writes, hence, does not impact the application time of the benchmarks. All this explains the big confidence intervals for utime too, because the amount of metadata written to disk significantly varies from run to run, and so does the application time.

The larger total cache size provided by four OSD+s also decreases the number of metadata reads from disk. As explained before, metarates creates the files required for the stat and utime benchmarks just before running them, leaving thousands of i–nodes and directory entries in main memory that do not have to be read from disk. With four OSD+s, the amount of metadata in main memory just after creating the files is four times larger than with only one OSD+, which accordingly reduces the amount of metadata read from disks. Stat and utime transactions benefit from this fact.

Finally, in the stat case, we can see that the scalability slightly increases with the number of clients, although it is clear from Figure 6.(e) that clients already achieve the maximum possible performance with a single OSD+. Therefore, the use of four OSD+s can hardly reduce the application time.

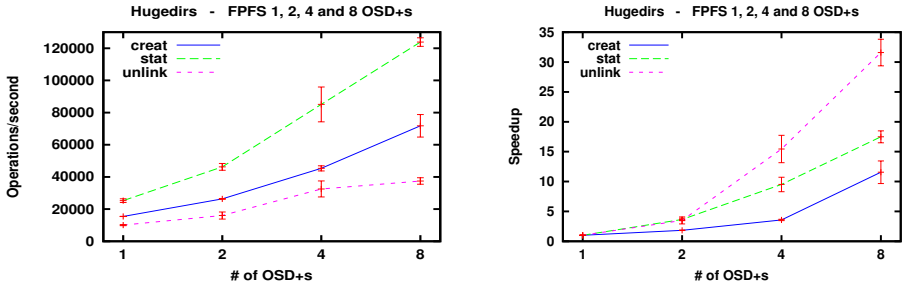

Fig. 8 (a) Operations per second achieved by FPFS when distributing a huge directory. The size of the directory is N*400,000 files, where N is the number of OSD+s. (b) Scalability achieved by FPFS when distributing a huge directory with 400,000 files. Each OSD+s receives 400,00/N files, where N is also the number of OSD+s.

5.3.4 Huge directories

Figure 8.(a) depicts the rates at which operations are performed when we distribute a huge directory. As we can see, FPFS is able to create around 70,000 files per second, and to stat more than 120,000 files per second, with just 8 OSD+s. These numbers exceed today's requirement for 40,000 files creates per second in a single directory [28], and prepare FPFS for the Exascale–era.

Figure 8.(b) shows the scalability for FPFS using the hugedirs benchmark. The speedup is computed by comparing the performance obtained when not distributing and when distributing the files of the single created directory. Accordingly, for one OSD+, the speedup is 1 because no distribution is possible. As the number of OSD+s increases, we obtain an outstanding performance, achieving a super–linear scalability for all tests.

That super–linear scalability is mainly due to the local file system used in the OSD+s. For this benchmark, we used the Ext4 file system, whose performance gets slightly worse as the number of entries in a directory grows. Hence, by distributing the management of a directory, we are not only sharing out the workload among various servers, but also creating smaller directories (e.g., smaller secondary directory objects) on the local file systems, and thus improving their local performance.

The above good results have been obtained with a dynamic distribution of the huge directory. As we have previously explained (see Section 4.2), a directory is initially managed by a single OSD+ and directory object; as soon as a directory is identified as huge (i.e., its size is greater than 244 kB), a distribution process starts to move directory entries from the primary OSD+ to the secondary OSD+s. We have measure the time taken by this migration and it is usually small (less than 2 seconds), so it does not hurt FPFS performance.

Finally, note that, given the lengths of the file pathnames in this hugedirs benchmark, the threshold of 244 kB, which decides when a directory is "huge", is equivalent to distribute directories when they contain more than 8,000 files. This is the same threshold as that used by GIGA+ [2] to decide when to split a partition, and it is

based on the observation that 99.99% of the directories contain less than 8,000 files (see Section 3.2).

6 Related Work

An important issue regarding the metadata management in a parallel file system is where to store metadata. Lustre [7] and Hadoop [9] store metadata in a single metadata server, while PVFS [4] uses one or more servers to store metadata. Ceph [5], however, stores metadata in objects located in the OSDs, although the management is performed by a small set of servers which form the metadata cluster and contact the OSDs to read and write metadata.

Ali *et al.* [15] also explore the use of OSD devices to store and partially manage directories. They save directory entries as attributes of empty objects, and enhances the OSD standard with a new *compare–and-swap* operation to make attribute changes atomic. Note that, despite this operation, OSDs are basically passive in their approach with respect to the metadata management, which has to be done by a small set of dedicated servers, as in Ceph. Other issues which are important to provide a complete metadata services, such as directory distribution, renames and permission changes, atomicity of operations involving several OSDs, etc., are not discussed in their work either.

In FPFS, unlike the above approaches, all the OSD+s actively participate in the storage and management of a complete directory hierarchy. OSD+s also profit the features of the underlying local file systems, avoiding the insertion of many new data structures and thick software layers.

The namespace distribution across the metadata servers is another important subject which is crucial to balance the use of resources, and to achieve a good performance. The namespace distribution may also determine some scalability problems, related to certain metadata operations or changes in the cluster due to additions, removals or failures of servers. Static Subtree Partition (used by Coda [29], AFS [30], etc.) statically assigns portions of the file hierarchy to metadata servers. This preserves the directory locality, but is vulnerable to distribution imbalances as the file system and workload change. A variant is Dynamic Subtree Partition, used by Ceph [31], which delegates authority for directory subtrees to different metadata servers. Periodically, busy servers transfer subtrees to non–busy servers.

In order to improve the metadata distribution, either file [17,32,33] or directory [34] hashing approaches can be used. These schemes, however, present several drawbacks, such as the loss of the directory locality and massive data migrations due to, for example, a cluster size change or a rename. *Lazy Hybrid* (LH) [17] mitigates the migration problem with a global *metadata look–up table* (MLT) and several lazy policies. The MLT maps hash value ranges to *server ids*. When a new server is added or removed, the MLT is updated to reflect the new hash–range distribution, but no migration is performed. Lazy policies defer migrations until the affected data is accessed again. Since LH hashes files, it also includes a dual–entry *access control list* (ACL) to avoid expensive directory traversals when checking permissions.

Features introduced by LH have been widely borrowed by other approaches, such as *Dynamic Hashing* (DH) [32], DOIDFH [33], and MHS [34], which modify the way LH works to circumvent some of its shortcomings. DH combines lazy policies and an MLT with several new strategies to dynamically adjust the metadata distribution. MHS is a directory hashing scheme that use LH's access control mechanisms to avoid directory traversals; it removes data migrations, due to rename operations, by assigning, to every directory, a globally unique *id* which never changes. Directory *ids* are created by a single metadata server, which may become a bottleneck. There is also a global bucket index table that may need to be updated as new directories are created or deleted. DOIDFH is similar to MHS in that it assigns a globally unique *id* to every directory too. DOIDFH, however, keeps the complete directory hierarchy in every metadata server. This prevents metadata servers from becoming hot–spots, because a client can contact any server to obtain a directory's *id*. Another difference is that DOIDFH uniformly distributes file metadata across metadata servers by hashing <file name, parent directory's id> pairs. Realize that a metadata–intensive workload that frequently modifies the hierarchy can easily saturate a system deploying any of these approaches.

FPFS also adopts LH's techniques like pathname hashing, dual–entry ACLs, and lazy migrations, although they are only applied to directories. This is an important difference because a rename does not produce a massive migration of file data, only directory objects are migrated. Permission changes do not produce a massive update of files' ACLs either, because a file's permissions are directly derived from its own ACL and its parent directory's ACL. FPFS also uses a different hashing function [35], which minimizes metadata migration on cluster changes, and handles links in a more straightforward and efficient way.

The management of directories with millions of files, accessed by thousands of clients at the same time, is a problem recently identified in HPC systems by different authors [2, 9, 10, 36, 37]. To deal with this problem, several approaches have been proposed. Patil and Gibson [2], for example, introduce a POSIX–compliant scalable directory design, called GIGA+, that distributes directory entries over a cluster of server nodes. GIGA+ incrementally hashes a directory into a growing number of partitions, which are migrated among metadata servers for load balancing. Migrations are individually performed by the servers, without a system–wide serialization, synchronization or notification. Ceph [31] uses dynamic sub–tree partitioning of the namespace and hashes individual directories when they get too big or experience too many accesses. Finally, there also exists a proposal to provide Lustre with a clustered metadata service [6] where directories can be statically striped over several MDTs as files over several OSTs.

Our proposal for managing huge directories in FPFS is similar to that proposed for Lustre, although, unlike Lustre's, it is dynamic because 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 huge directories but on showing that distributed directories can be efficiently implemented in an OSD+–based metadata cluster.

7 Conclusions and Future Work

In this paper, we have introduced OSD+, a new type of OSD device which handles not only data but also metadata requests. OSD+s support *directory objects*, that store file names and attributes, and support metadata–related operations. As in a traditional OSD, data in an OSD+ is stored in data objects, which mainly support `read` and `write` data operations.

Our new OSD+ devices profit the existence of a local file system in the storage nodes. OSD+s directly map directory–object operations to directory operations in the underlying file system, hence exporting many features of the local file system to the cluster file system, and achieving a significant flexibility, simplicity and small overhead.

We have also presented the architecture of the FPFS parallel file system, based on OSD+s, and particularly the design and implementation of its metadata cluster. Thanks to the OSD+s, metadata is managed by all the servers in the cluster, improving the performance, scalability and availability of the metadata service. In such large metadata clusters, issues like directory distribution for load balancing, and atomicity of metadata operations with several participating OSD+s are important. We face them by uniformly distributing the directory objects among all the servers, and coordinating OSD+s through a network–commit protocol, respectively. Atomicity of metadata operations which involve a single directory are independently handled by every OSD+.

The performance of our metadata cluster based on OSD+s has been compared with Lustre's. The results show that an FPFS metadata cluster with a single OSD+ can improve the throughput of a Lustre metadata server by more than 60–80%. Scalability of our proposal has also been evaluated, and the results confirm that it scales with the number of OSD+s.

Lastly, we have also included in FPFS the management of huge directories, which are common in some HPC applications. A huge directory has millions of entries that are accessed and modified by thousands of clients at the same time. FPFS dynamically distributes a directory among several OSD+s when it surpasses a given number of files, thereby distributing its workload.

The evaluation shows that FPFS achieves a high throughput of more than 70,000 creates per second, and more than 120,000 stats per second, for huge directories on a cluster with just 8 OSD+s, and a super–linear scalability as the number of OSD+s increases. These numbers exceed today's requirements, and prepare FPFS for the Exascale–era.

As work in progress, we are currently analyzing which modifications have to be done in the OSD standard [24] to support directory objects.

References

1. 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 the 2004 Mass Storage Systems and Technologies*, Apr. 2004.

2. S. Patil and G. Gibson, “Scale and concurrency of giga+: File system directories with millions of files,” in *Proceedings of the 9th USENIX Conference on File and Storage Technology (FAST’11)*, Feb. 2011, pp. 15–30.
3. D. Roselli, J. Lorch, and T. Anderson., “A comparison of file system workloads,” in *Proceedings of the 2000 USENIX Annual Technical Conference*, June 2000, pp. 41–54.
4. R. Latham, N. Miller, R. Ross, and P. Carns., “A next-generation parallel file system for linux clusters,” *LinuxWorld*, pp. 56–59, Jan. 2004.
5. S. Weil., “Ceph: reliable, scalable, and high-performance distributed storage,” Ph.D. dissertation, University of California, Santa Cruz, (CA), Dec. 2007.
6. W. Di, “CMD code walk through,” 2009. [Online]. Available: http://wiki.lustre.org/images/7/70/SC09-CMD-Code.pdf
7. P. J. Braams, “High-performance storage architecture and scalable cluster file system,” 2008. [Online]. Available: http://wiki.lustre.org/index.php/Lustre_Publications
8. M. Mesnier, G. R. Ganger, and E. Riedel, “Object-based storage,” *IEEE Communications Magazine*, pp. 84–90, Aug. 2003.
9. K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The Hadoop distributed file system,” in *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, May 2010, pp. 1–10.
10. R. Freitas, J. Slember, W. Sawdon, and L. Chiu, “GPFS scans 10 billion files in 43 minutes,” IBM Almaden Research Center, Tech. Rep. RJ10484, July 2011. [Online]. Available: http://www.almaden.ibm.com/storagesystems/resources/GPFS-Violin-white-paper.pdf
11. B. Welch, M. Unangst, Z. Abbasi, D. Gibson, J. Mueller, B. Small, J. Zelenka, and B. Zhou., “Scalable performance of the Panasas Parallel File System,” in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, 2008.
12. D. Hildebrand and P. Honeyman., “Exporting storage systems in a scalable manner with pNFS,” in *Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies*, 2005.
13. E. Polyakov., “The elliptics network, http://www.ioremap.net/projects/elliptics,” 2009. [Online]. Available: http://www.ioremap.net/projects/elliptics
14. ——, “Parallel optimized host message exchange layered file system,” 2009. [Online]. Available: http://www.ioremap.net/projects/pohmelfs
15. N. Ali, A. Devulapalli, D. Dalessandro, P. Wyckoff, and P. Sadayappan., “An OSD-based approach to managing directory operations in parallel file systems,” in *IEEE International Conference on Cluster Computing*, 2008, pp. 175–184.
16. R. J. Honicky and E. L. Miller, “Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution,” in *Proceedings of the 18th International Parallel & Distributed Processing Symposium (IPDPS)*, Apr. 2004.
17. 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 IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies*, 2003, pp. 290–298.
18. 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 2010 USENIX Annual Technical Conference (ATC’10)*, 2010.
19. D. Skeen and M. Stonebraker, “A formal model of crash recovery in a distributed system,” *IEEE Transactions on Software Engineering*, vol. 9, pp. 219–228, May 1983. [Online]. Available: http://dx.doi.org/10.1109/TSE.1983.236608
20. Sun-Oracle, “Lustre tunning,” 2010. [Online]. Available: http://wiki.lustre.org/manual/LustreManual18_HTML/LustreTuning.html
21. S. Tweedie, “Journaling the Linux ext2fs Filesystem,” in *LinuxExpo’98*, 1998.
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
23. “Kerberos: An authentication service for computer networks,” *IEEE Communications Magazine*, vol. 32, pp. 33–38, Sep. 1994. [Online]. Available: http://gost.isi.edu/publications/kerberos-neuman-tso.html
24. INCITS Technical Committee T10, “SCSI object-based storage device commands-3 (OSD–3). Project t10/2128–d. Working draft, revision 02,” http://www.t10.org/drafts.htm#OSD_Family, July 2010.
25. Hewlett–Packard, “Fstrace,” http://tesla.hpl.hp.com/open-source/fstrace, 2002.

26. University Corporation for Atmospheric Research, "metarates," 2004. [Online]. Available: http://www.cisl.ucar.edu/css/software/metarates/
27. C. Morrone, B. Loewe, and T. McLarty, "mdtest HPC Benchmark," 2010. [Online]. Available: http://sourceforge.net/projects/mdtest
28. 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
29. M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, and E. H. Siegel., "Coda: A highly available file system for a distributed workstation enviroment," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 447–459, 1990.
30. J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith., "Andrew: A distributed personal computing enviroment," *Communications ACM*, vol. 29, pp. 184–201, March 1986. [Online]. Available: http://doi.acm.org/10.1145/5666.5671
31. S. A. Weil, K. T. Pollack, S. A. Brandt, , and E. L. Miller., "Dynamic matadata management for petabyte-scale file systems," in *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, Nov. 2004.
32. L. Weijia, X. Wei, J. Shu, and W. Zheng., "Dynamic hashing: Adaptive metadata management for petabyte-scale file systems," in *Proceedings of the 14th IEEE / 23rd NASA Goddard Conference on Mass Storage Systems and Technologies*, May 2006, pp. 159–164.
33. D. Feng, J. Wang, F. Wang, and P. Xia., "DOIDFH: an effective distributed metadata management scheme," in *Proceedings of the 5th International Conference on Compurational Science and Applications*, Oct. 2007.
34. J. Wang, D. Feng, F. Wang, , and C. Lu., "MHS: A distributed metadata management strategy," *The Journal of Systems and Software*, vol. 82, no. 12, pp. 2004–2011, July 2009.
35. S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn., "CRUSH: Controlled, scalable, decentralized placement of replicated data," in *Proceedings the 2006 ACM/IEEE Conference on Supercomputing*, Nov. 2006.
36. R. Hedges, K. Fitzgerald, M. Gary, and D. M. Stearman, "Comparison of leading parallel NAS file systems on commodity hardware," in *Petascale Data Storage Workshop 2010 (poster)*, Nov. 2010. [Online]. Available: http://www.pdsiscidac.org/events/PDSW10/resources/posters/parallelNASFSs.pdf
37. R. Wheeler, "One billion files: Scalability limits in linux file systems," in *LinuxCon'10*, Aug. 2010. [Online]. Available: http://events.linuxfoundation.org/slides/2010/linuxcon2010-wheeler.pdf