A Metadata Cluster Based on OSD+ Devices

Ana Avilés-González, Juan Piernas, and Pilar González-Férez
Dept. of Computer Engineering
University of Murcia,
Murcia, Spain
Email: {ana.aviles, piernas, pilar}@ditec.um.es

***Abstract*—We present the design and implementation of both an enhanced type of OSD device, the *OSD+ device*, and a metadata cluster based on it. OSD+s support *data objects* and *directory objects*. A directory object stores file names and attributes, and supports metadata–related operations. OSD+s profit the directory implementation and features of the underlying file systems used by the storage nodes, achieving a great flexibility, simplicity and small overhead. By using OSD+ devices, we show how a metadata cluster can effectively be managed by all the servers in a system, improving the performance, scalability and availability of the metadata service. The performance of our new metadata cluster has been evaluated and compared with Lustre's. 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.**

***Keywords*-Metadata cluster, OSD+, parallel file system.**

I. Introduction

Modern distributed storage systems deal not only with a large volume of data but also with an increasing number of files. Accordingly, an efficient metadata management becomes a fundamental aspect of a system's storage architecture to prevent bottlenecks and achieve the desired features of high performance and scalability [1].

Although metadata is usually less than 10% of the overall storage capacity, its operations represent 50%–80% of all the requests [2]. Metadata operations are also very CPU consuming, and a single metadata server can easily be overloaded by a few clients. Hence, to improve the performance and scalability of metadata operations, a cluster of servers is needed. PVFS [3] and Ceph [4], for instance, use a small set of servers as metadata cluster, and Lustre expects to provide a similar production–ready service for version 2.2 [5].

With respect to data, many modern cluster file systems [6], [3], [4] use hundreds or thousands of OSD [7] (Object Storage Device) 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. 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*), the directory objects store file names and attributes, and support metadata–related operations, such as the creation and deletion of regular files.

Although OSD+s are basically independent, they should also 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 on OSD+s 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 produces several benefits: (a) many features of the local file system (atomicity, POSIX semantics, etc.) are directly exported to the parallel file system for metadata operations of a single directory; (b) utilization of the resources of the storage nodes (CPU, memory, secondary storage, etc.) is increased; and (c) the software layer which creates the OSD+ interface is thin and simple, producing a small overhead and hopefully improving the 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. FPFS metadata cluster will be as big as the data cluster, effectively distributing metadata requests among as many nodes as OSD+s, and improving metadata performance and scalability. Metadata availability will also be increased because the temporal failure of a node only affects 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 all the operations, 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. Therefore, this paper only focuses on the FPFS metadata cluster.

1550-6533/11 $26.00 © 2011 IEEE
DOI 10.1109/SBAC-PAD.2011.12

IEEE computer society

Two main problems must be addressed in order to build a metadata service based on OSD+ devices: (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.

We have evaluated our metadata cluster and compared its performance with Lustre's [6]. The results show that a single OSD+ can improve the throughput of a Lustre metadata server by more than 60–80%, and that the performance of our metadata cluster scales with the number of OSD+s.

To sum up, the main contributions of this paper are: (a) design and implementation of OSD+ devices (focused on the support of directory objects), (b) design and implementation of a metadata cluster using OSD+s, and (c) evaluation of the performance achieved by this metadata cluster.

II. Related Work

An important issue is where to store the metadata. Ceph [4] uses objects located in the OSDs themselves, although the management is handled by a small set of servers which contact the OSDs to read and write metadata. Ali *et al.* [8] explore the use of OSD devices to store and partially manage directories. They save directory entries as attributes of empty objects, and introduce an operation to make attribute changes atomic. But OSDs are basically passive with respect to metadata operations. They do not discuss other important issues either: directory distribution, handling of renames and permission changes, atomicity of operations involving several OSDs, etc. In FPFS all the OSD+s actively participate in the storage and management of a complete directory hierarchy. OSD+s also take advantage of the features of the underlying file systems, avoiding the insertion of new data structures and software layers.

The namespace distribution across metadata servers is crucial to balance the use of resources and to get a good performance. It also determines 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 [9], AFS [10], etc.) statically assigns portions of the file hierarchy to metadata servers. It preserves directory locality, but is vulnerable to distribution imbalances due to changes. A variant is Dynamic Subtree Partition, used by Ceph [4], which delegates authority for directory subtrees to different metadata servers. Periodically, busy servers transfer subtrees to non–busy servers.

Hashing approaches can be used [11], [12], [13] to improve metadata distribution, but present drawbacks such as the loss of directory locality and massive data migrations due to, for example, a rename. *Lazy Hybrid* (LH) [11] mitigates the migration with a *metadata look–up table* (MLT) which maps hash value ranges to *server ids*. It also applies lazy policies to defer a migration until a data is accessed again, and includes a dual–entry *access control list* (ACL) to avoid directory traversals to check permissions.

Features introduced by LH have been widely borrowed by schemes such as *Dynamic Hashing* (DH) [13] or MHS [12]. DH combines lazy policies and an MLT with several strategies to dynamically adjust the metadata distribution. MHS is a directory hashing scheme that uses LH's access control mechanisms; it avoids data migrations due to rename operations by assigning to every directory a unique *id* which never changes.

FPFS also adopts LH's techniques like pathname hashing to distribute metadata, 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 directory's. FPFS also uses a different hashing function [4] which minimizes metadata migration on cluster changes, and handles links in a more straightforward and efficient way.

III. The Metadata Cluster: Design

The metadata cluster uses OSD+ devices to provide a high performance and scalable metadata service. It also profits them to tackle with directory renames, links and permission changes, in a consistent and atomic manner.

A. Metadata Distribution

FPFS distributes the directory objects (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 [4], a deterministic pseudo–random function that guarantees a probabilistically balanced distribution of objects through the system. For 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 results from hashing the directory's full pathname.

Hash partition strategies present different scalability problems during cluster resizing, renames and permission changes. When adding and removing nodes in the cluster, our design avoids the metadata migration or imbalance through CRUSH. Likewise, for minimizing rename overheads and permission changes, FPFS employs lazy techniques [11]. Nevertheless, note that, in our case, renames and permission changes only affect directories. The experimental results will show that these operations are infrequent in directories (similar results have recently been obtained by other authors [11]). This fact, along with the use of lazy

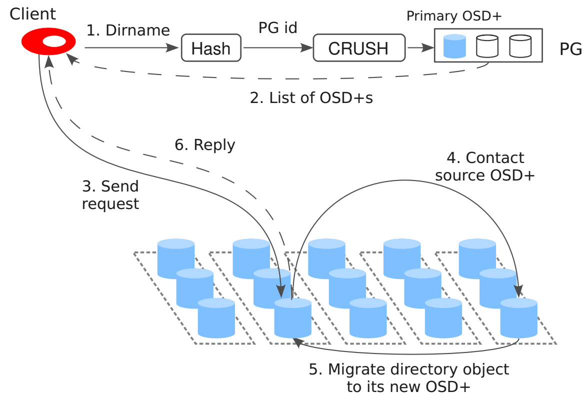

Figure 1. Directory object migration.

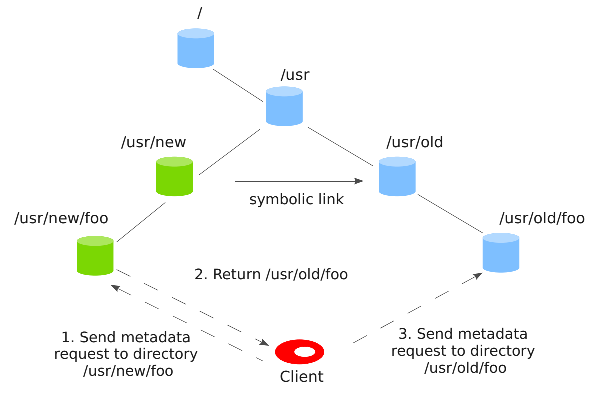

Figure 2. Access to a directory containing a symbolic link.

techniques and CRUSH, will further minimize the impact of these operations on the metadata cluster performance.

Albeit 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 directory), and to determine access permissions.

B. Directory Renames

If a directory name changes, so does its location and the location of the underlying directories in the hierarchy. This can incur a massive migration of metadata. To minimize this problem, lazy policies, similar to LH [11], are applied. Unlike LH, file renames do not produce migrations 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 (as an optimization, note that the rename message can be sent only to those nodes affected by the renamed path).

When an OSD+ receives an operation on a directory whose pathname has changed, but whose object has not been migrated yet, the OSD+ starts the migration of the object to carry out the operation instead of returning an error (see Fig. 1). 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.

C. Permission Changes

To directly determine access permissions and avoid directory traversals, dual–entry ACLs are used [11]. Given a directory, one contains its permissions, whereas the other its path permissions (the intersection of the directory's own permissions and its ancestors' path permissions) Only directories have dual–entry ACLs. A file's permissions are derived from its ACL, and its 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 path. If they exist, the parent directory is accessed to get its dual–entry ACL. Once permissions are updated, the requested ACL is calculated. Since parent's permissions might also be invalid, this process is repeated recursively until the changed directory or an updated directory is reached. In this way, permissions are also updated in a lazy fashion.

D. Links

Placing directories by hashing their pathnames presents the problem of locating the correct OSD+s for paths that include symbolic links. LH [11] proposes the creation of shortcuts to deal with files whose pathnames contain symbolic links. A shortcut to one of these files is created during the first access by traversing the directory hierarchy. Any subsequent access to the file will use the shortcut. This approach presents two problems: (a) shortcuts take up space and, (b) when a file access fails, we cannot know if it 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 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 Fig. 2).

E. Atomicity

An important aspect is that all the metadata operations must be atomic to provide a coherent view. 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 through the backend file system, and a *three–phase commit*

protocol (3PC) [14], where one node acts as the *coordinator* directing the remaining nodes or *participants*.

IV. 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. 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 (see Section IV-C).

For every new established connection from a client or another OSD+, one thread is launched. It 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 implemented in many cluster file systems and can be borrowed from them.

A. 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.

To preserve the hierarchy, directory objects maintain an entry for every file and subdirectory they contain. Hence, there are several 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 directories objects; and finally, temporal directories to keep renamed metadata which has not been migrated yet.

Fig. 3 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. Note 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.

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 [1].

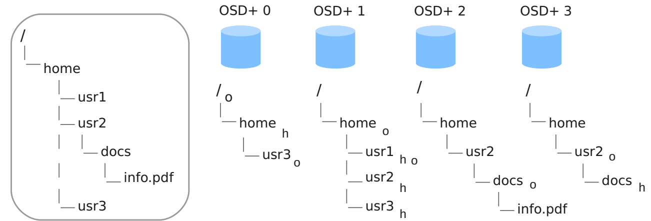

Figure 3. Implementation of the parallel file system hierarchy in the OSD+ devices.

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. But huge directories are common for some HPC applications, and new mechanisms would be necessary to deal with them[1].

B. Client–OSD+ Interaction

Communication between clients and OSD+s is established via TCP/IP connections and request/reply messages. As requests, FPFS supports the most frequently used metadata operations (see Table I).

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+ having the directory object of `/home/usr2/docs`.

When an operation involves several OSD+s, that contacted by the client carries out the operation collaborating with other OSD+s. For example, to create `/home/usr2` (see Fig. 3), OSD+ 1, which has `/home`$_o$, initially creates the directory `/home/usr2`$_h$. If the creation is successful, OSD+ 2 completes the directory object `/home/usr2`$_o$.

FPFS must also prevent clients from doing malicious operations on the system. Current implementation entirely runs in user–space for fast prototyping and evaluation. But in a production system, the client side of the file system would be implemented in the kernel, and applications would access the cluster file system through the VFS interface.

Authentication of the clients against the servers would occur at mount time. Mechanisms as Kerberos or that described in the OSD standard [15] could be used to this end.

C. Files and Data Objects

A file's metadata is initially stored as an empty file in its parent directory. This improves operations like stat, since the directory entry and its metadata are in the same OSD+. Clients are able to see all the usual attributes (timestamps, mode, etc.) and extended attributes stored in the empty file.

To make a fair comparison with Lustre, FPFS creates data objects for files, implemented as regular files in the OSD+s.

[1]Some of those mechanisms already exist like GIGA+ [1].

Table I
OVERVIEW OF THE 21-HOUR HP TRACE.

Operation type	*Count*
Lookup	13908189
Stat	2827387
Open	2572124
Unlink	67883
Create	41755

Operation type	*Count*
File rename	7683
Mkdir	7389
Rmdir	6973
Directory rename	5

Each data object has an *id*, stored as an extended attribute in its file's metadata, and is made up of 2 values: object name, and OSD+ where it is stored.

D. 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 3PC 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/Ext4 to commit their metadata.

V. EXPERIMENTAL RESULTS

The performance of the FPFS's metadata cluster has been evaluated and compared with Lustre's.

A. System under Test and Benchmarks

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 2.50GHz, 4GB of RAM, and two Seagate ST3250310NS disks of 250GB. On each node, one of the disks has a 64-bit Fedora Core 11 distribution which 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 8 nodes. For FPFS, two configurations are set up: one with 1 OSD+, and another with 4 OSD+s. For Lustre, only one configuration is set up with 1 node running all its services (MGS/MDS, and one OST), equivalent to our configuration with 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 in total) to analyze the servers' performance under heavy workloads.

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 [5]. 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 ("ldiskfs") is supported. Finally, we run the tests on the latest version 2.0.0.1, but the results were generally worse than in 1.8.2, so they are not presented here.

Since the ldiskfs can be considered as something between Ext3 and Ext4, FPFS has been evaluated using both as backend. Lustre are due, in many cases, to the smaller overhead and better performance provided by the OSD+ and metadata cluster implementation in FPFS.

The metadata performance also depends on the formatting options of a file system. FPFS has been using Ext3 and Ext4 formatted with the same options that 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 [16].

Finally, we have also tried to evaluate the latest version of the Ceph's metadata cluster [4], 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.

The following tests are used to evaluate and compare the performance of FPFS and Lustre on metadata workloads:

HP Trace: it is 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 [17]. The selected period, one of the most active, covers from 6am on the fifth trace day to 3am on the next day. Table I shows an overview of the metadata requests in the trace. Since we are only interested in metadata operations, data operations are omitted.

The trace is replayed by a multithreaded program that simulates a system with concurrent metadata operations takes into account dependencies between those operations.

Creation/traversal of directories: made up of two tests: the first creates directory hierarchies with empty regular files, and the second traverses those hierarchies. Every directory hierarchy is created by uncompressing the Linux kernel 2.6.32.9 source tree whose files are truncated to zero bytes. Each process accesses its own copy of the tree.

Metarates [18]: evaluates the rate at which metadata transactions are performed. It measures aggregate transaction rates when multiple processes read or write metadata concurrently. We use 640000 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[2]: create–close, stat, and utime calls.

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. The disk is formatted between runs, and unmounted/remounted between the directory tree creation and traversal tests. The number of client processes per benchmark varies from 1 to 256 processes, in powers of two.

For any benchmark, the scalability is calculated from 1 and 4 OSD+s results. Also, FPFS and Lustre's performances are compared using one node as either an OSD+ or a Lustre server containing both an MDS/MGS and OST.

B. Results

HP Trace: Although Lustre is a full-fledged parallel file system and FPFS only implements an incomplete metadata service, both roughly perform the same operations. This, along with the large performance differences in this test (see Fig. 4.(a)), which reaches 82% for 16/32 threads and Ext4, ensures FPFS represents a significant improvement with respect to 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.

FPFS's scalability, shown in Fig. 5.(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 limits 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.

Creation/Traversal of Directories: Figs. 4.(b) and 4.(c) show, respectively, that FPFS's improvement over Lustre can reach 86% during the directory tree creation, and more than 90% for the directory tree traversal, but it greatly depends 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 directory creation, but downgrades the directory traversal for more than 64 processes. However, when *flex_bg* is unset, Ext4 roughly behaves as Ext3. Hence, in these tests, the underlying system and formatting options can be decisive. Due to its flexibility, FPFS can easily be set up to get the best performance.

The scalability achieved for Ext4 increases with the number of clients (see Figs. 5.(b) and 5.(c)). It is greater than 4 for the creation test, which can be explained by looking at Fig. 4.(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.

For Ext3, the scalability can also be quite good for the directory tree creation, but results for the traversal test are rather bad. We have not found a plausible explanation yet.

Metarates: FPFS outperforms Lustre as shown in Fig. 4, except for Ext3 in the create–close test where it performs badly. That results are consistent with those obtained by Ext3 in the creation of a directory tree (see Fig. 4.(b)).

The great results in the stat and utime tests are because they first create the files, and then perform the operations, so thousands of inodes and directory entries are already in the caches. Hence, the performance is limited by CPU and network bandwidth, and not by hard disks or directory sizes. But, Lustre's abstraction layers introduce a larger overhead.

FPFS's scalability is super–linear in create–close and utime tests (see Figs. 5.(d) and 5.(f)), mainly due to the system's write–back caches, and the number of processes: when it is high, the increase of OSD+s reduces the application time, reducing the number of write operations to disk during the tests. The huge total cache size provided by four OSD+s also decreases the number of metadata reads from disk, which also improves the utime transactions. All this explains the big confidence intervals for utime too, because the amount of metadata written to disk greatly varies from run to run, and so does the application time.

In the stat test, the scalability slightly increases with the number of clients, although it is clear that, with a single OSD+, the clients almost get the maximum performance.

VI. Conclusions and Future Work

We have introduced OSD+, a new type of OSD device which handles not only data but also metadata requests. OSD+s support directory objects, which store file names and attributes, and support metadata–related operations. As in the traditional OSDs, data in OSD+s is stored in data objects, which mainly support `read` and `write` 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 system, hence exporting many features of the local file system to the cluster one, and achieving a great flexibility, simplicity and small overhead.

We have also presented a metadata cluster, based on OSD+s, for our FPFS file system. Metadata is managed by all the servers, improving the performance, scalability and availability of the service. Atomicity of metadata operations involving several OSD+s is guaranteed through a network–commit protocol, and by the local file system in each OSD+ for operations on a single directory.

[2]Note that the same create–close and stat metadata workloads can be generated by more up–to–date benchmarks like mdtest [19]. 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.

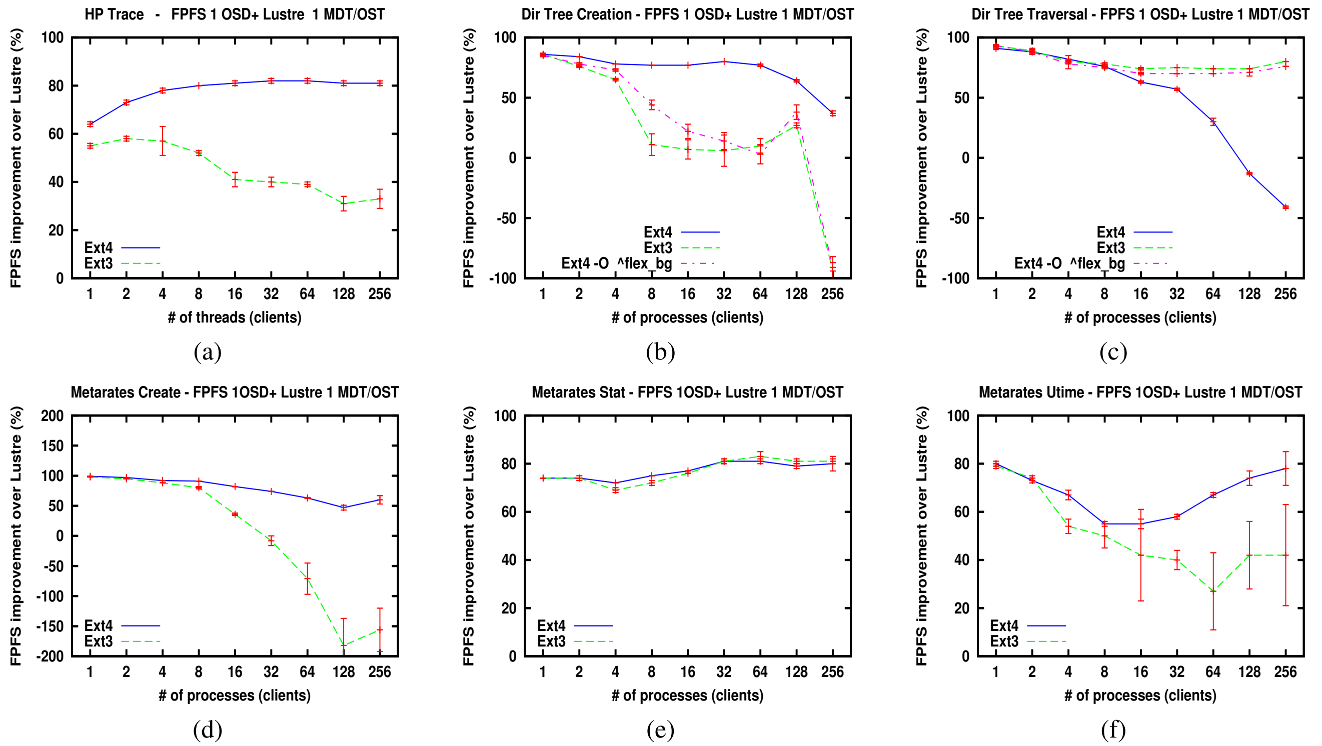

Figure 4. **Improvement obtained by FPFS 1OSD+ over Lustre**: (a) HP Trace; (b) Creation of directories; (c) Traversal of directories; (d) Metarates: create-close transactions; (e) Metarates: stat transactions; (f) Metarates: utime transactions.

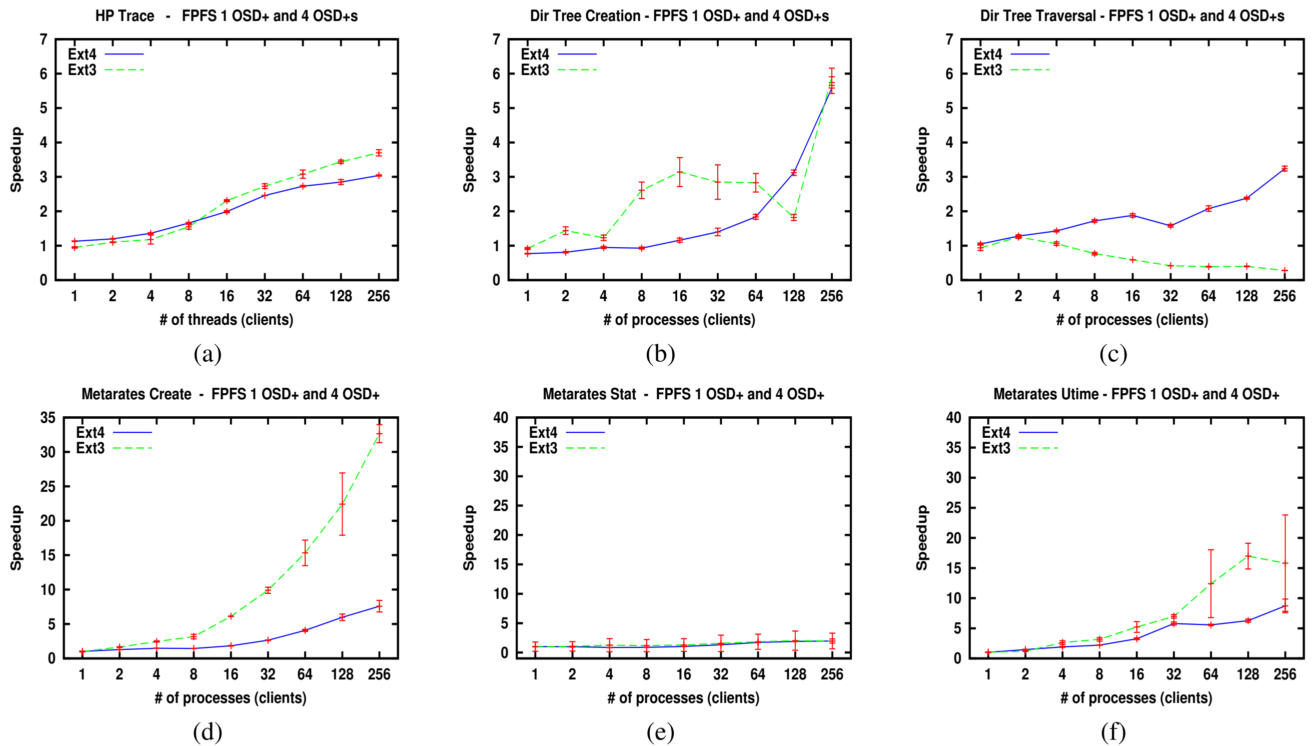

Figure 5. **Scalability for FPFS 1 OSD+ and 4 OSD+s configurations**: (a) HP Trace; (b) Creation of directories; (c) Traversal of directories; (d) Metarates: create-close transactions; (e) Metarates: stat transactions; (f) Metarates: utime transactions.

The scalability of our metadata cluster has been evaluated, and its performance compared with Lustre's. The results show that a metadata cluster with a single OSD+ can improve the throughput of a Lustre metadata server by more than 60–80%, and that it scales with the number of OSD+s.

As a work in progress, we are currently integrating the management of huge directories in the OSD+s.

Acknowledgment

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

References

[1] S. Patil and G. Gibson, "Scale and concurrency of giga+: File system directories with millions of files," in *In Proc. of the 9th USENIX Conf. on File and Storage Technology (FAST'11)*, Feb. 2011, pp. 15–30.

[2] D. Roselli, J. Lorch, and T. Anderson., "A comparison of file system workloads," in *Proc. of the 2000 USENIX Annual Tech. Conf.*, June 2000, pp. 41–54.

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

[4] S. Weil., "Ceph: reliable, scalable, and high-performance distributed storage," Ph.D. dissertation, University of California, Santa Cruz, (CA), Dec. 2007.

[5] 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

[6] P. J. Braams, "High-performance storage architecture and scalable cluster file system," 2008. [Online]. Available: http://wiki.lustre.org/index.php/Lustre_Publications

[7] M. Mesnier, G. R. Ganger, and E. Riedel, "Object-based storage," *IEEE Commun. Magazine*, pp. 84–90, Aug. 2003.

[8] 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 Conf. on Cluster Computing*, 2008, pp. 175–184.

[9] 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 Trans. on Computers*, vol. 39, no. 4, pp. 447–459, 1990.

[10] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith., "Andrew: A distributed personal computing enviroment," *Commun. ACM*, vol. 29, pp. 184–201, March 1986. [Online]. Available: http://doi.acm.org/10.1145/5666.5671

[11] S. A. Brandt, E. L. Miller, D. D. E. Long, and L. Xue., "Efficient metadata management in large distributed storage systems," in *Proc. of the 20 th IEEE/11 th NASA Goddard Conf. on Mass Storage Systems and Technologies*, 2003, pp. 290–298.

[12] J. Wang, D. Feng, F. Wang, , and C. Lu., "MHS: A distributed metadata management strategy," in *The Journal of Systems and Software*, vol. 82, no. 12, July 2009, pp. 2004–2011.

[13] L. Weijia, X. Wei, J. Shu, and W. Zheng., "Dynamic hashing: Adaptive metadata management for petabyte-scale file systems," in *Proc. of the 14th IEEE / 23rd NASA Goddard Conf. on Mass Storage Systems and Technologies*, May 2006, pp. 159–164.

[14] D. Skeen and M. Stonebraker, "A formal model of crash recovery in a distributed system," *IEEE Trans. on Software Engineering*, vol. 9, pp. 219–228, May 1983. [Online]. Available: http://dx.doi.org/10.1109/TSE.1983.236608

[15] INCITS Tech. 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.

[16] Sun-Oracle, "Lustre tunning," http://wiki.lustre.org/manual/LustreManual18_HTML/LustreTuning.html, 2010.

[17] Hewlett–Packard, "Fstrace," http://tesla.hpl.hp.com/opensource/fstrace, 2002.

[18] University Corp. for Atmospheric Research, "metarates," http://www.cisl.ucar.edu/css/software/metarates/, 2004.

[19] C. Morrone, B. Loewe, and T. McLarty, "mdtest HPC Benchmark, http://sourceforge.net/projects/mdtest," 2010. [Online]. Available: http://sourceforge.net/projects/mdtest