Efficient Implementation of Data Objects in the OSD+-Based Fusion Parallel File System

Juan Piernas(✉) and Pilar González-Férez

Departamento de Ingeniería y Tecnología de Computadores,
Universidad de Murcia, Murcia, Spain
{piernas,pilar}@ditec.um.es

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 OSD+s 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. Results show that FPFS provides a performance 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 for data writes. Results also show serious scalability problems in Lustre and OrangeFS.

Keywords: FPFS · OSD+ · Data objects · Lustre · OrangeFS

1 Introduction

File systems for HPC environment have traditionally used a cluster of data servers for achieving high rates in read and write operations, for providing fault tolerance and 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 processes at the same time [3,8,12], some of these file systems also utilize a cluster of specialized metadata servers [6,10,11] and have recently added support for distributed directories [7,10].

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+) [1] to implement those clusters. OSD+s are improved OSDs that handle not only data objects (as traditional OSDs do) but also directory objects. Directory objects are a new type of object able to store file names and attributes, and support metadata-related operations. By using OSD+s, an FPFS metadata cluster is as large as its data cluster, and metadata is effectively distributed among all OSD+s comprising the system. Previous results show that OSD+s have a small overhead, and provide a high throughput [1,2].

D. B. Heras and L. Bougé (Eds.): Euro-Par 2017 Workshops, LNCS 10659, pp. 737–747, 2018.
https://doi.org/10.1007/978-3-319-75178-8_59

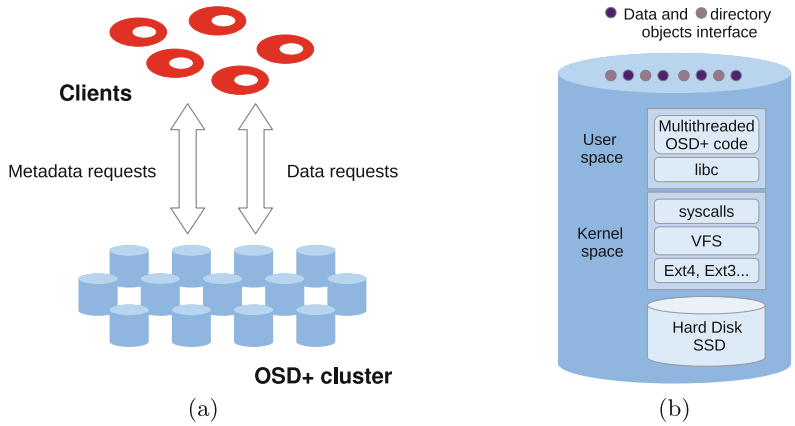

Fig. 1. (a) FPFS's overview. Each OSD+ supports both data and metadata operations. (b) Layers implementing an OSD+ device.

So far, we have focused on the development of the metadata part of FPFS. In this paper, however, we describe how we implement the support for data objects. We 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.

We evaluate the performance and scalability of FPFS with data-object support through different benchmarks, and, for the first time, we compare those results with that obtained by OrangeFS [10] 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. 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

FPFS [1] uses a single kind of server, called OSD+ device, that acts as both data and metadata server (see Fig. 1(a)). This approach consequently enlarges the metadata cluster's capacity that becomes as large as the data cluster's. Moreover, having a single cluster increases system's performance and scalability, since there will not be underutilized data or metadata servers.

Traditional OSDs deal with *data objects* that support operations like creating and removing objects, and reading/writing from/to a specific position in an object. We extend this interface to define *directory objects*, capable of managing

directories. Therefore, OSD+ devices also support metadata-related operations like creating and removing directories and files, getting directory entries, etc.

Since there exist no commodity OSD-based disks (note that Seagate's Kinetic drives are not full-fledged OSD devices), we use mainstream computers for implementing OSD+s (see Fig. 1(b)). Internally, a local file system stores the objects; we profit this by *directly mapping* operations in FPFS to operations in the local file system, thus reducing the overhead introduced by FPFS.

A directory object is implemented as a regular directory in the local file system of its OSD+. 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.

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. The exceptions are size and modification time attributes of the file, which are stored at its data object(s). These "embedded i-nodes" are i-nodes of empty files in the local file system; the number and location of the data objects are also stored in those empty files as extended attributes.

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 [9], without client involvement.

FPFS distributes directory objects (and so the file-system namespace) across the cluster to make metadata operations scalable with the number of OSD+s, and to provide a high performance metadata service. For the distribution, FPFS uses the deterministic pseudo-random function CRUSH [11] that, given a hash of a directory's full pathname, returns the ID of the OSD+ containing the corresponding directory object. This allows clients to directly access any directory *without performing a path resolution*. Thanks to CRUSH, migrations and imbalances when adding and removing devices are minimized. FPFS manages renames and permission changes via lazy techniques [4].

FPFS also implements management for huge directories, or *hugedirs* for short, which are common for some HPC applications [8]. FPFS considers a directory is huge when it stores more than a given number of files. Once this threshold is exceeded, the directory is shared out among several nodes [2].

A hugedir is supported by a *routing OSD+* and a group of *storing OSD+s*. The former is in charge of providing clients with the hugedir's distribution information. The storing OSD+ store the directory's content. A routing object can also be a storing object. The storing objects work independently of each other, thereby improving the performance and scalability of the file system.

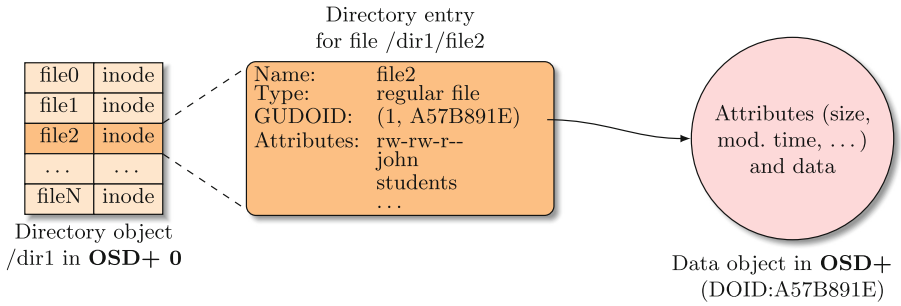

Fig. 2. A regular file in FPFS. The directory entry contains the i-node and also a reference to the data object.

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).

3.1 Data Objects of a Regular File

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. Figure 2 depicts this situation.

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 the network traffic during file creations because no other OSD+s participate in the operation. The latter chooses a random OSD+ instead. This second approach can potentially achieve a better use of resources in some cases by keeping a more balanced workload, although it increases the network traffic during creations. Regardless the allocation policy, the i-node of a regular file stores 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. When a data object is created, its DOID is generated as a random integer number. A string with the

hexadecimal representation of that number is used as name of the regular file supporting the object. To avoid too large directories, which usually downgrade performance, files for data objects are distributed into 256 subdirectories.

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 `read()`, `write()`, `fstat()`, `lseek64()`, and `fsync()` operations. All of them operate on the data object whose descriptor is passed as argument. The `open()` call also returns a key (called *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.

3.3 Optimizing the Implementation

If the default 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 speed up the creation of files and other operations. For instance, when a file is created, the target OSD+ internally creates an empty file in the directory supporting its directory object. This empty file acts as dentry and embedded i-node (see Sect. 2). But because data objects are also implemented as files internally, that empty file 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. 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 the network traffic generated to perform those operations and guarantee their atomicity.

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, or when a directory becomes huge and it is distributed (only dentries are moved, data objects remain in their original servers). 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. Finally, third case happens when there exist files having more than one link. For each of these files, FPFS creates an *i-node object* that also has a GUDOID and stores all the file's attributes (except size and modification time, as explained), references to its data objects, and a link counter. The directory entry for a new hard link simply stores the new file name and the GUDOID of the i-node object of the source file.

4 Experimental Results

We analyze FPFS's performance, and compare it with that of OrangeFS 2.9.6 and Lustre 2.9.0. This section describes experimental environment and results.

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 used by the test disk, and the formatting and mounting options used by Ext4, to try to obtain maximum throughput with FPFS and OrangeFS. Lustre, however, sets 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 OrangeFS crashes for relatively small values (<1000) of its `DistrDirSplitSize` parameter, and because Lustre does not allow a dynamic distribution of directories.

We use the following scenarios of version 1.2.0-rc1 of the HPCS-IO suite [5]:

- 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 s.
- 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.
- 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.

Some scenarios of HPCS-IO place synchronization points among the 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 also shown as error bars (95% confidence level). Test disks are formatted before every run of the 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 rest of the benchmarks, disks are unmounted/remounted between tests.

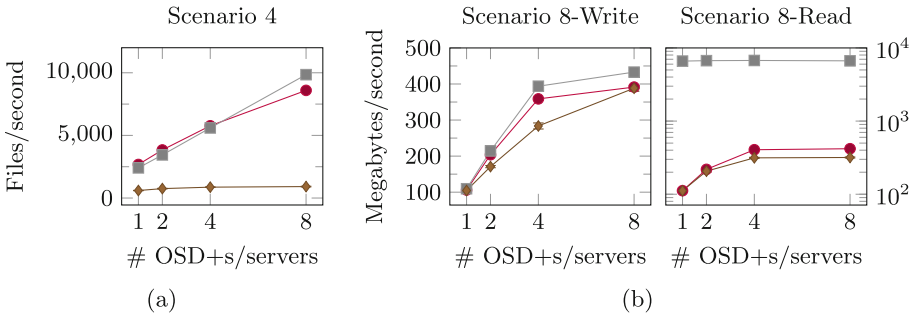

Fig. 3. HPCS-IO scenarios 4 and 8. Results for FPFS (—●—), Lustre (—■—), and OrangeFS (—◆—). Note the different Y-axis labels and ranges, and the log scale for the Y-axis in the read test of scenario 08.

4.2 HPCS-IO

Figure 3(a) depicts the results obtained for scenario 4 of HPCS-IO. We see that the performance provided by FPFS competes with that provided by Lustre, and it is almost one order of magnitude better than that of OrangeFS when 8 servers are used. Moreover, OrangeFS hardly improves its performance by adding servers. Since this scenario creates many small files, we conclude that FPFS and Lustre deals with data and metadata operations much better than OrangeFS.

Figure 3(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. 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 provide no 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.

Figure 4 depicts results for HPCS-IO scenarios 9, 10, and 12, which only issue `stat()` operations 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 achieves around one order of magnitude more operations/s than OrangeFS, and around 4× the throughput achieved by Lustre. FPFS and Lustre provide a steady performance regardless the number of servers, while OrangeFS's performance slightly decreases when there are more servers.

In scenario 10, FPFS's performance is more than 12× better than OrangeFS's and more than 4× than Lustre's. All the file systems provide a quite steady performance, regardless the number of servers. The situation changes for FPFS 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

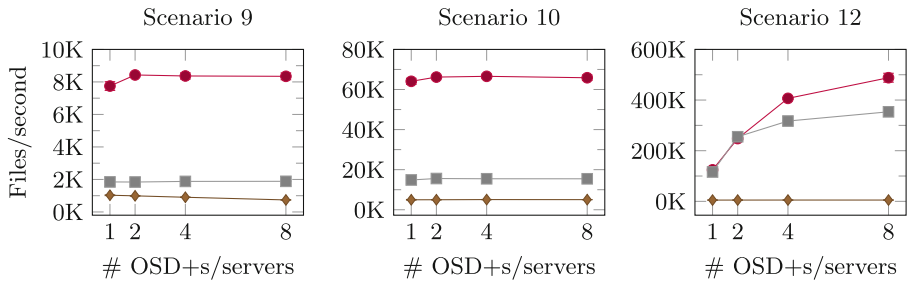

Fig. 4. HPCS-IO scenarios 9, 10, and 12. Results for FPFS (—●—), Lustre (—■—), and OrangeFS (—◆—). Note the different Y-axis ranges.

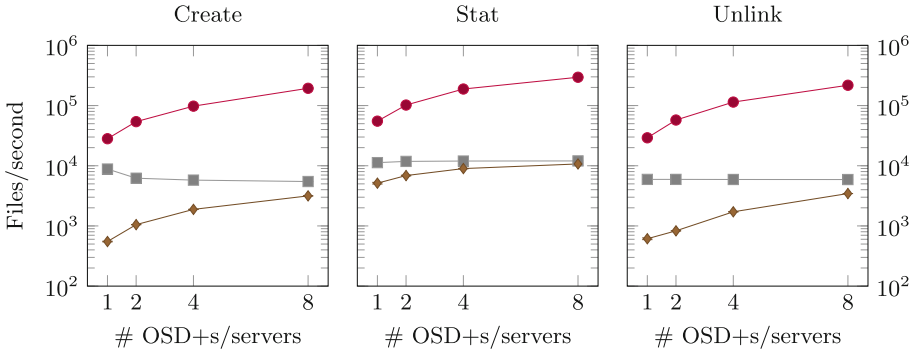

Fig. 5. Shared huge directory. Weak scaling when the number of files per server is set to 400 000. Results for FPFS (—●—), Lustre (—■—), and OrangeFS (—◆—). Note the log scale in the Y-axis.

its behavior, and basically provides the same performance as in scenario 10. Due to this, FPFS gets more than 95× operations/s than OrangeFS. Note that Lustre hardly improves its performance with more than two servers. After analyzing the network traffic, we have seen that Lustre puts different requests for a server in the same message. This "packaging" adds delays that downgrade performance.

4.3 Single Shared Directory

Figure 5 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 400 000 × N files, where N is the number of servers. The directory is distributed right from the start. Files are uniformly distributed among the servers, which roughly receive the same load.

Graphs show the huge performance of FPFS with respect to the other file systems. FPFS always gets, at least, one order of magnitude more operations/s, but it is usually much better (up to 70× more operations/s than OrangeFS, and 37× more than Lustre, in some cases of the unlink test). It is worth noting that, with just 8 OSD+s and a Gigabit interconnect, FPFS is able to create, stat, and delete more than 205 000, 298 000 and 221 000 files per second, respectively.

These performance differences between FPFS and the rest can be explained by the network traffic generated by each file system and some serialization problems. Lustre and OrangeFS generate a high network traffic that even increases with the number of servers. Both also usually have a metadata server that sends and receives much more packets than the rest. Consequently, Lustre and OrangeFS present serious scalability problems that limit their performance.

5 Related Work

This section focuses only on some of the existing file systems that use a metadata cluster, support the distribution of directories, and use OSD or similar devices.

Ceph [11] stores data objects in a cluster of OSD devices that work in an autonomous manner to provide data-object redundancy for fault tolerance, etc. Contents of directories are written to objects in the OSD cluster, and metadata operations are carried out by a small cluster of metadata servers. Each metadata server adaptively splits a directory when it gets too big or experiences too many accesses. Despite all these features, setting a stable metadata cluster in Ceph has been no possible (we still have to test latest releases), so we have discarded this file system in our benchmarks.

OrangeFS [10] also uses a cluster of data servers. They are not OSD devices, but play a similar role. OrangeFS has supported several metadata servers for quite a long time, but only recently has introduced the distribution of a directory among several servers based on ideas from extendible hashing and GIGA+ [8]. 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.

Lustre [7] offers a cluster of data servers through OSD devices. Latest versions of this file system also allow to use several MDTs in the same file system. A directory can also be shared out among several servers, but this distribution is static, 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 OSDs 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 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 software layer that leverages the underlying local file system to provide an efficient service.

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 some 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 add support for data operations to our OSD+-based Fusion Parallel File System, and compare its performance with that achieved by Lustre and OrangeFS. Results show that, for metadata-intensive workloads such as creating, stating and deleting files, FPFS provides a throughput that is, at least, one order of magnitude better than that achieved by the other file systems, and up to 95× better than OrangeFS's, and 37× than Lustre's. For workloads with large data transfers, FPFS can obtain up to 34% more aggregated bandwidth than OrangeFS, while can compete with Lustre for data writes. Results also show serious scalability problems in Lustre and OrangeFS that limit their performance.

Acknowledgements. Work supported by the Spanish MEC, and European Commission FEDER funds, under grants TIN2012-38341-C04-03 and TIN2015-66972-C5-3-R.

References

1. Avilés-González, A., Piernas, J., González-Férez, P.: Scalable metadata management through OSD+ devices. Int. J. Parallel Program. **42**(1), 4–29 (2014)
2. Avilés-González, A., Piernas, J., González-Férez, P.: Batching operations to improve the performance of a distributed metadata service. J. Supercomput. **72**(2), 654–687 (2016)
3. Bent, J., Gibson, G., Grider, G., McClelland, B., Nowoczynski, P., Nunez, J., Polte, M., Wingate, M.: PLFS: a checkpoint filesystem for parallel applications. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC 2009), pp. 1–12 (2009)
4. 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 2003), pp. 290–298 (2003)
5. Cray Inc.: HPCS-IO, October 2012. http://sourceforge.net/projects/hpcs-io
6. Dilger, A.: Lustre metadata scaling, April 2012. http://storageconference.us/2012/Presentations/T01.Dilger.pdf. Tutorial at the 28th IEEE Conference on Massive Data Storage (MSST 2012)
7. OpenSFS, EOFS: The Lustre file system, December 2016. http://www.lustre.org
8. 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 2012), pp. 1–6, November 2012
9. Skeen, D., Stonebraker, M.: A formal model of crash recovery in a distributed system. IEEE Trans. Softw. Eng. **9**(3), 219–228 (1983)
10. The PVFS Community: The Orange file system, October 2016. http://orangefs.org

11. 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 2006), pp. 307–320 (2006)
12. Wheeler, R.: One billion files: scalability limits in Linux file systems. In: LinuxCon 2010, August 2010. http://events.linuxfoundation.org/slides/2010/linuxcon2010_wheeler.pdf