Leveraging OSD+ Devices for Implementing a High-Throughput Parallel File System

Journal:	*Concurrency and Computation: Practice and Experience*
Manuscript ID	CPE-17-0574
Editor Selection:	Special Issue Submission
Wiley - Manuscript type:	Special Issue Paper
Date Submitted by the Author:	29-Nov-2017
Complete List of Authors:	Piernas, Juan; Universidad de Murcia, González-Férez, Pilar; Universidad de Murcia
Keywords:	FPFS, OSD+, Data Objects, Lustre, OrangeFS

Received <day> <Month>, <year>; | Revised <day> <Month>, <year>; | Accepted <day> <Month>, <year>

DOI: xxx/xxxx

ARTICLE TYPE

Leveraging OSD+ Devices for Implementing a High-Throughput Parallel File System

Juan Piernas* | Pilar González-Férez

[1]Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia, Murcia, Spain

Correspondence
*Juan Piernas, Facultad de Informática. Campus de Espinardo. 30100 Murcia (Spain). Email: piernas@ditec.um.es

Present Address
Present address

Abstract

OSD+s are enhanced object-based storage devices (OSDs) able to deal with both data and metadata operations via data and directory objects, respectively. So far, we have focused on designing and implementing efficient directory objects in OSD+s. This paper, however, presents our work on also supporting data objects, and describes how the coexistence of both kinds of objects in each OSD+ is profited to efficiently implement data objects and to speed up some common file operations. We compare our OSD+-based Fusion Parallel File System (FPFS) with Lustre and OrangeFS through different microbenchmarks and HPCS-IO scenarios. Results show that FPFS provides a throughput up to 37× better than Lustre, and up to 95× better than OrangeFS, for metadata workloads. FPFS also provides 34% more bandwidth than OrangeFS for data workloads, and competes with Lustre in data writes. Results also show serious scalability problems in Lustre and OrangeFS that limit their performance.

KEYWORDS:
FPFS; OSD+; Data objets; Lustre; OrangeFS

1 | INTRODUCTION

File systems for HPC environment have traditionally used a cluster of data servers for achieving different goals: high rates in read and write operations, fault tolerance, scalability, etc. However, due to a growing number of files, and an increasing use of huge directories with millions or billions of entries accessed by thousands of clients at the same time (1, 2, 3), some of these file systems also utilize a cluster of specialized metadata servers (4, 5, 6) and have recently added support for distributed directories (5, 7).

Unlike those file systems, that have separate data and metadata clusters, our in-house Fusion Parallel File System (FPFS) uses a single cluster of *object-based storage device*+ (OSD+) (8) to implement those clusters. OSD+s are improved object-based storage devices (OSDs) that, in addition to handle data objects as traditional OSDs do, can also manage directory objects. Directory objects are a new type of object able to store file names and attributes, and support metadata-related operations. By using these OSD+ devices, an FPFS metadata cluster is as large as its corresponding data cluster, and metadata is effectively distributed among as many nodes as OSD+s comprising the system. OSD+s are implemented through a thin software layer on top of existing mainstream computers, leveraging many features of the underlying file system. Thanks to this approach, OSD+s add a small overhead, and provide a high throughput (8). FPFS also supports huge directories by dynamically distributing them among several OSD+s (9). The OSD+s storing a distributed huge directory work independently of each other, thereby improving the performance and scalability of the file system.

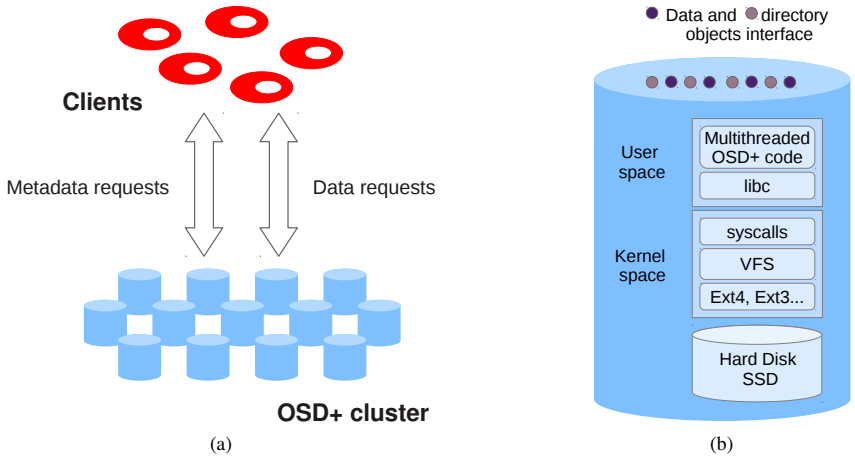


FIGURE 1 (a) FPFS's overview. Each OSD+ supports both data and metadata operations. (b) Layers implementing an OSD+ device.

So far, we have focused on the development of the metadata part of FPFS. In this paper, however, we describe how we have implemented the support for data objects. We will show that the utilization of a unified data and metadata server (i.e., an OSD+ device) provides FPFS with a *competitive advantage* with respect to other file systems that allows it to speed up some file operations, such as creating and deleting files, getting the status of files, etc.

We have evaluated the performance and scalability of this new version of FPFS with data-object support through different microbenchmarks and HPCS-IO scenarios (10), and compared the results with that obtained by OrangeFS (5) and Lustre (7), which only recently have added stable support for distributed directories (and, in the case of Lustre, for a metadata cluster too). Results show that, for metadata-intensive workloads, FPFS provides a throughput that is, at least, one order of magnitude better than that of OrangeFS and Lustre, reaching up to 95× and 37× more operations per second than those achieved by OrangeFS and Lustre, respectively. For workloads with large files and large data transfers, FPFS can obtain a bandwidth up to 34% better than the bandwidth achieved by OrangeFS, and can compete with Lustre in data writes. Interestingly, results have also spotted some scalability problems of OrangeFS and Lustre that severely affect their performance in metadata workloads.

2 | OVERVIEW OF FPFS

Generally, parallel file systems have three main components: clients, data servers and metadata servers. Data servers are usually OSD or OSD-alike devices that export an object interface. Metadata servers, however, frequently implement customized interfaces, and permanently store metadata in private storage devices (7) or in objects allocated in the data servers (6).

Unlike these file systems, FPFS (8) uses a single kind of servers that act as both data and metadata servers (see Figure 1 .(a)). This approach consequently enlarges the metadata cluster's capacity that becomes as large as the data cluster's. To merge data and metadata servers into a single one, FPFS uses OSD+ devices. OSD+s are enhanced OSD devices capable of managing not only data (as regular OSDs do), but also metadata. OSD+ devices simplify the complexity of the storage system as well, since no difference between two types of servers is made. Moreover, having a single cluster increases system's throughput and scalability, since all the servers will be used for data and metadata operations, and there will not be underutilized data or metadata servers. Note that, for instance, in a file system with separate data and metadata clusters, data servers will be idle during metadata-only workloads; this does not happen in FPFS.

2.1 | OSD+

Traditional OSDs deal with *data objects* that support operations like creating and removing objects, and reading from and writing to a specific position in an object. Our design extends this interface to define *directory objects*, capable of managing directories. As a consequence, OSD+ devices support metadata-related operations like creating and removing directories and files, getting

entries for listing a directory, etc. In addition to the usual operations on directories, OSD+s also provide functions to internally deal with operations that may involve the collaboration of several OSD+s (e.g., renames, directory permission changes, etc.).

Currently, there exist no commodity OSD-based disks, so mainstream computers exporting an OSD-based interface are usually used (11)[1]. Internally, a local file system stores the objects; we take advantage of this by *directly mapping* operations in FPFS to operations in the local file system.

Each OSD+ is composed of a user-space multithreaded process and a conventional file system. The process uses the file system as storage backend, and issues Linux syscalls to perform operations on that file system. The local file system must be POSIX-compliant and support extended attributes, since they are used by our implementation. Figure 1 .(b) shows the layers that compose an OSD+. FPFS clients also run in user-space and access OSD+s through a library. This approach is similar to that used by PVFS2/OrangeFS (5).

2.2 | Namespace Distribution

FPFS distributes directory objects (and so the file-system namespace) across the metadata cluster for making metadata operations scalable with the number of OSD+s, and for providing a high performance metadata service. For the distribution, FPFS uses the deterministic pseudo-random function CRUSH (6):

$$oid = CRUSH(hash(dirfullpath)). \quad (1)$$

CRUSH receives the hash of a directory's full pathname as input, and returns the ID of the OSD+ containing the corresponding directory object as output. This allows clients to directly access any directory *without performing a path resolution*.

Hash partition strategies present different scalability problems on cluster resizings, permission changes, and renames. FPFS addresses the first problem through CRUSH, which minimizes migrations and imbalances when adding and removing devices. FPFS manages renames and permission changes via lazy techniques (14); fortunately, these operations are infrequent for directories (14), so they will not impact the overall performance.

2.3 | Directory Objects

A directory object is implemented as a regular directory in the local file system of its OSD+. In this way, any directory-object operation is directly translated to a regular directory operation. The full pathname of the directory supporting a directory object is the same as that of its corresponding directory in FPFS. Therefore, the directory hierarchy of FPFS is imported within the OSD+s by partially replicating its global namespace.

Internally, an OSD+ uses three types of directories, differentiated through extended attributes. These directory types can be seen in Figure 2 , which shows how an FPFS's directory hierarchy is mapped to a 4-OSD+ cluster. The first type (attribute **o**) is assigned to directory objects stored in the OSD+, i.e., objects that CRUSH and their full pathnames have assigned to the OSD+. The second type (attribute **h**) refers to empty directories created in a directory object; they represent subdirectories and allow FPFS to preserve the complete filesystem hierarchy to provide standard directory semantics (e.g., scan). The third one (no attribute) is for directories used for supporting the paths of the directories implementing objects.

For each regular file that a directory has, the directory object conceptually stores its attributes, and the number and location of the data objects that store the content of the file. There are two exceptions: size and modification time attributes of the file, which are stored at its data object(s). In our current implementation, these "embedded i-nodes" (15) are i-nodes of empty files in the directory in the local Linux file system supporting the directory object; the number and location of the data objects are also stored in those empty files as extended attributes.

Implementing directory objects by means of regular directories in a local file system has, at least, two important advantages. The first one is that the implementation is simpler and its overhead smaller since most part of the functionality is provided by the underling file system. The second one is that, when a metadata operation is carried out by a single OSD+ (`creat`, `unlink`, etc.), the backend file system itself ensures its atomicity and POSIX semantics. Only for operations like `rename` or `rmdir`, that usually involve two OSD+s, the participating OSD+s need to deal with concurrency and atomicity by themselves through a *three-phase commit protocol* (3PC) (16), without client involvement.

[1]Seagate's Kinetic drives are key/value servers with Ethernet connectivity. They have a limited object-oriented interface that supports a few operations on objects identified by keys. Kinetic drives could be seen as an early implementation of something similar to Gibson's proposal (12), but, due to their limited design, they still need a higher level layer like Swift (13) to carry out basic operations, such as mapping large objects, coordinating race conditions on write operations, etc.

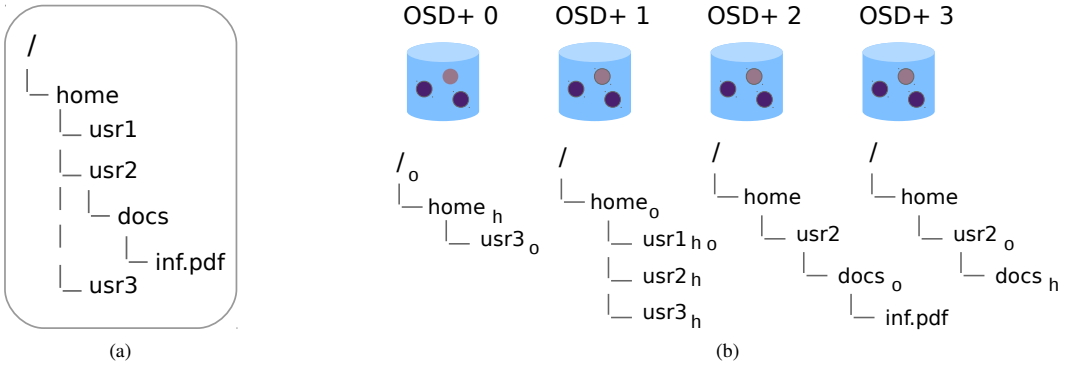


FIGURE 2 (a) An FPFS namespace. (b) A possible mapping of the FPFS namespace to a 4-OSD+ cluster.

2.4 | Huge directories

FPFS also implements management for huge directories, or *hugedirs* for short. These directories are common for some HPC applications, such as those that create a file per thread/process (17), and those that use a directory as a light-weight database (e.g. check pointing) (18). To manage hugedirs, FPFS proposes a dynamic distribution of their entries among multiple OSD+s (9). The subset of OSD+s supporting a hugedir is composed of a *routing OSD+* and a group of *storing OSD+s*. The former contains the *routing directory object* and is in charge of providing clients with the hugedir's distribution information. The latter have the *storing directory objects* that store the directory's content. The *routing OSD+* can also be part of the *storing* group in case it keeps any directory's content. Indeed, for small directories, the routing and storing objects are the same.

A client unaware of the distribution of a directory contacts its routing object by using Equation 1, as it does with any other regular directory object. As reply, the client receives a *distribution list* with the *IDs* of the routing and storing OSD+s. The client caches this information and then retries the operation, but changes the previous directory-level function for a file-level counterpart:

$$oid = osd\_set[(hash(filename)\ \%\ osd\_count], \tag{2}$$

where *osd_set* is the list of storing OSD+s, and *osd_count* is the size of that list. The result is the storing OSD+ having the entry of the given file name.

3 | DATA OBJECTS

Data objects are storage elements able to store information of any kind. They can also have associated attributes that users can set and get. Data objects support different operations, being the reading and writing of data the most important. Data objects receive an ID, which we call *data object ID* (DOID), when they are created. These DOIDs allow us to unequivocally identify an object inside a given device, although there can be duplicated DOIDs among different devices. Therefore, a data object is globally identifiable by means of its DOID and the ID of the device holding it. We call this pair (device ID, data object ID) a *globally unique data object ID* (GUDOID).

In this section, we first explain the relationship between regular files and data objects. Then, we describe how data objects are implemented in FPFS. Finally, we show how the implementation can be optimized thanks to the use of OSD+ devices.

3.1 | Regular files and data objects

As we have seen in Section 2.3, when a regular file is created in FPFS, three related elements are also created: a directory entry, an i-node and a data object. From a conceptual perspective, the i-node is embedded into the directory entry, so these two elements are stored together in the corresponding directory object, while the data object is stored separately (see Figure 3).

FPFS can use two different policies for selecting the OSD+ to store the data object of a file: *same OSD+* and *random OSD+*. The former, used by default, stores a data object in the OSD+ of the directory object storing its file's entry. This approach reduces network traffic because many file operations can be carried out by a single OSD+ without involvement of other OSD+s. The

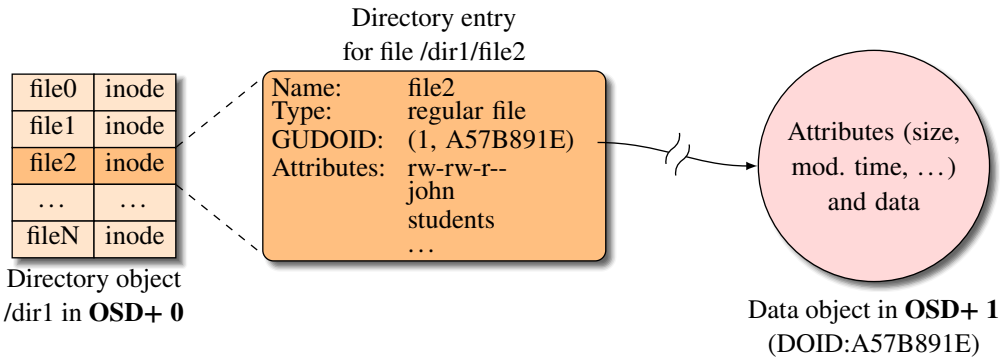


FIGURE 3 A regular file in FPFS. The directory entry contains the i-node and also a reference to the data object.

latter chooses a random OSD+ instead. This second approach can potentially achieve a better use of resources in some cases by keeping a more balanced workload, although it increases network traffic, and this downgrades the throughput of a networked file system, as experimental results will show. Regardless the allocation policy for data objects, the i-node of a regular file will store a reference to its data object by means of its GUDOID.

3.2 | Implementation of data objects

FPFS internally implements data objects as regular files in the underlying local file system used by an OSD+. There exists a data directory where those regular files are stored. This directory is different from that holding directory objects, although they can share the same local file system, and, thus, the same storage device(s).

When a data object is created, its DOID is generated as a random integer number. Although collisions are highly unlikely, if another object exists with that number, a different random number is generated. A string with the hexadecimal representation of the random number is used as name of the local regular file supporting the object. To avoid too large directories, which usually downgrade performance, files are distributed into 256 subdirectories (this number can be increased as needed). A file is located in a directory whose name is made up of the first two hexadecimal characters of the file's name.

An `open()` call on an FPFS file always returns a file descriptor in the OSD+ storing its data object to directly operate on the object. Current implementation supports the following operations on data objects: `read()`, `write()`, `fstat()`, `lseek64()`, and `fsync()`. All of them operate on the object whose descriptor is passed as argument.

The `open()` call also returns a key, which we call *secret*, for the data object, so only those clients that have been granted access to the file can use the returned descriptor to operate on the data object. These clients send the secret along with each request, and the target OSD+ verifies the secret before allowing the operation on the data object.

Note that, while a `stat()` call on a regular file operates on the file's dentry, embedded i-node and data object, a `fstat()` call only operates on the data object whose descriptor is passed to. Therefore, the former call will return all the attributes a traditional `stat()` returns, while the latter will only return size and modification time attributes. Therefore, if a process wants to obtain a "traditional" behavior for `fstat()`, it should combine both operations. Obviously, a library can hide these details, and optionally exports these differences through appropriate functions or flags.

3.3 | Optimizing the implementation

If the default same-OSD+ allocation policy for data objects is active (i.e, a directory entry for a regular file and its corresponding data object are stored in the same OSD+), we can profit the fact that an OSD+ device manages both data and metadata to speed up the creation of files and other operations. When a file is created, the target OSD+ internally creates an empty file in the directory of the local file system supporting the directory object. This empty file usually acts as dentry and embedded i-node, as we have seen in Section 2.3. But because, at the end, data objects are also implemented as files internally, it can also act as data object. Consequently, creation is quite fast, and atomic too: the three elements will either exist or not after the operation. As experimental results show, file systems with separate data and metadata servers (at least, from a conceptual point of view)

incur in a noticeable overhead due to independent operations in different servers, and network traffic generated to perform those operations and guarantee their atomicity. FPFS also bears this overhead if the random-OSD+ policy (see Section 3.1) is used.

The default policy for allocating data objects also has two other effects: (1) distribution of data objects is carried out on a per-directory basis; this increases spatial locality, which can improve the performance in some cases; and (2) if a directory is shared out among several servers, their files and associated data object will be distributed too, thereby improving throughput when accessing files in the directory.

The overlap between a dentry-inode and its data object disappears, however, in a few cases: (a) when a directory object is moved, (b) when a file has several data objects, and (c) for hard links.

First case occurs when a directory object is migrated from an OSD+ to another due to a rename, because only dentries are moved; data objects remain in the server they are, where they arise with their own identity. Fortunately, renames of directories are infrequent, as we have seen in Section 2.2. A similar situation occurs when a directory becomes huge and it is distributed: some directory entries are moved to other directory objects, but their associated objects are not moved. However, realize that, in the case of a hugedir, the dentry-inode-dataobject overlap appears again for new files created after distribution of the directory if the same-OSD+ allocation policy is active.

The second case appears when a file has several data objects, each on a different OSD+ device. In this case, those objects will exist by themselves right from the start. Usually, a file uses more than one data object for improving and scaling its read and write transfer rates, since those operations can be issued to its data objects in parallel. Note, however, that we have not implemented this kind of files yet.

Finally, the third case happens when there exist hard links. For every file having more than one link, FPFS creates an *i-node object*. Internally, this i-node object is the i-node of a file supporting an empty data object. As data object, it has a GUDOID (called *globally unique i-node object ID* or GUIOID in this case), which we can see as a sort of system-wide unique i-node number. This makes the creation of hard links straightforward: the directory entry for a new hard link simply stores the new file name and the GUIOID of the i-node object of the source file. The i-node object for a hard link stores all the file attributes (except size and modification time, as we have explained), references to its data objects, and a counter to track the number of links.

Note that, whether a dentry-inode and a data object are internally supported by the same file or not, an `open()` call always returns a file descriptor to directly operate on the data object.

4 | EXPERIMENTAL RESULTS

We have analyzed FPFS's performance, and compared it with that of OrangeFS 2.9.6 and Lustre 2.9.0. This section describes the experimental environment and results obtained.

4.1 | System under Test and Benchmarks

The testbed system is a cluster made up of 12 compute and 1 frontend nodes. Each node has a Supermicro X7DWT-INF motherboard with two 2.50 GHz Intel Xeon E5420 CPUs, 4 GB of RAM, a system disk with a 64-bit CentOS 7.2 Linux distribution, and a test disk (SSD Intel 520 Series of 240 GB). The test disk supports the OSD+ device for FPFS, and the storage device for OrangeFS and Lustre. Interconnect is a Gigabit network with a D-Link DGS-1248T switch.

We use Ext4 as backend file system for both FPFS and OrangeFS, while Lustre uses its Ext4-based file system. We properly set the I/O scheduler to "noop" for the SSD test disk. We also set formatting and mounting options used by Ext4, to try to obtain maximum throughput with FPFS and OrangeFS. Lustre sets all these parameters automatically, and we do not change them.

We configure the three parallel file systems to shared out directories among all the available servers right from the start. This is because Lustre does not allow a dynamic distribution of directories, and because OrangeFS crashes for relatively small values (< 1000) of its `DistrDirSplitSize` parameter, which specifies the number of entries in a directory before splitting it.

To evaluate the performance, we use the following scenarios of version 1.2.0-rc1 of the HPCS-IO suite (10):

- Scenario 4: there are 64 processes with 10 directories each. Processes create as many files (with sizes between 1 kB and 64 kB) as possible in 50 seconds.
- Scenario 8: there are 128 processes, each creating a file of 32 MB.
- Scenario 9: a single process issues `stat()` operations on empty files in a sequential order.

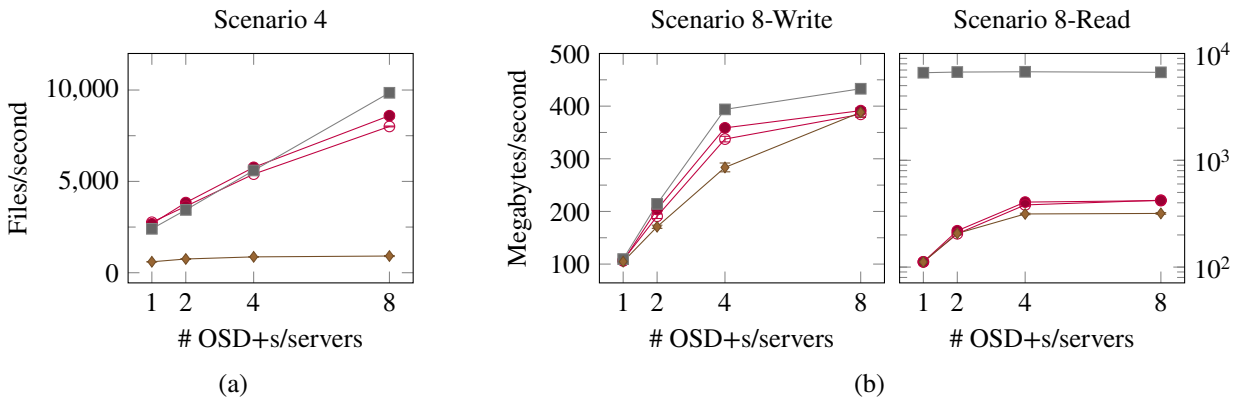


FIGURE 4 HPCS-IO scenarios 4 and 8. Results for FPFS-S (—●—), FPFS-R (—○—), Lustre (—■—), and OrangeFS (—◆—). Note the different Y-axis labels and ranges, and the log scale for the Y-axis in the read test of scenario 8.

- Scenario 10: like scenario 9, but `stat()` operations are issued by 10 processes (this small number of processes is imposed by the scenario).
- Scenario 12: like scenario 10, but operations are issued by 128 processes.

Scenarios 9, 10, and 12 operate on 256 directories, each containing 10 000 empty files, so they use 2 560 000 files altogether. We discard scenarios that involve large or shared files (we do not support multi-dataobject files yet), and scenario 11 (we obtain results identical to those obtained for scenario 9).

Processes in the different tests are shared out among four compute nodes. We have tried to use more than 64/128 processes in some scenarios, but processes start crashing if the file system is OrangeFS. FPFS and Lustre do not have this problem.

Some scenarios of HPCS-IO place synchronization points among client processes, so achieved performance is not as high as it could be. They do not operate on a single directory either, so benefits of distributing hugedirs are not clear. Due to this, we also run the following benchmarks, where there is no synchronization among processes, and a benchmark finishes when the last process completes:

- *Create*: each process creates a subset of empty files in a shared directory.
- *Stat*: each process gets the status of a subset of files in a shared directory.
- *Unlink*: each process deletes a subset of files in a shared directory.

Results shown in the graphs are the average of five runs. Confidence intervals are shown as error bars (95% confidence level). Test disks are formatted before runs of scenarios 4 and 8, and the preprocess for scenarios 9–12 of HPCS-IO. Test disks are also formatted before every run of the create test. For the other benchmarks, disks are unmounted/remounted between tests.

By default, FPFS and Lustre store each file's data in a single server, while OrangeFS shares out a file's content among all the available data servers, with a stripe size of 64 KiB (19). This can affect results achieved by OrangeFS in scenario 8 of HPCS-IO, where processes read and write 32 MiB files. However, in any other case, OrangeFS will use a single server for storing a file's data, much like FPFS and Lustre do.

Note that graphs show the results achieved by two configurations of FPFS: FPFS-S and FPFS-R. The former uses same-OSD+ as allocation policy for data objects, while the latter uses random-OSD+ and, consequently, behaves much like Lustre and OrangeFS. We expect FPFS-R to obtain a smaller throughput than FPFS-S due to the overhead added by independent operations in different servers, and the network traffic generated to perform those operations and guarantee their atomicity, as we have explained in Section 3.3. Remember, however, that same-OSD+ is the default policy. Realize that we use "FPFS" when talking about FPFS-S and FPFS-R without distinction.

4.2 | HPCS-IO

Figure 4 .(a) depicts the results obtained for scenario 4 of HPCS-IO. We see that the performance provided by both FPFS-S and FPFS-R competes with that provided by Lustre, and it is almost one order of magnitude better than that of OrangeFS when

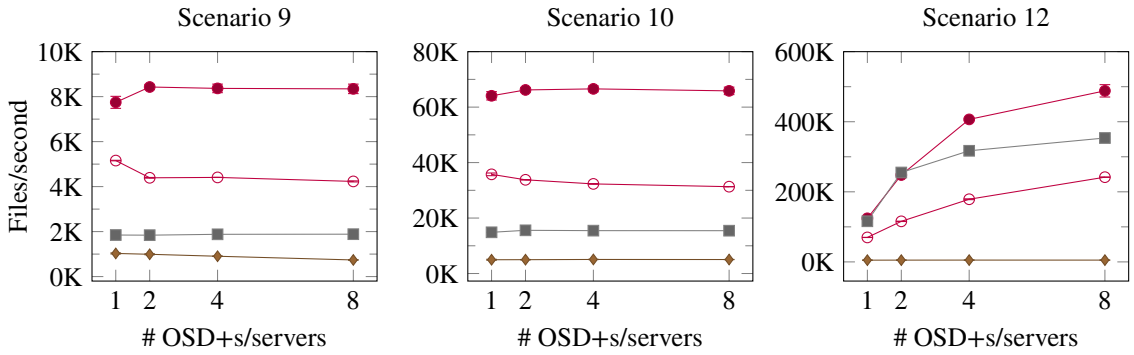


FIGURE 5 HPCS-IO scenarios 9, 10, and 12. Results for FPFS-S (—●—), FPFS-R (—○—), Lustre (—■—), and OrangeFS (—◆—). Note the different Y-axis ranges.

8 servers are used. Moreover, OrangeFS hardly improves its performance by adding servers. Since this scenario creates many small files, with sizes ranging between 1 kB and 64 kBs, there is a large number of metadata operations, but also a greater number of data operations. Consequently, differences between FPFS-S and FPFS-R are small.

Figure 4 .(b) shows results for scenario 8. When there are only a few files and large data transfers, results of each file system depend on its implementation and features. Lustre implements a client-side cache that provides significantly better aggregated read rates (up to 6 741 MB/s in our system). Lustre is also implemented in kernel space and uses the interconnect in a more optimized way, introducing a smaller overhead that allows it to obtain higher aggregated write rates. On the contrary, FPFS and OrangeFS are implemented in user space and do not provide client-side caches. Despite this, FPFS still obtains a higher aggregated bandwidth than OrangeFS: up to 23,5% for writes and 4 servers, and up to 34% for reads and 8 servers. Note that rates hardly increase when the number of servers grows to 8, because network interface cards (NICs) in the clients are saturated with 8 servers. Since this scenario is dominated by data operations, differences between FPFS-S and FPFS-R are nonexistent for reads, and quite small although visible for writes, due to the extra work that the creation of independent data objects represents when the random-OSD+ policy is enforced.

Figure 5 depicts results for HPCS-IO scenarios 9, 10, and 12, where only `stat()` operations are issued on 2 560 000 empty files. In scenario 9, only one process carries out operations, so performance does not increase with the number of servers. FPFS-S achieves around one order of magnitude more operations/s than OrangeFS, and around 4× the throughput achieved by Lustre. FPFS-R roughly obtains half those numbers because each `stat()` operation on a file now represents two operations: one on the embedded i-node in the directory object hosting the file's dentry, and a second one on the data object of the file. The same-OSD+ allocation policy used by FPFS-S allows it to optimize this operation to just a single operation on the dentry-inode-dataobject trio (see Section 3.3). FPFS-S and Lustre provide a steady performance regardless the number of servers, while FPFS-R's and OrangeFS's performances slightly decrease when the number of servers increases. In the case of FPFS-R, this happens because, the greater the number of servers, the smaller the probability of finding the embedded i-node and the data object of a file in the same server. Since the server with a file's embedded i-node is also responsible for fetching size and modification time attributes of the data object of that file, if both elements are in the same OSD+, the fetching operation is carried out by an intra-server communication that is more efficient than that between servers. This is quite noticeable when the number of servers goes from one to two; with one OSD+, all requests on data objects happen inside the OSD+; with two or more servers, a great number of requests to obtain sizes and modification times occur remotely.

In scenario 10, FPFS-S's performance is more than 12× better than OrangeFS's and more than 4× than Lustre's. FPFS-R behaves as in scenario 9 and achieves half the throughput of FPFS-S. All the file systems provide a quite steady throughput in this scenario, regardless the number of servers. The situation changes for FPFS-S, FPFS-R and Lustre in scenario 12, where they greatly improve their performance, which also scales up with the number of servers. OrangeFS, however, does not change its behaviour, and basically provides the same performance as in scenario 10. Due to this, FPFS-S gets more than 95× operations/s than OrangeFS with eight servers. OrangeFS suffers two problems in all these stat-only scenarios. First, bytes and packets per file sent and received by servers increase with the number of servers linearly. Second, there is a server that always handles twice as much network traffic as any other server. Consequently, this server seems to become a bottleneck that also serializes operations, severely downgrading OrangeFS' throughput and impeding its scalability too. Note that Lustre hardly improves its performance

with more than two servers. An analysis of the network traffic reveals that Lustre seems to put different requests for a server in the same message. This "packaging" adds delays that downgrade performance. As in scenarios 9 and 10, FPFS-R gets half the throughput of FPFS-S due to the aforementioned reasons.

4.3 | Single Shared Directory

Figure 6 shows performance achieved, in operations per second, when 256 processes, spread across four compute nodes, concurrently access a single shared huge directory to create, get the status and delete files. Processes work on equally-sized disjoint subsets of files. The directory contains $F \times N$ files, where F is either 200 000, 400 000 or 800 000, and N is the number of servers. By changing the number of files per directory, we can analyze the effect of the directory size on performance and scalability. The directory is distributed right from the start, as we have already explained. Files are uniformly distributed among the servers, which roughly receive the same load.

Graphs show the huge throughput achieved by FPFS-R and, specially, by FPFS-S with respect to the other file systems. With four or more servers (and, in many cases, with just two servers), FPFS-S always gets, at least, one order of magnitude more operations/s than Lustre and OrangeFS. Differences significantly increase with the number of servers, reaching up to 70× more operations/s than OrangeFS, and 37× more than Lustre, in some cases of the unlink test. Throughput of Lustre in the create test is surprisingly small taking into account that Lustre does not create data objects when creating empty files. We have verified this fact by analyzing the network traffic of a Lustre file system with one MGS/MDS and several OST servers; network traffic at OSTs is very small when creating empty files. As we can see too, Lustre is the only file system that does not scale, and its throughput can even decrease with the number of server, as results for create show. These performance differences between both configurations of FPFS and the other file systems can be explained by the network traffic generated by each file system and some serialization problems. We analyze these problems in Section 4.4.

Another remarkable fact is that the directory size can determine the performance in some cases, specially for FPFS-S in the unlink test. This is because Ext4's performance usually drops as the number of directory entries grows. Indeed, the problem in unlink is that, if a directory grows, disk writes increase by a factor much greater than the increase in the number of files. Consequently, FPFS gets a better throughput when there are 200 000 files per object than when there are 800 000. Lustre uses a backend file system that is similar to Ext4. However, Lustre's throughput is so small in the unlink test that the impact of the directory size is not visible. The situation is different in OrangeFS. Since OrangeFS stores directory entries in a few files of a BerkeleyDB, it does not produce huge directories, so results are usually very similar for different directory sizes, and there are only some appreciable differences in the stat test.

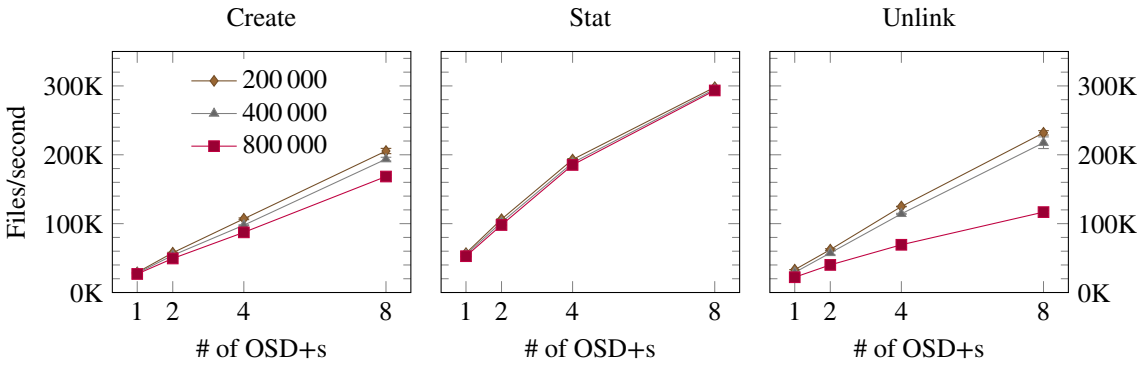
4.4 | Analysis of Network traffic

Due to the noticeable small throughput achieved by Lustre and OrangeFS with respect to FPFS in the create, stat and unlink tests when a single shared directory is used, we have analyzed network traffic at the servers in order to find out a reason that can explain that low performance in those tests.

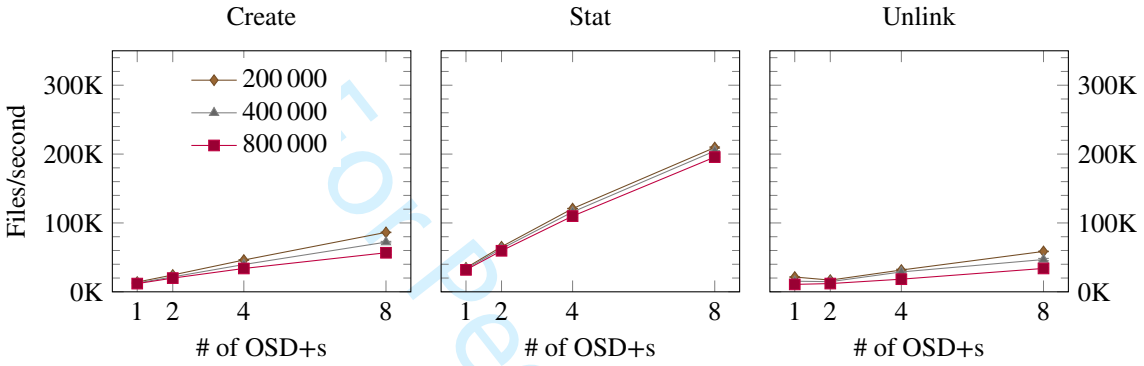
First of all, we have computed how many bytes per file servers receive and transmit. We have divided the total amount of bytes either received or transmitted by the servers by the total amount of files created, stat'ed or unlinked. We have done the same to compute the packets per file received and transmitted by the servers. In both cases, we have used information provided by the `ifconfig` command of Linux. Figure 7 shows graphs for those metrics when there are up to 200 000 files per server. We have also analyzed network traffic when there are up to 400 000 and 800 000 files per server, but it is quite the same as for 200 000 files per server, so we have omitted those results.

As we can see, servers send and receive much more bytes per file with Lustre and OrangeFS than with FPFS-S and FPFS-R, specially in the create test, where, when there are eight servers, servers send and receive, with respect to FPFS-S, more than 26× bytes per file when the file system is Lustre, and more than 17× bytes per file when it is OrangeFS. These numbers are 7× and 5× with respect to FPFS-R.

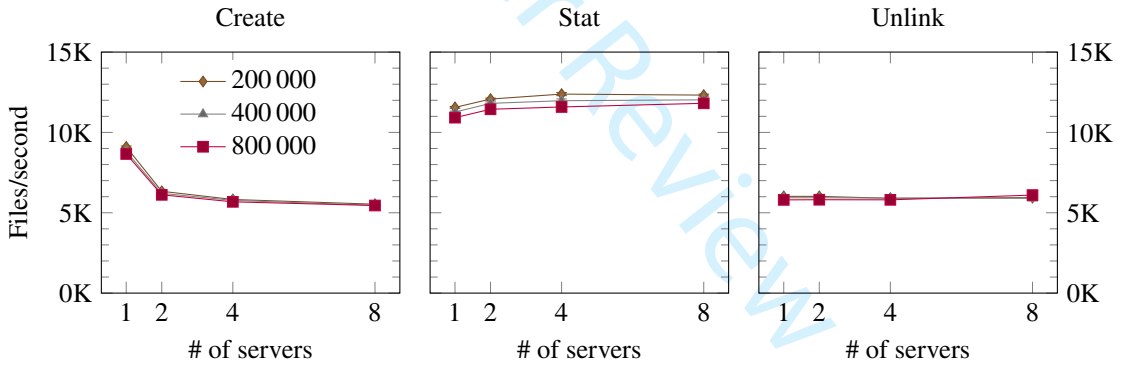
Regarding packets sent and received per file, Figure 7 shows that FPFS-S behaves better than any other file system; now, Lustre and OrangeFS send and receive, respectively, more than 4× and more than 9× packets per file than FPFS-S does. However, FPFS-R roughly sends and receives the same amount of packets per file as Lustre in the create test, while its network traffic falls between those of OrangeFS and Lustre in the stat and unlink tests.



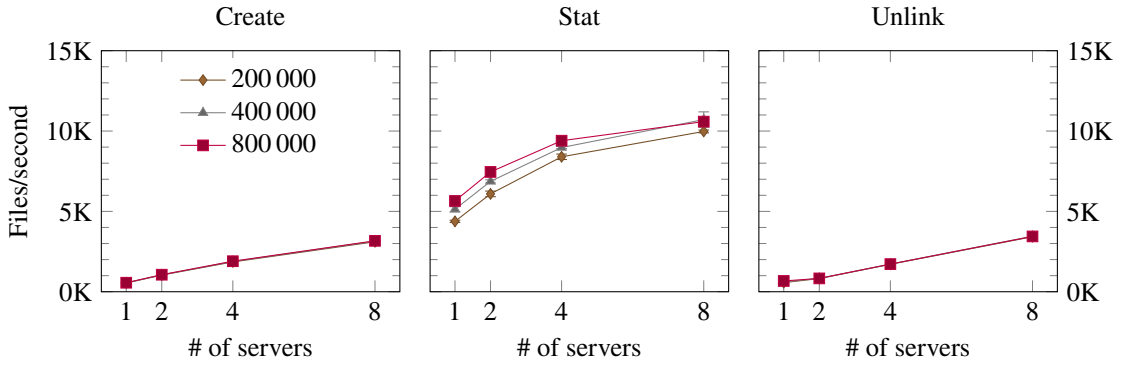
(a) FPFS-S



(b) FPFS-R



(c) Lustre



(d) OrangeFS

FIGURE 6 Shared huge directory. Weak scaling when the number of files per server is 200 000, 400 000 or 800 000. Results for (a) FPFS-S, (b) FPFS-R, (c) Lustre, and (d) OrangeFS. Note the different Y-axis scales among FPFS, FPFS-R, and the other file systems.

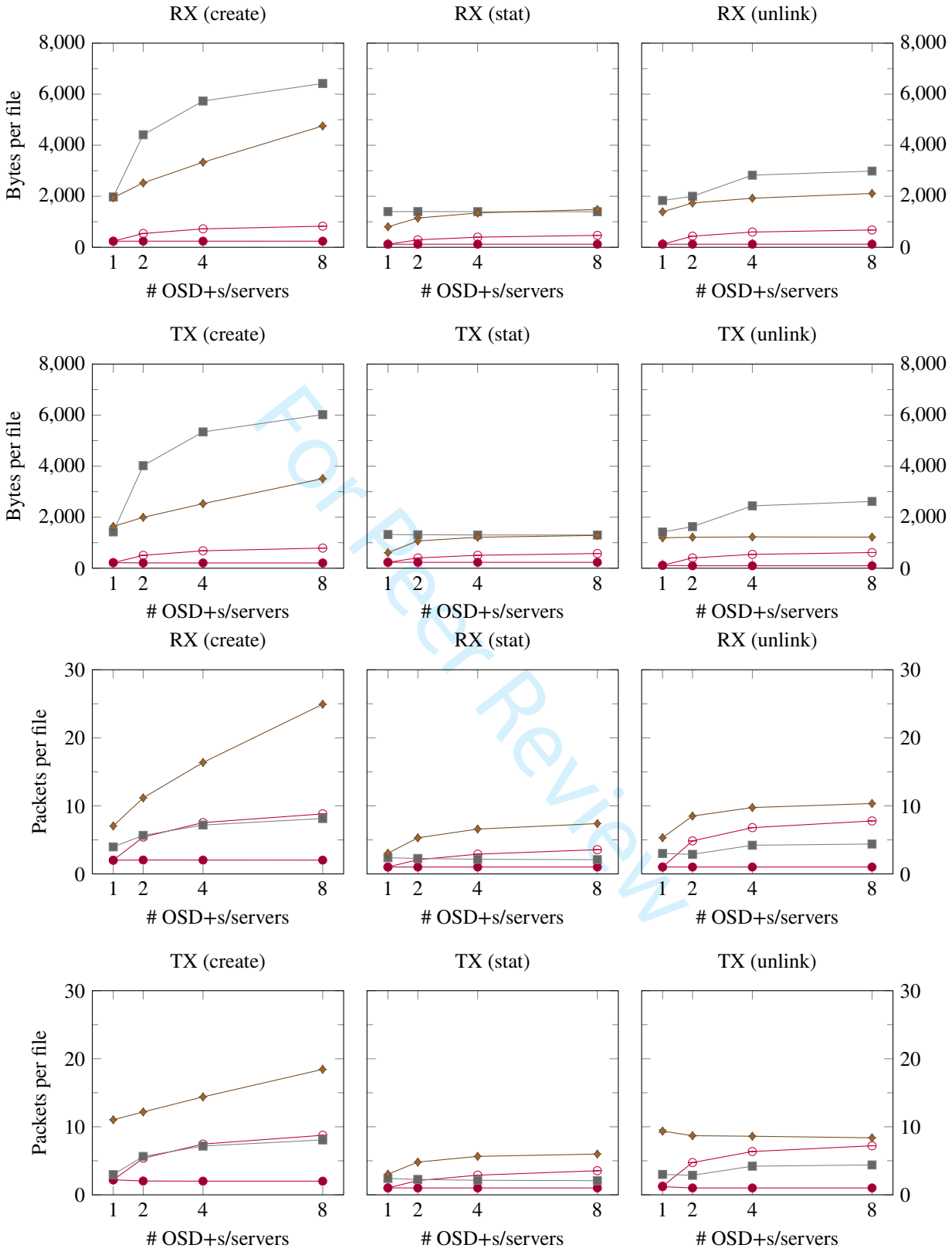


FIGURE 7 Single shared directory: bytes (first six graphs) and packets (last six graphs) per file received (RX) and transmitted (TX) by servers during create, stat and unlink tests. Results for FPFS-S (—●—), FPFS-R (—○—), Lustre (—■—), and OrangeFS (—◆—).

Note that Lustre and FPFS-R increase their network traffic as the number of servers grows, but that network traffic tends to get flat. As we have explained in Section 4.2, the greater the number of servers, the smaller the probability of finding a data object and its related metadata elements in the same server, and, therefore, the higher the network traffic. OrangeFS behaves the same except in the create test, where its network traffic linearly increases with the number of servers; this represents a serious scalability problem for this file system. FPFS-S behaves differently than any other file system, and its network traffic per file does not change with the number of servers.

As we see in Figure 7 , Lustre doubles the bytes and packets sent and received per file in the unlink test with respect to the stat test. Lustre also increases bytes and packets sent and received per file in the unlink test as the number of servers grows, while those bytes and packets per file remain constant in the stat test. This behavior of Lustre regarding network traffic seems to affect its throughput (see Figure 6), because Lustre achieves half the number of operations per file in the unkink than in the stat test, and its performance slightly decreases in the unlink test while it remains constant in the stat test.

A network traffic per file that increases with the number of servers represents a scalability problem for Lustre and, particularly, for OrangeFS. However, this is not the only problem these file systems have. When analyzing network data, we realized that there is always a server that sends and receives much more bytes and packets than the other servers. This problem is specially important for OrangeFS in any test, and for Lustre in the create test. FPFS, however, does not have this issue in any configuration (FPFS-S and FPFS-R), and all its servers manage the same amount of network traffic.

To show this workload-balance problem among the servers, we have identified the servers that send and receive the minimum and maximum amounts of bytes and packets in every test; we have also obtained those minimum and maximum values, and computed average values per server considering the network traffic generated by all the servers. Those values have been normalized to the minimum, so the line representing minimum values is always horizontal with value 1. Results can be seen in Figure 8 for Lustre, and in Figure 9 for OrangeFS. Note that the line representing average values is usually near the line representing minimum values. This means that all the servers but one roughly behave the same. As we have said, FPFS does not present any balance problem, so we have omitted the corresponding graphs because they all would show three overlapping lines.

As Figure 8 shows for the create test, there is a Lustre server that sends up to 17× more bytes than any other server, and almost 8× more packets. Note that scenario 4 of HPCS-IO is somehow similar to the create test, but Lustre does not present any scalability problem in that scenario. The difference is that there are ten directories per process in scenario 4, and those directories are not shared with other processes. Our create test, however, uses a single shared directory, and the use of this type of directories seems to seriously downgrade Lustre's throughput when creating files.

For the stat test, all the Lustre servers roughly send and receive the same amount of bytes and packets. For the unlink test, half of the Lustre servers send and receive more bytes and packets than the other half do (see Figure 8), although differences are quite small. Therefore, Lustre behaves much like FPFS in those tests. Despite that, Lustre's throughput is significantly smaller than that provided by FPFS. Although Lustre servers send and receive more bytes and packets per file than FPFS-S ones (see Figure 7), this does not fully explain that poor throughput of Lustre. We believe that there is a serialization problem in one or more Lustre MDTs when a single shared directory is accessed that prevents them to obtain a higher rate of metadata operations per seconds. Note, for instance, that Lustre achieves more than 353 000 file operations per second in scenario 12 of HPCS-IO, where 128 processes issue stat operations on 256 directories, while it hardly gets more than 12 000 file operations per second in the stat test, where 256 processes issue those stat operations on a single directory.

As we have seen in Figure 6 , throughput achieved by OrangeFS is even smaller than that provided by Lustre, which is already rather small compared with any configuration of FPFS. A possible reason is that OrangeFS servers send and receive much more packets per file than Lustre servers in any test (see Figure 7). Moreover, the amount of packets noticeable increases with the number of servers in the create test. These packets, however, are small, since OrangeFS servers send and receive much less bytes per file than Lustre servers do, as Figure 7 also shows.

Another reason of the poor throughput of OrangeFS is that, unlike FPFS and Lustre, OrangeFS always has a server that sends and receives much more bytes and packets than any other server in any case (see Figure 9). This unbalance even increases with the number of servers. For instance, with 8 servers, the overloaded server sends around four times as many bytes and packets than any other server in the create test. In the stat and unlink tests, this server sends and receives, at least, three times as many bytes and packets than any other server. Clearly, this unbalance means a big scalability problem since there is an overloaded server that becomes a bottleneck.

To summarize, after analyzing network traffic when using a single shared directory, we have seen that Lustre and OrangeFS usually send and receive much more bytes and packets per file than FPFS, and that those values can even increase with the number of servers. Since network traffic is an important limiting factor of networked file systems (9), we can claim that, in this regard,

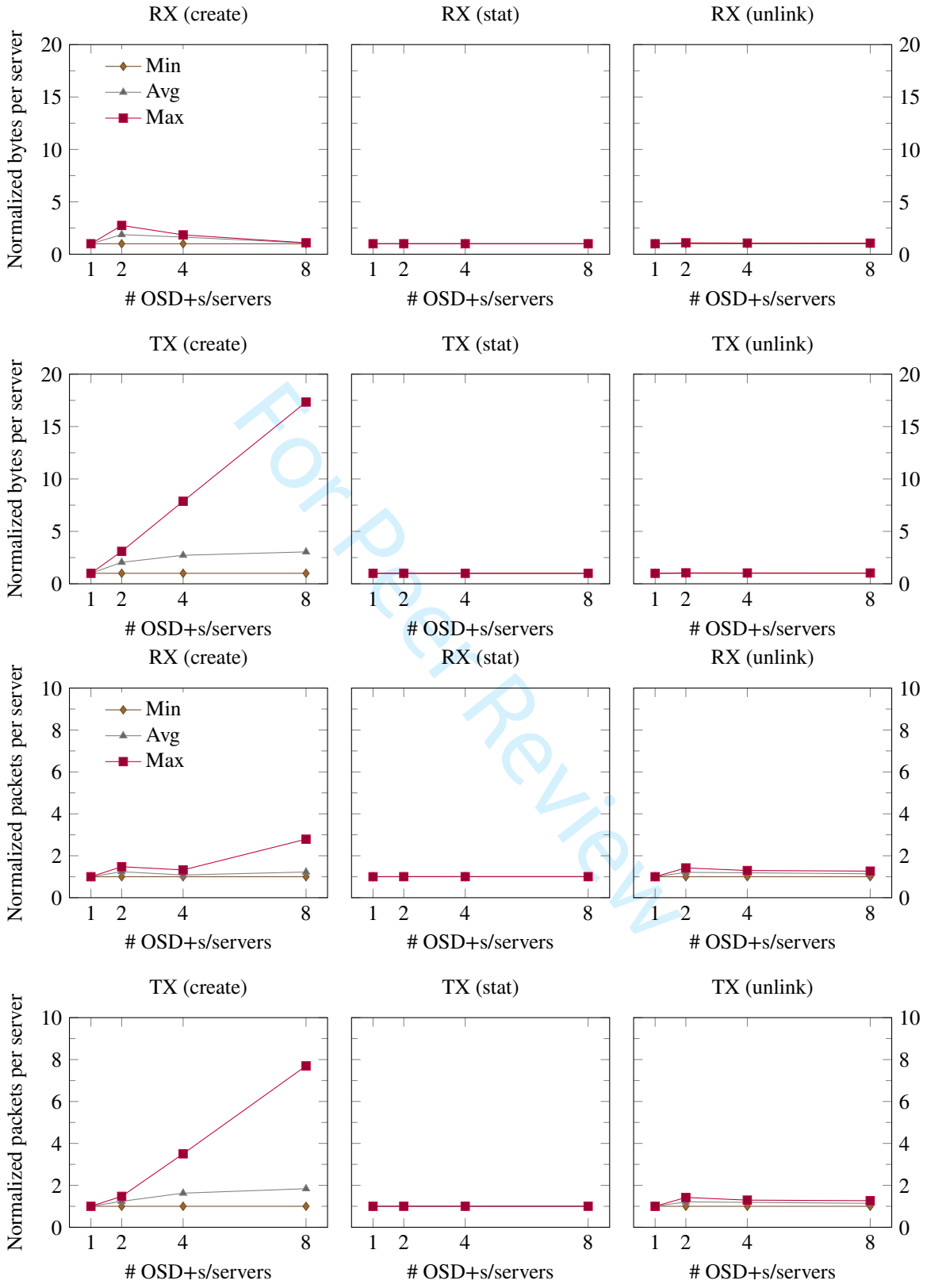


FIGURE 8 Lustre: minimum, average, and maximum bytes (first six graphs) and packets (last six graphs) received (RX) and transmitted (TX) per server. Values have been normalized to the minimum.

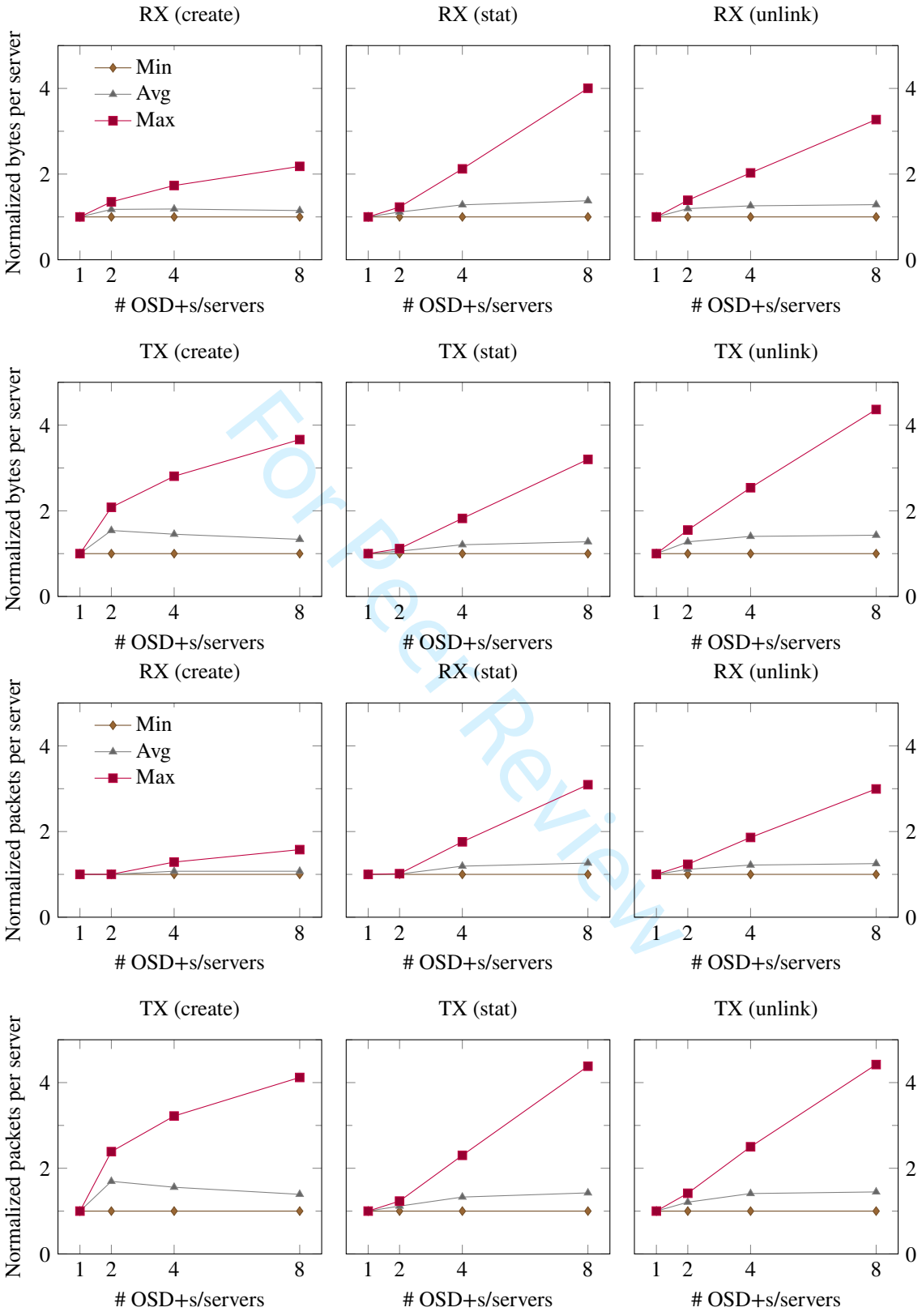


FIGURE 9 OrangeFS: minimum, average, and maximum bytes (first six graphs) and packets (last six graphs) received (RX) and transmitted (TX) per server. Values have been normalized to the minimum.

FPFS-R and, specially, FPFS-S are quite efficient. We have also seen that both Lustre and OrangeFS usually have a single server that handles much more network traffic than any other server, and this overloaded server seems to serialize many operations too. As a consequence of all of these facts, Lustre and OrangeFS present serious scalability problems that limit their performance.

5 | RELATED WORK

There exist many file and storage systems nowadays. This section, however, focuses on some of the existing file systems that implement a metadata cluster, support the distribution of directories, and use OSD or similar devices.

Ceph (6) has a cluster of OSD devices where data objects are stored. OSD devices work in an autonomous manner in order to provide data-object redundancy for fault tolerance, among other things. Ceph uses CRUSH and placement groups to distribute objects and their replicas across the cluster. However, it uses dynamic sub-tree partitioning for the namespace. Contents of directories are written to objects in the OSD cluster, although metadata operations are carried out by a small cluster of metadata servers. Each metadata server controls the popularity of metadata within a directory, and adaptively splits a directory when it gets too big or experiences too many accesses. Despite all these features, the metadata part of Ceph has been unstable for years, with many missing features. At the time of this writing, however, the latest version of Ceph, called Luminous, promises to implement a production-ready metadata cluster.

Like Ceph, OrangeFS (5) provides a cluster of data servers to improve the performance and scalability. These servers are not OSD devices, although they play a similar role. OrangeFS has supported several metadata server for quite a long time, but only recently the distribution of a directory among several servers has been introduced (20). When a directory is created, an array of *dirdata objects* (each on a metadata server) is allocated. Directory entries are then spread across the different dirdata objects, whose number is configurable per directory.

Finally, Lustre (7) also offers a cluster of data servers through OSD devices. Traditionally, Lustre has provided a single metadata server with failover features, but latest versions also allow to use several MDTs in the same file system. A directory can also be shared out among several MDTs, but this distribution is statical, and it is set up when the directory is created.

FPFS shares some important features with all the above file systems: existence of several data and metadata servers, use of OSD or similar devices, data objects, distributed directories, etc. However, design and implementation aspects determine the performance and scalability of all of them. For instance, all but FPFS utilize separate data and metadata services, which makes it difficult when not impossible to optimize some operations that involve both data and metadata elements. OSD+ devices deployed in FPFS also add a small-overhead thin software layer that leverages the underlying local file system to provide their services. Thanks to these devices, FPFS can provide better throughput and scalability than the other parallel file systems in many workloads, specially in those that are metadata-intensive. This is remarkable when compared with Lustre, since the latter provides an in-kernel implementation, while FPFS is implemented in user-space only.

6 | CONCLUSIONS

In this paper, we describe the implementation of data objects in an OSD+ device. We show how OSD+s can internally optimize their implementation to speed up common file operations. This kind of optimizations are not possible in other file systems like Lustre, OrangeFS or Ceph, where data and metadata elements are, from a conceptual point of view, managed independently.

We have added support for data operations to our OSD+-based Fusion Parallel File System, and compared its performance with that achieved by Lustre and OrangeFS. We have also evaluated two configurations of FPFS: FPFS-S, which uses same-OSD+ allocation policy for data objects, and FPFS-R, which uses random-OSD+ as allocation policy. While the former shows the benefits that an optimized implementation of data objects can be achieved by OSD+s, the latter tries to behave like Lustre and OrangeFS, which manage data and metadata elements independently.

Results show that, for metadata-intensive workloads such as creating, stating and deleting files, FPFS-S provides a throughput that is, at least, one order of magnitude better than that achieved by the other file systems, being up to 95× better than OrangeFS's, and 37× better than Lustre's. FPFS-R usually achieves half the throughput of FPFS-S specially due to an increased amount of network packets. For workloads with large data transfers, FPFS-S and FPFS-R can obtain up to 34% more aggregated bandwidth than OrangeFS, while compete with Lustre for data writes. Results also show serious scalability problems in Lustre and OrangeFS that limit their performance. An analysis of network traffic reveals that, at least partially, these problems are due to large amounts of big packets sent and received by servers in Lustre and OrangeFS in many tests, and the existence of a server

that usually handles much more network traffic than other servers do. This overloaded server also seems to serialize operations, so downgrading throughput of these file systems.

As future work, we will evaluate the latest version of Ceph, and compare its throughput with that provided by FPFS. We also plan to add resilience support to both data and directory objects in FPFS.

ACKNOWLEGDEMENTS

Work supported by the Spanish MEC, and European Comission FEDER funds, under grant TIN2015-66972-C5-3-R.

References

[1] Patil S, Gibson G. Scale and Concurrency of GIGA+: File System Directories with Millions of Files. In: Proceeding of the 9th USENIX Conference on File and Storage Technologies (FAST'11):15–30; 2011.

[2] Wang F, Xin Q, Hong B, et al. File System Workload Analysis for Large Scale Scientific Computing Applications. In: Proceedings of the 21st IEEE Conference on Massive Storage Systems and Technologies (MSST'04):139–152; 2004.

[3] Wheeler R. One Billion Files: Scalability Limits in Linux File Systems. In: LinuxCon'10; 2010.

[4] Dilger A. *Lustre Metadata Scaling.* http://storageconference.us/2012/Presentations/T01.Dilger.pdf. Tutorial at the 28th IEEE Conference on Massive Data Storage (MSST'12); 2012.

[5] The PVFS Community . *The Orange File System.* http://orangefs.org; 2017.

[6] Weil S. A, Brandt S. A, Miller E. L, Long D. D. E, Maltzahn C. Ceph: A scalable, high-performance distributed file system. In: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06):307–320; 2006.

[7] OpenSFS , EOFS . *The Lustre File System.* http://www.lustre.org; 2017.

[8] Avilés-González A, Piernas J, González-Férez P. Scalable Metadata Management Through OSD+ Devices. *International Journal of Parallel Programming.* 2014;42(1):4–29.

[9] Avilés-González A, Piernas J, González-Férez P. Batching Operations to Improve the Performance of a Distributed Metadata Service. *The Journal of Supercomputing.* 2016;72(2):654–687.

[10] Cray Inc . *HPCS-IO.* http://sourceforge.net/projects/hpcs-io; 2012.

[11] Ali N, Devulapalli A, Dalessandro D, Wyckoff P, Sadayappan P. An OSD-based approach to managing directory operations in parallel file systems. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC'08):175–184; 2008.

[12] Gibson G. A, Nagle D, Amiri K, et al. A Cost-Effective, High-Bandwidth Storage Architecture. In: Proceedings of the international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98):92–103; 1998.

[13] SwiftStack Inc . *Kinetic Motion With Seagate and OpenStack Swift.* https://swiftstack.com/blog/2013/10/22/kinetic-for-openstack-swift-with-seagate/; 2013.

[14] Brandt S. A, Miller E. L, Long D. D. E, Xue L. Efficient Metadata Management in Large Distributed Storage Systems. In: Proceedings of the 20th IEEE Conference on Mass Storage Systems and Technologies (MSST'03):290–298; 2003.

[15] Ganger G. R, Kaashoek M. F. Embedded inodes and explicit groupings: Exploiting disk bandwidth for small files. In: Proceedings of USENIX Annual Technical Conference (ATC):1–17; 1997.

[16] Skeen D, Stonebraker M. A Formal Model of Crash Recovery in a Distributed System. *IEEE Transactions on Software Engineering.* 1983;9(3):219–228.

[17] Fikes A. Storage Architecture and Challenges. In: Google Faculty Summit 2010; 2010.

[18] Patil S, Ren K, Gibson G. A Case for Scaling HPC Metadata Performance through De-specialization. In: Proceedings of 7th Petascale Data Storage Workshop Supercomputing (PDSW'12):1–6; 2012.

[19] Omnibond Systems LLC . *OranteFS 2.9 Documentation. Concepts Guide.* 2014.

[20] Yang S, Ligon III W. B, Quarles E. C. Scalable Distributed Directory Implementation on Orange File System. In: Proceedings of 7th international Workshop on Storage Network Architecture and Parallel I/Os (SNAPI'11):1–7; 2011.

How cite this article: J. Piernas., P. González-Férez (2017), Leveraging OSD+ Devices for Implementing a High-Throughput Parallel File System, *Concurrency and Computation: Practice and Experience*, *201X;XX:X–XX*.