The Journal of Supercomputing manuscript No.
(will be inserted by the editor)

Batching Operations to Improve the Performance of a Distributed Metadata Service

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

Received: date / Accepted: date

Abstract Interconnects can limit the performance achieved by distributed and parallel file systems due to message processing overheads, latencies, low bandwidths and possible congestions. This is specially true for metadata operations, because of the large number of small messages that they usually involve. These problems can be addressed from a hardware approach, with better interconnects, or from a software approach, by means of new designs and implementations. In this paper we take the software approach and propose to increase the rate of metadata operations by sending several operations to a server in a single request. These metadata requests, that we call *batch operations* (or *batchops* for short), are particularly useful for applications that need to create, get the status information of and delete thousands or millions of files. With batchops, performance is increased by saving network delays and round-trips, and by reducing the number of messages, which, in turn, can mitigate possible network congestions. We have implemented batchops in our Fusion Parallel File Systems (FPFS). Results show that batchops can increase the metadata performance of FPFS by between 23% and 100%, depending on the metadata operation and backed file system used. In absolute terms, batchops allow FPFS to create, stat and delete around 200 000, 300 000 and 200 000 files per second, respectively, with just 8 servers and a regular Gigabyte network.

Keywords Batch operations · FPFS · high-performance scalable metadata service · parallel and distributed file systems

Ana Avilés-González, Juan Piernas, Pilar González-Férez
Facultad de Informática
Universidad de Murcia (Spain)
Tel.: +34-868887657
Fax: +34-868884151
E-mail: {ana.aviles, piernas, pilar}@ditec.um.es

1 Introduction

Currently, modern distributed storage systems have to deal with a growing number of files [18,39,51,54], and an increasing use of huge directories with millions or billions of entries accessed by thousands of clients at the same time [2,7,18,39]. To manage both problems (or, at least, the growing number of files), some file systems use a small cluster of specialized metadata servers [13,45,49,53], while others plan to provide a similar service shortly [44].

Our Fusion Parallel File System (FPFS) uses *Object-based Storage Device+* (OSD+) [3] to implement such a metadata cluster. OSD+s are improved Object-based Storage Device (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, like the creation and deletion of regular files and directories. By using these OSD+ devices, an FPFS metadata cluster is as large as its corresponding data cluster, and effectively distributes metadata 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, which leverages many features of the underlying file system. Thanks to this approach, OSD+s have a small overhead, and provide a large performance. Indeed, FPFS is able to create, stat and delete thousands of files per second with a few OSD+ devices [3]. FPFS also supports huge directories by dynamically distributing them among several OSD+s in the cluster. The OSD+s storing a distributed huge directory work independently of each other, thereby improving the performance and scalability of the file system.

Despite its great performance, FPFS shares with other distributed file systems one of their limiting factors: the interconnect. The network latency and the overhead introduced by the processing of messages and packets limit the number of metadata operations per second that a server can dispatch. Interconnect characteristics also affect data operation, but, since applications can issue large data transfers, bandwidth is the main limiting factor here. Therefore, in order to increase the metadata performance, we can use a better interconnect or reduce the network processing overhead.

In this paper, we propose to reduce the processing overhead per message by sending several metadata operations to a server in a single request that we call *batch operation* (or *batchops* for short). Batching is not a new idea, and it has been used extensively in network systems (see Section 6). However, to the best of our knowledge, this is the first time that it is applied in the domain of parallel file systems. Batchops leverage the directory objects of FPFS and embed hundreds to thousands of entries of the same directory (i.e., same directory object) into a single message to perform a given operation on all of them. Applications use batchops through new POSIX-alike functions (`openv`, `statv`, etc.), following the idea that many exascale challenges need to be faced with APIs beyond POSIX [12].

We also show how batchops are integrated in FPFS and, specially, in its distributed huge directories. This kind of directories make the design and implementation of batchops difficult because a regular directory can become distributed during a batch request, and because we should take advantage of the distribution of huge directories in order to efficiently implement batch operations on such directories.

Batchops are possible in FPFS because its namespace is distributed based on directories, which usually contain related files. Therefore, batchops are particularly useful for applications that need to create, get the status information of or delete thousands or millions of entries in the same directory. For instance, applications that use a directory as a light-weight database [42], and operations like `ls -l` or `rm -fr`, can significantly benefit from batchops. But batchops are also useful for parallel file systems that need to migrate or distribute directories (hence, moving a large number of directory entries), as FPFS does. Note, however, that file systems that distribute their namespaces by means of other strategies, such as file hashing [9,29,52,55], make operations like batchops difficult when not impossible.

Batch operations make a much more efficient use of the network, shifting the bottleneck from the network to the servers in many cases. With batchops, FPFS improves its metadata performance by saving network delays and round-trips, and by reducing the number of messages, which, in turn, can mitigate a possible network congestion.

The present work contributes an extensive set of experimental results for batchops on FPFS when using different backend file systems (Ext4 and ReiserFS) and devices in a Linux environment. Specifically, since a metadata service's performance largely depends on the number of IOPS supported by the underlying storage [18], we have compared results obtained by hard disks with those achieved by "seek-free" SSD devices.

Results show that batchops can increase the performance of FPFS by 50% at least when creating a single shared directory, achieving a 100% improvement in some cases. For the `stat` operation, improvement provided by batchops is always around 25%. Finally, when deleting files, the backend file systems determine performance to a large extent, being Ext4 the one that better leverages batchops with an improvement of 60%, while ReiserFS obtains a 23% when using this kind of operations. In absolute terms, batchops allow FPFS to create, stat and delete around 200 000, 300 000 and 200 000 files per second, respectively, with just 8 SSD-OSD+ devices (i.e., OSD+ devices supported by SSD drives) and a regular Gigabit interconnect. Unfortunately, other available parallel file systems, such as Ceph [53], Lustre [36] or OrangeFS [49], do not support batch or bulk operations, so we have not compared FPFS with them.

The rest of the paper is organized as follows. Section 2 describes the FPFS architecture. Section 3 details how batchops are designed and implemented. Results are provided in Sections 4 and 5. Related work is described in Section 6. Finally, Section 7 concludes the paper.

2 Architecture of FPFS

Generally, parallel file systems have three main components: clients, data servers and metadata servers. Data servers are usually OSD [31] or OSD-alike devices that export an object interface. Metadata servers, however, frequently implement customized interfaces, and permanently store metadata in private storage devices [8] or in objects allocated in the data servers themselves [53].

Unlike these file systems, FPFS [3] uses a single kind of server that acts as both a data and a metadata server (see Figure 1), and it consequently enlarges the meta-

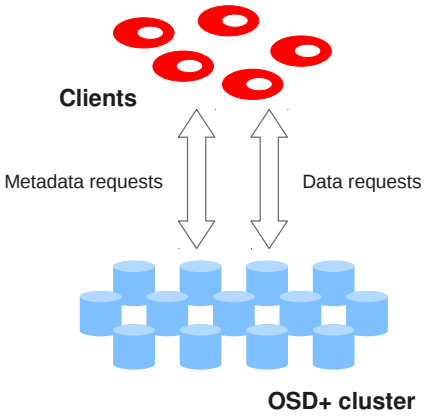


Fig. 1 FPFS's overview. Each OSD+ supports both data and metadata operations.

data 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 that are new enhanced OSD devices. OSD+s are capable of managing not only data (as common OSDs do), but also metadata. These new devices simplify the complexity of the storage system as well, since no difference between two types of servers is made. In addition, having a single cluster increases system's performance and scalability.

2.1 OSD+

OSDs implement not only low-level block allocation functions, but also more complex tasks by taking advantage of their intelligence [41,53]. OSD+s leverage this intelligence too, taking it a step further by delegating metadata management to storage devices.

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. OSD+ devices support metadata-related operations like creating and removing directories and files, or getting directory entries. In addition to the usual operations on directories, OSD+s also provide functions to internally deal with metadata operations that may involve the collaboration of several OSD+s (e.g., renames, directory permission changes, and links).

Currently, there exist no commodity OSD-based disks, so mainstream computers exporting an OSD-based interface through emulators [1], or other software elements, are usually used[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.

[1] Recently, Seagate announced Kinetic [43], a drive that is a key/value server with Ethernet connectivity. It has a limited object-oriented interface that supports a few operations on objects identified by keys. Kinetic could be seen as an early implementation of something similar to Gibson's proposal [20], but, due to its limited design, it still needs a higher level layer like Swift [48] to carry out basic operations, such as mapping large objects, coordinating race conditions on write operations, etc.

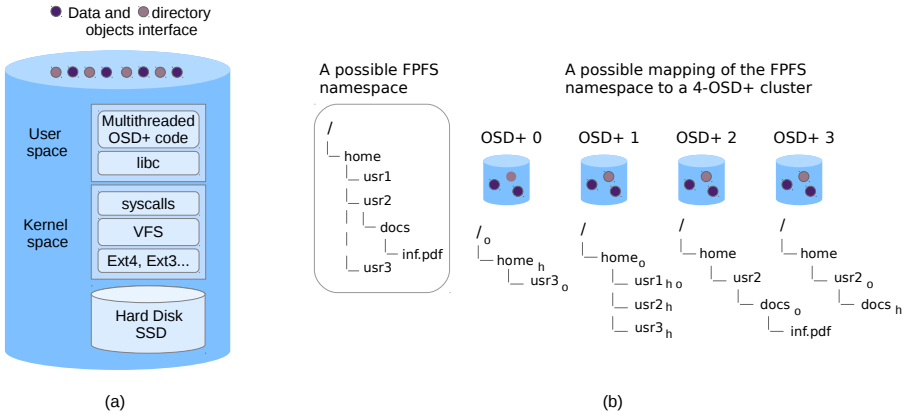


Fig. 2 (a) OSD+ layers. (b) Mapping of an FPFS namespace to a 4-OSD+ cluster.

Each OSD+ is composed of a user-space multithreaded process and a conventional file system. The user-space multithreaded process uses the file system as storage backend. Figure 2.(a) shows the layers that compose an OSD+. The local file system must be POSIX-compliant and support extended attributes (used by our implementation). The Linux syscall interface is used to access the local file system.

2.2 Clients

Clients are the processes accessing FPFS. For fast prototyping and evaluation, the current implementation entirely runs clients in user-space. There exists an FPFS library (`libfpfs`) that clients use to issue requests. This approach is similar to that used by PVFS2/OrangeFS [49] and other file systems [44].

FPFS establishes communications between clients and OSD+s via TCP/IP connections, and request/reply messages. Each OSD+ launches one thread to attend the requests of a client, and to perform operations on the local disk on behalf of the client. When an operation involves several OSD+s, the OSD+ contacted by a client carries out the operation transparently to the client (see Section 2.4).

2.3 Namespace Distribution

FPFS distributes directory objects (and so the file-system namespace) across the metadata cluster to make metadata operations scalable with the number of OSD+s, and to provide a high performance metadata service. For the distribution FPFS uses the deterministic pseudo-random function CRUSH [53]:

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

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*.

We choose CRUSH because it guarantees a probabilistically balanced distribution of objects through the system. However, FPFS does not depend on a particular distribution function, so other functions are also possible [33].

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 [9]. Fortunately, these operations are infrequent for directories [3,9], so they will not impact the overall performance.

2.4 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.(b), 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 stores its attributes, and the number and location of the data objects that store the content of the file. In our current implementation, these "embedded i-nodes" [19] are i-nodes of empty files. The number and location of the data objects of a file are also stored in extended attributes of its associated empty file. The exceptions are the size and modification time attributes of the file, which are stored at its data object(s), so the directory object does not store this information.

Therefore, an FPFS client first contacts the OSD+ storing the directory object of a target file to obtain its data layout. With that information, the client can send read/write operations to the OSD+s storing the corresponding data objects. The same procedure is followed by other parallel file systems.

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) [46], without client involvement.

2.5 Huge directories

FPFS also implements management for huge directories, or *hugedirs* for short. These are directories with millions or billions of entries accessed by thousands of clients at the same time. Hugedirs are common for some HPC applications, such as those that create a file per thread/process [17,35], and those that use a directory as a light-weight database (e.g. check pointing) [40]. Hugedirs can become a bottleneck; therefore, they should be handled properly.

To efficiently manage hugedirs, FPFS proposes a dynamic distribution of its entries among multiple OSD+s [4]. 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. The threshold can also be 0, thereby distributing a directory right from the start. This is useful for directories known to be huge.

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 has the *storing directory objects* that store the directory's content. The storing directory objects are those OSD+s contacted by clients aware of the directory's distribution. The *routing OSD+* can also be part of the *storing* group in case it keeps any directory's content. For small directories, the routing and storing objects are the same; hence, a directory object can play both roles.

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 the *distribution list* with the *ids* of the routing and storing OSD+s. Then, the client 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 objects, and *osd_count* is the size of that list. As index, we use the value returned by a hash applied on the file name. The result is the storing OSD+ having the file entry.

The distribution list of a hugedir is cached by the client accessing the directory. FPFS uses timestamps to detect when this cached information gets out of date due to the rename or deletion of a hugedir. In that case, clients clean up the cached information for the directory and retry the operation following Equation 1. If the directory gets huge again, clients will receive a new distribution list and will change to Equation 2.

3 Batch operations

In this section we describe how FPFS increases the rate of metadata operations by sending several operations to a server in a single request. These new requests, which

we call *batch operations* or *batchops* for short, are particularly useful for applications that concurrently handle thousands or millions of files.

A batchop embeds in a single request hundreds or thousands of entries of a directory to perform a given operation on all of them. A batchop is sent as a single network message. The message for a batchop includes operation type, directory name, list of directory entries, and operation parameters, whereas a regular operation includes operation type, full pathname, and operation parameters. For creation operations, a batchop also includes a semantics that indicates how to act on failures. Our current implementation considers two options: *perform-all-operations* or *stop-on-failure*. The first option tells a server to perform the operation for all the entries regardless of its outcomes. The second option tells a server to stop on the first failed operation. Both decisions are local to a server. Therefore, when a batchop is performed by several OSD+s (see Section 3.1), each one will locally apply the given semantics to the operations it has to carry out.

As an example, Figure 3.(a) shows the format of a regular create operation, while Figure 3.(b) depicts a batch create operation. The directory name is specified separately in a batch operation, since it is the same for all the entries.

Once the message is received on the server's side, the server performs the operation for all the specified entries over the corresponding directory by taking into account the semantics parameter. Operation results are batched as well, and the server sends this information back when all the operations are completed or the first operation fails, depending on the semantics parameter. Therefore, the semantics not only informs servers about how to perform the batchop, but also informs clients about the reply they will receive.

A reply message for a batchop includes three fields (see Figure 4.(b)): operation type, *#errno* and *list of errnos*. *#errno* is the number of performed operations, which is also the number of elements in the list of *errno* values. *list of errnos* contains the returned errno value for each performed operation (actually, the errno is returned as a negative number). The first value in this list corresponds to the operation on the first file in the batch request, the second value to the second file, and so on.

The reply of some batchops contains additional fields. This is the case for `stat`, where an operation returns not only the operation result but also information about the requested files. Therefore, the reply message for a `stat` includes two extra fields: *#succ values* and *list of infos*. *#succ values* is the number of successful operations. *list of infos* contains the stat information for each file, so the length of this list is the same as *#succ values*. Figure 4.(c) depicts the format of this kind of reply messages.

Operations supported by our current implementation of batchops are: `openv`, `closev`, `statv` and `unlinkv`. Their signatures appear in Figure 5. All of them, except for `closev`, follow the message format previously described. In the case of `closev`, instead of a directory name and list of file names, we send a list of open file descriptors. The reply of `closev` follows the same format as the rest, with the lists of successful values and errno values.

As we have said, the interconnect can become a bottleneck on many parallel file systems. By introducing batchops, we significantly reduce the number of messages transmitted between the client and the server and that, in turn, reduces the number of packets transmitted through the TCP/IP stack. For instance, a batchop can create

(a) Regular create operation

CREATE	Fullpathname	Flags	Mode

(b) Batch create operation

CREATE | BATCH	Dirname	#Entries	List of entries	Flags	Mode	Semantics

Fig. 3 Request mesages for create operations.

(a) Regular reply message

Op type	errno

(b) Reply message for batchops

Op type	#errnos	List of errnos

(c) Reply message for a stat batchop

STAT | BATCH | REPLY	#errnos	#succ values	List of errnos	List of infos

Fig. 4 Reply message format for regular operations and batchops.

```
int openv(const char *dirname, char **paths, int flags, mode_t mode,
          semantics_t semantics, int *returnvalues, int count);
int closev(const int *fds, semantics_t semantic, int *returnvalues, int count);
int statv(const char *dirname, char **paths, semantics_t semantics,
          int *returnvalues, struct stat *returnstats, int count);
int unlinkv(const char *dirname, char **paths, semantics_t semantics,
            int *returnvalues, int count);
```

Fig. 5 Signatures of batchops supported by our current implementation of FPFS.

8192 files in a directory with only 2 messages (one request and one reply) instead of 16384 messages (8192 requests and 8192 replies); the number of network packets transmitted will be significantly reduced as well, although it will depends on the size of the batchop messages and the limit impossed by the TCP/IP stack[2]. Therefore, with bathops, we reduce the network traffic, optimize the network bandwidth, and reduce the network overhead due to the processing of packets and messages. Indeed, the improvement achieved by batchops is to a large extent due to the reduction of the network time obtained. Moreover, thanks to batchops, servers receive more work in each batchop message, so they can operate more efficiently, making a better use of caches, disks, etc. in many cases.

[2] The Ethernet protocol limits the maximum payload of a frame to 1500 bytes by default (called Maximum Transfer Unit (MTU)). Consequently, the transport layer limits to 1460 bytes the Maximum Segment Size (MSS), so a message larger than 1460 bytes will be split into several segments to fit this requirement.

Finally, it is important to note that batchops are possible in FPFS because of the way it implements directories. Since every directory corresponds with a directory object (or a few directory objects if it is distributed), and every directory object is stored in a single OSD+, it is easy and makes sense to bundle several related file operations into a single message. Batchops, however, provide little (or none) benefit in other file systems, such as those that distribute the namespace by hashing file names, since every file of a directory can potentially be stored in a different server (hence, each file operation will be issued in a separate message), and files stored in the same server are not probably related.

3.1 Batchops over huge directories

As explained in Section 2.5, FPFS handles huge directories by storing them among a group of OSD+ devices. Therefore, batchops on hugedirs have to be handled differently than on regular directories. In order to exploit the hugedir distribution, clients perform a batchop on a hugedir by sending batch messages in parallel to every storing OSD+ composing the hugedir. Each of those messages contains the directory entries of the original batch message that are stored on the destination OSD+. Once a client receives all servers' replies, it sorts them in the same order in which they were initially requested. Note that an application does not need to know whether a directory is distributed or not in order to issue a batchop to it. The FPFS library (see Section 2.2) used by the application takes care of the distribution, and transparently performs the requests in parallel and reorganizes the replies when a directory is distributed.

We have explained that semantics are local to servers, and this is specially true for the semantics *stop-on-failure* on hugedirs. For these directories, since requests are sent in parallel to different servers, there is no way (at least, not without losing parallelism) of stopping the processing of requests on the servers when an operation fails in one of them. Therefore, the semantics should be necessarily local if we want to improve the performance. This design decision also means that a client has to process the whole *list of errnos* of the batchop reply to verify the return value of each operation.

To clarify the use of batchops on hugedirs, let's look at the example in Figure 6. An application performs an open batch request (`openv`) to open sixteen files on the directory `/home/usr3` (step 1), which is distributed. The FPFS library in the client is already aware of the distribution of the directory and has cached its corresponding distribution list (0 as routing, and 2, 4, 8, 10 as storing OSD+s) (step 2). The library composes four open batchop messages by calculating the storing OSD+ of each file through the distribution list and the distribution function of hugedirs (see Equation 2). Next, client sends in parallel those four batch requests to the storing OSD+s (step 3). Once the servers perform the operations, they send to the client the batchop reply with the list of return values (step 4). In this example, we assume that the creation of `f7` and `f12` files fails. We also assume a *stop-on-failure* semantics, so OSD+ 2 does not create the `f14` file after the failed creation of the `f12` file, and OSD+ 8 does not create the `f10` and `f15` files after the failure on the `f7` file. Finally, the FPFS library in the client sorts the replies in the initial order in which they were requested by

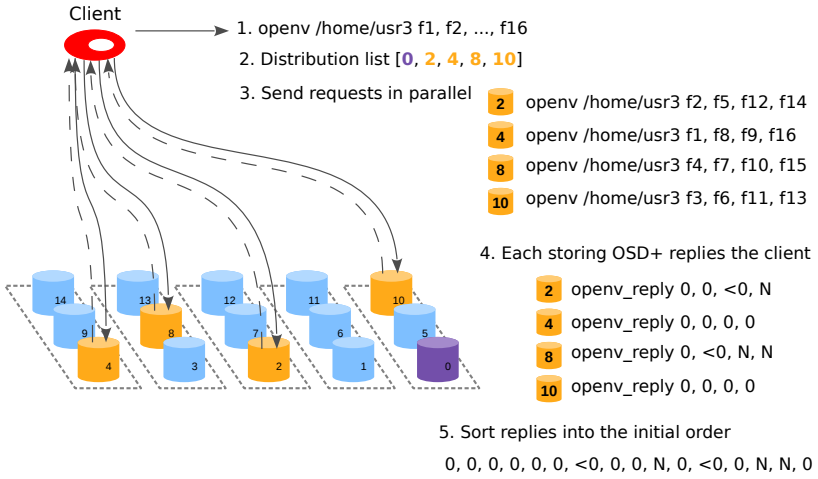


Fig. 6 Example of a client requesting a batch open (`openv`) on a hugedir using semantics *stop-on-failure*. The creation of files `f7` and `f12` fails, so files `f10`, `f14` and `f15` are not created. Reply reflects the result.

using again the distribution list and the distribution function (step 5). For the sake of simplicity, we have used 0 as the return value of a successful operation (it is actually a file descriptor), a negative value (< 0) for a failed operation, and N for an operation that has not been performed due to the semantics.

Our implementation of batchops also considers the case when a regular directory becomes huge during the processing of a batch request, particularly when such a batch request creates hundreds or thousands of files. We manage this situation at the storing OSD+ of the regular directory by processing one operation of the batch request at a time. If the directory gets huge, the processing stops, the OSD+ distributes the hugedir among the storing OSD+s, and a reply containing the results of the already completed operations of the batch request and the distribution list of the now huge directory is sent back to the client that issued the batch request. This client can then continue issuing more batch requests, which will proceed in parallel as we have already described.

4 Experiments and Methodology

In order to analyze the performance of a metadata cluster of FPFS supporting batchops, we have run different benchmarks, and compared FPFS' performance with and without batch operations. This section describes the system under test, the benchmarks run to carry out the analysis, and the objectives that our experiments pursue.

4.1 System under test

The testbed system is a cluster made up of 12 compute and 1 frontend nodes. Technical specifications of each compute node are summarized in Table 1. Test disks support

Table 1 Cluster nodes' technical specifications.

Platform	Supermicro X7DWT-INF
CPU	Two Intel Xeon E5420 quad-core at 2.50 GHz
RAM	4 GB
System disk	Seagate ST3250310NS (250 GB)
Test disks	HDD: Seagate ST3250310NS (250 GB)
	SSD: Intel 520 Series (240GB)
Operating system	64-bit Fedora 11
Interconnect	Gigabit network
Switch	D-Link DGS-1248T

OSD+ devices. We call HDD-OSD+ to an OSD+ device on a hard drive, and SSD-OSD+ to an OSD+ device on a SSD drive.

As I/O scheduler, CFQ is set for HDDs, whereas Noop is set for SSDs. CFQ is the default I/O scheduler in the Linux kernel since 2.6.23. Noop usually achieves the best performance for SSDs compared to the other available Linux schedulers [14,28].

Since metadata performance depends on the backend file system, we use Ext4 and ReiserFS as backend file systems; formatting and mounting options are also important [3]. We format Ext4 file systems with options

```
-J size=400 -i 4096 -I 512 -O dir_index,extents,uninit_groups
```

which set the journal size, bytes-per-inode ratio, i-node size, and use of hashing in directories, extents and some structures uninitiated, respectively. They follow options used by Lustre when formatting its metadata server [47]. In the case of ReiserFS, we use the option

```
--journal-size 32749
```

to set the journal to 32749 blocks (of 4 kB), which is its maximum allowed size when not on a separate device. Mount options are quite similar for both file systems, and try to increase the metadata performance obtained by each one. For Ext4, we use `noatime`, `nodiratime` and `data=writeback`, while we use `notail`, `noatime` and `nodiratime` for ReiserFS. We have not used the `discard` option in Ext4 for issuing `trim` commands to the SSD-OSD+s since ReiserFS does not support this option.

The version of FPFS evaluated in this paper only supports metadata operations, since we focus on improving that kind of operations.

4.2 Benchmarks

To evaluate the performance of batchops in metadata operations, we use the following benchmarks:

- *Create*: each process creates a subset of empty files in either shared or non-shared directories. This benchmark basically generates a write-only metadata workload.
- *Stat*: each process gets the status of a subset of files in shared or non-shared directories. This is a read-only metadata workload (remember that `noatime` and `nodiratime` mount options are used).

– *Unlink*: each process deletes a subset of files in shared or non-shared directories. This is a read-write metadata workload.

Similar tests can be performed by means of well-known benchmarks such as mdtest [34] and even HPCS-IO [11,35]. However, unlike those benchmarks, our tests support batchops of different sizes. We have not considered benchmarks that involve both data and metadata operations since, as aforementioned, we focus on metadata operations only.

4.3 Objectives

Through the experiments, we aim to analyze four different aspects of batchops:

(a) Optimum number of operations per batch operation.
(b) Throughput and scalability for a single shared directory.
(c) Performance when several shared and non-shared hugedirs are accessed in parallel.
(d) Performance when there are one shared and one non-shared hugedir accessed concurrently.

5 Results

The experiments evaluate the performance and scalability of batchops in FPFS considering the objectives described in the previous section. We use HDD-OSD+s and SSD-OSD+s as storage devices, and Ext4 and ReiserFS as backend file systems.

Since it is usual to find many processes running and accessing the storage in an HPC system, we use several clients (up to 256) in our experiments.

We use FPFS in all the test, since, to the best of our knowledge, no other similar file system provides batch operations or equivalent mechanisms. Therefore, a comparison between FPFS and other parallel file systems regarding batchops has not been possible.

Results shown for every system configuration are the average of at least five runs of each benchmark. Confidence intervals are also shown as error bars, for a 95% confidence level. We format the test disks before every run of the create test, and unmount/remount them between tests for the rest.

Before discussing the results, we should mention an issue that has arisen during the experiments. Theoretically, batchops provide some clear benefits: they reduce the number of network messages interchanged and the network overhead, and increase the amount of operations per second sent to servers. Due to this, batchops can reduce the application time, which lessens the chance of a block of being rewritten, and hence, decreases the number of writes to disk issued by the kernel flush daemon. However, while batchops are usually beneficial for SSD-OSD+s, there are some cases where they downgrade the performance when HDD-OSD+ devices are used.

The problem with HDD-OSD+ devices is that more factors influence their performance, and it is not clear how batchops affect the results as a whole in some cases.

For instance, the way files are allocated on disk can affect the performance. When files are created with batchops, a set of i-nodes for the same client can be allocated together on disk. However, if files are created without batchops, i-nodes are more likely to be stored in an interleaved pattern. These two forms of allocation affect performance, mainly because of two factors: head-seeks and disk cache.

We have performed some internal tests (not shown here) to see the behavior of the stat and unlink tests after creating files with and without batchops. Also, we have calculated the number of read and write operations for each of these configurations. In those tests, we have seen that:

- In the case of stat (read-only workloads), having the files created in an interleaved pattern (no batch) obtains better results due to the prefetching performed by different caches. This prefetching allows clients to help each other by bringing to cache i-nodes from other clients. Conversely, when files are created with batch, a client only helps itself in the stat test, reading mainly its i-nodes. The other clients have to read their i-nodes, stored in different disk areas, by themselves; this causes larger head seeks, and can also evict from the disk cache blocks that could use other clients in the near future.
- In the case of unlink, both read and write factors affect the test. Here, batchops help some configurations, but significantly downgrade performance in others. Reducing the time of the test by sending more operations to the servers allows us to reduce the number of writes (as in the create test), but we also need to consider the use of caches for reads in this test (as in the stat case). Given all this, we cannot always determine to what extent each factor affects.

Hence, considering the aforementioned findings, we only provide results with HDD-OSD+s for a single shared hugedir (see Section 5.2). For the remaining benchmarks, we only show results with SSD-OSD+ devices, as they always improve HDD-OSD+s' results, and because the behavior of batchops is more homogeneous with SSD-OSD+s. Moreover, results with SSD-OSD+s have an easier explanation given that there are less factors influencing the results (especially, there are no head seeks).

5.1 Size of batch operation

We start measuring the optimum number of operations embedded per batch request. We perform a test where 256 clients create concurrently files in a single directory. When the directory is shared out among several OSD+s (i.e., it is distributed), the clients create $N \times 400\,000$ files altogether, where N is the number of OSD+s. In our experiments, N is either 1, 2, 4 or 8, so the clients end up creating 400 000, 800 000, 1 600 000 or 3 200 000 files in total. Since the directory is uniformly distributed, every OSD+ receives around 400 000 files. When the directory is stored in a single OSD+ (i.e., there is no distribution), the 256 clients also create either 400 000, 800 000, 1 600 000 or 3 200 000 files altogether, making it easy to compare the results when the directory is and is not distributed.

Figures 7 and 8 show the throughput in operations/s when not distributing and dynamically distributing the directory, respectively. The figures show the performance

Table 2 Batch-operation sizes evaluated. Note that 1 operation per batchop is really equivalent to not having batchops.

Size (operations per batchop)	Label in figures
1	NoBa
10	Ba-10
50	Ba-50
100	Ba-100
500	Ba-500
1000	Ba-1000

for different batchop sizes (see Table 2). These tests use SSD-OSD+ devices. Results for HDD-OSD+s are equivalent, although they are not showed here.

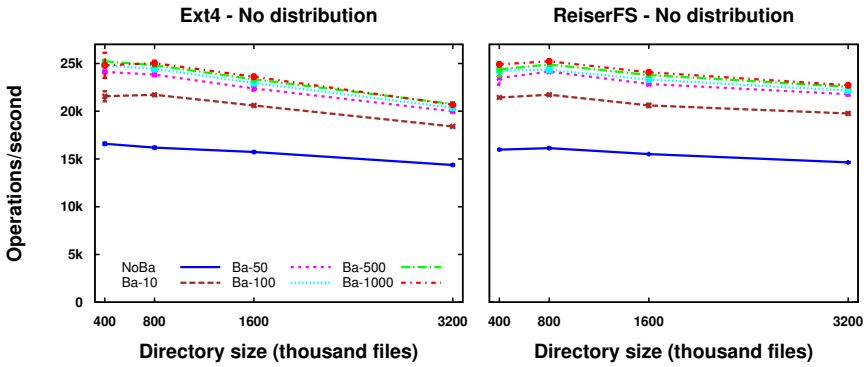
Figure 7 shows that when the shared directory is not distributed, with 1000 operations per batchop we increase the performance between 34% and 73% for Ext4 with respect to no-batch operations, and between 38% and 75% for ReiserFS, depending on the test and number of OSD+s. We also see that we already achieve almost the maximum possible improvement with only 50 operations per batchop, for both Ext4 and ReiserFS, and for any test. Indeed comparing Ba-50 with Ba-1000, although the number of operations included in the batchop is increased by 20×, Ba-1000 only improves performance by up to 21.5% and, on average, only 8.4%. Note that in this test there is a single server receiving concurrent requests from 256 clients. Therefore, the server is saturated and more operations per batch cannot improve the performance further.

Regardless the number of operations per batch, the performance downgrades when the size of the directory increases. This problem is more evident in the unlink test when the backend file system is Ext4. The problem is that Ext4 handles larger directories worse than ReiserFS and, despite SSD-OSD+ devices help avoiding this problem by removing seek latencies, it is still noticeable in this test.

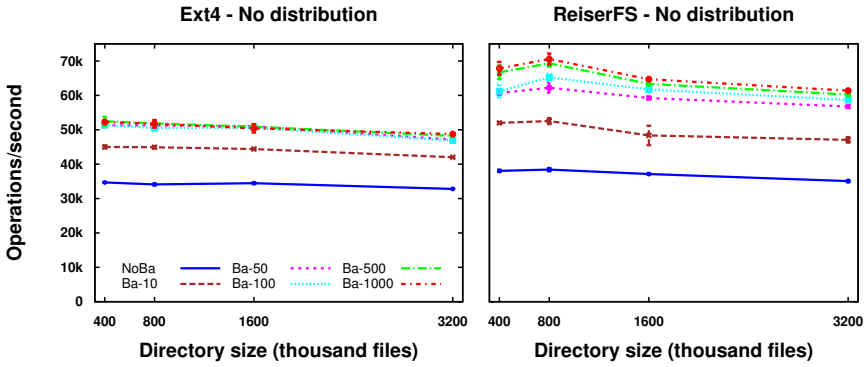
Figure 8 depicts the results when dynamically distributing the directory. We use a dynamic distribution that shares out the directory when it exceeds 8 000 files. Once distributed, the directory object size becomes the same on each OSD+, resulting in a balanced workload. The larger the number of operations per batch, the better the throughput. In general, the largest performance is obtained at 500 or 1000 ops/batch, although, similar to not distributed, Ba-1000 only improves on average 13.0% the performance achieved by Ba-50.

Focusing on 1000 ops/batch, the largest improvements are obtained in the create and stat tests. With Ext4, performance is improved between 39% and 88%, and with ReiserFS, between 46% and 72%. With both file systems and only 8 SSD-OSD+s, and thanks to batchops, we can produce more than 200 000 creates/s and 350 000 stats/s.

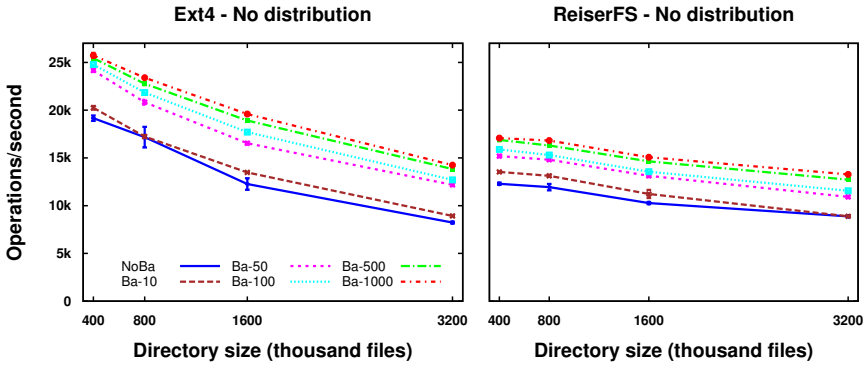
In the unlink test, batchops also improve performance significantly. With Ext4, we gain between 25% and 55%. With ReiserFS, we obtain improvements between 21% and 23%. In the case of Ext4, thanks to batchops, we reach 200 000 unlinks/s with just 8 SSD-OSD+ devices. There is, however, an odd behavior of Ba-10 on Ext4. As aforementioned, although batchops always reduce the network time, there can appear side effects that can improve the performance even more, or downgrade the performance despite the reduction in network time, specially for small batchops.



(a) create

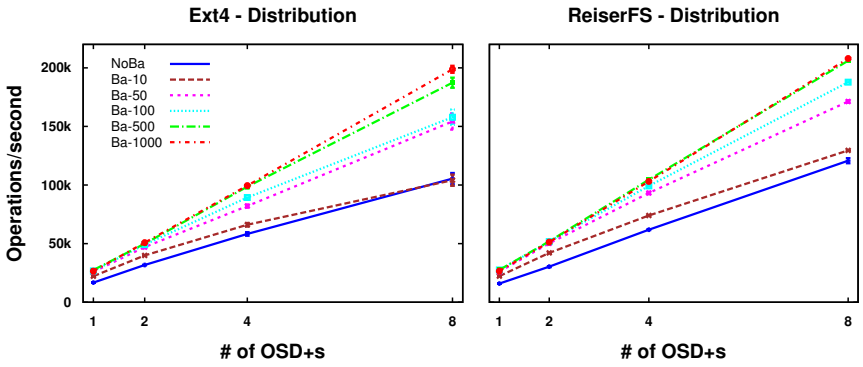


(b) stat

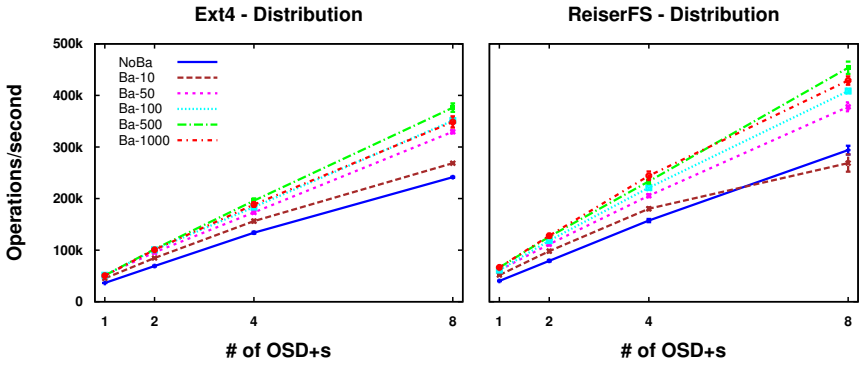


(c) unlink

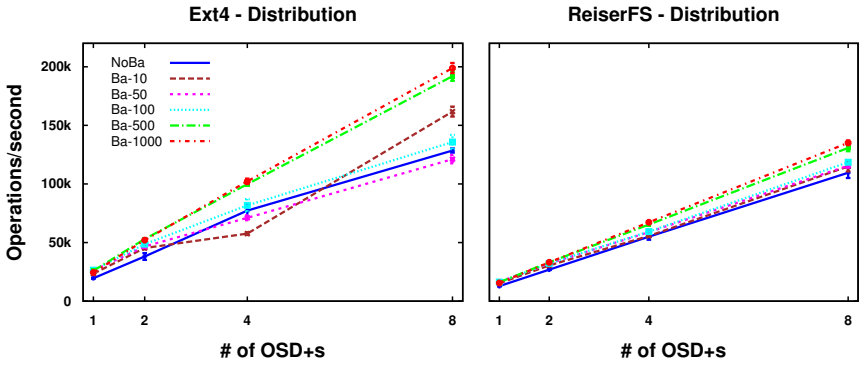
Fig. 7 Operations per second obtained by FPFS on SSD-OSD+s with different number of operations embedded in a batchop, when 256 clients create, stat or unlink files on one *non-distributed* shared directory. NoBa means that batching is not applied. Graphs on the left show results for Ext4 and those on the right for ReiserFS. Note that the range of the Y axis can change from one test to another.



(a) create



(b) stat



(c) unlink

Fig. 8 Operations per second obtained by FPFS on SSD-OSD+s with different number of operations embedded in a batchop, when 256 clients create, stat or unlink files on one *distributed* shared directory. NoBa means that batching is not applied. Graphs on the left show results for Ext4 and those on the right for ReiserFS. Each SSD-OSD+ ends up storing 400 000 files. Note that the range of the Y axis can change from one test to another.

This seems to be the case for Ba-10, which reduces the amount of bytes written by 10% with respect to NoBa when there are 8 OSD+s, but increases that amount by 8.6% when using 4 OSD+s, thereby eliminating the improvement that the reduction in the network time could provide to the overall time. We have not found a satisfactory explanation for this different behavior of Ba-10 yet.

To summarize, given these results, particularly when the directory is distributed, embedding 500 or 1000 operations per batch request seems a good option. Larger requests, although possible, would provide little benefit, since the improvement achieved when going from 500 to 1000 is usually small already, and even negative in some cases. Indeed, Ba-1000 includes 2× more opertions per batchop than Ba-500, but it improves only by up to 6.1% the performance provided by Ba-500.

5.2 Single shared directory

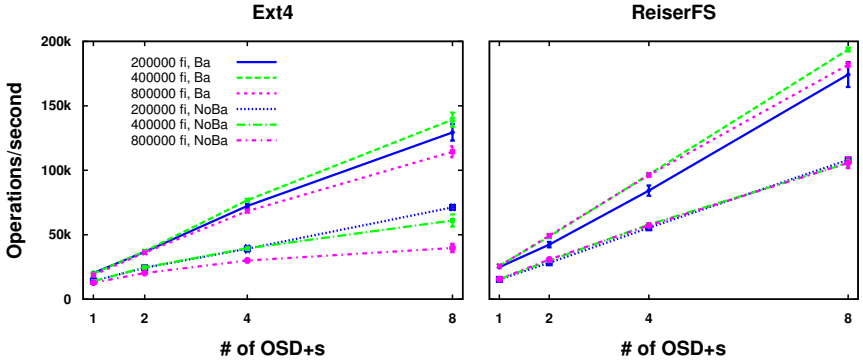
Now, we compare the performance and scalability of batch and regular operations on a single distributed shared huge directory. In this test, a hugedir is accessed by 256 clients at the same time to create, get the status of and delete files. In addition, we evaluate the effect of the directory size by creating $F \times N$ files in the directory, where F is either 200 000, 400 000 or 800 000, and N is the number of OSD+s. For instance, when having 8 OSD+s, the directory has 1 600 000, 3 200 000 or 6 400 000 files, respectively, that are equally distributed among the 8 OSD+s. Unless otherwise indicated, each batch request includes 1000 file operations.

For Ext4 and ReiserFS, Figures 9 and 11 depict FPFS performance in operations/s obtained with HDD-OSD+ and SSD-OSD+ devices, respectively. Figures 10 and 12 show the speedup achieved for the same devices. In the figures, results are labelled as "N fi, Ba" and "N fi, NoBa", where N corresponds to the final number of files in every directory object (or OSD+, since there is only one directory object per OSD+ in this test), and "Ba" and "NoBa" stand for batching and no batching, respectively.

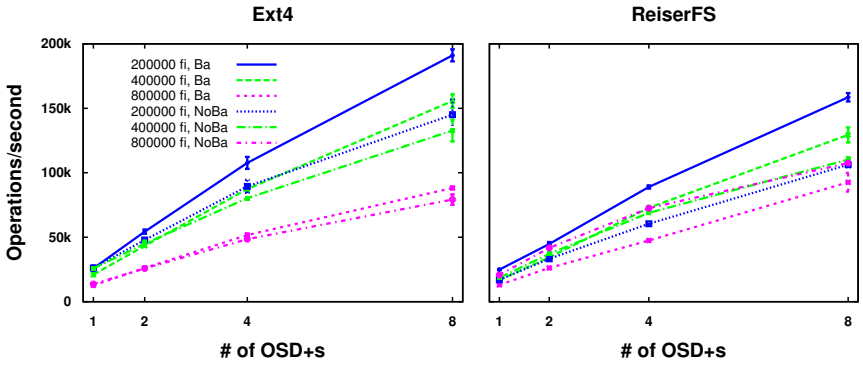
5.2.1 HDD-OSD+

Results for HDD-OSD+s and a single shared hugedir are depicted in Figures 9 and 10. As we advanced in the beginning of the section, with HDD-OSD+s there are more factors involved in the results, and it is not always clear to what extent they affect the different configurations. Moreover, since the behavior and performance observed here are repeated in the other tests carried out with HDD-OSD+, conclusions showed here can be extrapolated to a large extent.

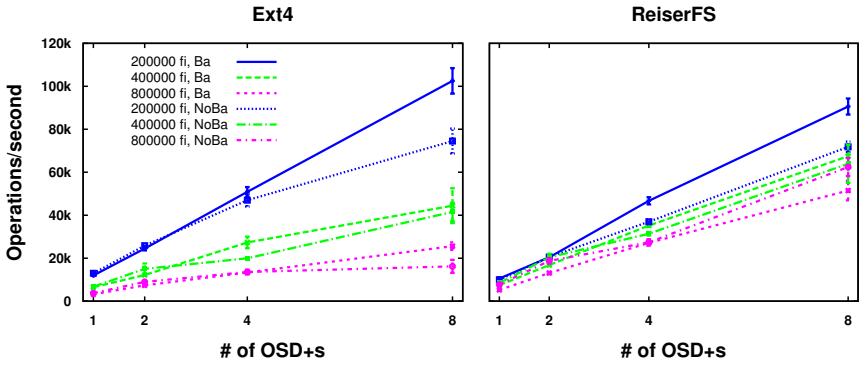
For the create tests, Figure 9.(a) shows that batchops always perform better than no-batch. Namely, with Ext4, batchops improve performance over 50% for 8 OSD+s, whereas with ReiserFS, the improvement of batchops is more than 40%. The configurations with no-batch suffer the network limitation. In the create test, four network messages are generated per file: two (request and reply) for an `open` or `creat` call, and another two for closing the returned file descriptor. This significantly increases the amount of network messages compared to the other tests, and, therefore,



(a) create

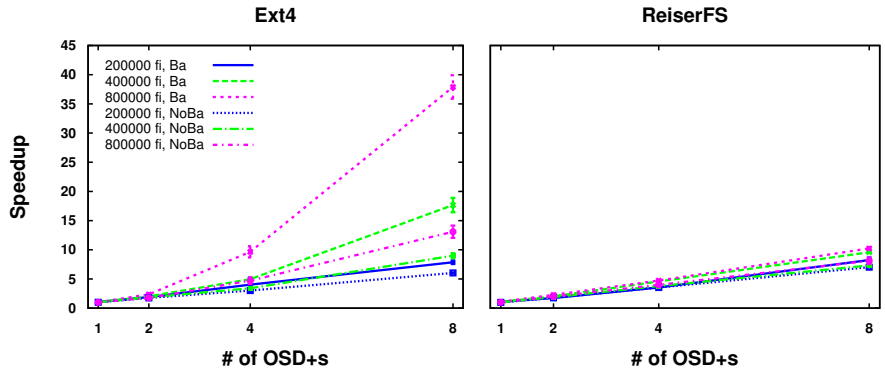


(b) stat

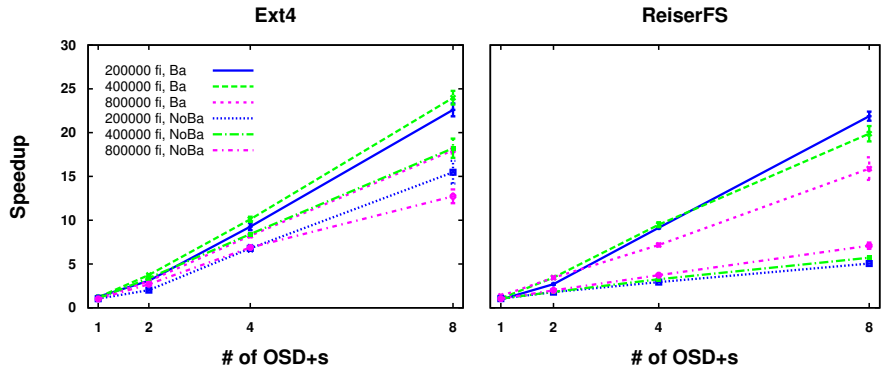


(c) unlink

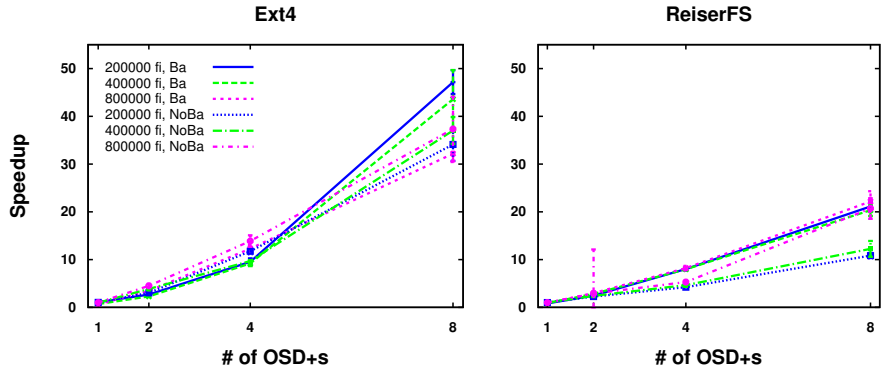
Fig. 9 Operations per second obtained by FPFS with HDD-OSD+s when using one shared hugedir. Graphs on the left show results for Ext4 and those on the right for ReiserFS. Note that the range of the Y axis can change from one test to another.



(a) create



(b) stat



(c) unlink

Fig. 10 Scalability obtained by FPFS with HDD-OSD+s when using one shared hugedir. Graphs on the left show results for Ext4 and those on the right for ReiserFS. Note that the range of the Y axis can change from one test to another.

batchops are more effective. For instance, when creating 200000 files, FPFS interchanges 800000 network messages when regular operations are used, whereas with batchops only 800 messages are interchanged since each batch request includes 1000 operations. Moreover, this benchmark only issues write requests that go to cache, instead of directly accessing the disk. Therefore, batchops perform better than regular operations by sending more requests in each message.

However, for stat and unlink, batchops are not always beneficial, and performance of this test depends not only on the backend file system, but also on how files are created, as we have already explained at the beginning of this Section 5.

For stat and Ext4, batchops improve performance compared to no-batch for small directory objects (200000 files), and for 4 and 8 OSD+s for larger directory objects, where the reduction of network traffic and the higher number of operations per second are more noticeable. For ReiserFS, batchops also improve for directory objects with small number of files, and for 400000 with 4 and 8 OSD+s, but not for 800000 files. In general, as directory objects are larger, batchops performance decreases mainly because of the increase in head seeks, and the poor use of caches compared with no-batch. To understand this fact, we should take into account the way files are created and then read (see the beginning of Section 5). When we use no batch, files are in an interleaved pattern. Each disk block read by a client will probably help other clients because it will probably contain some of their i-nodes. A side effect of this is that clients roughly proceed at the same pace, so disk heads usually move forwards and disk caches are more efficiently used too. When using batchops, each client only helps itself, so the other clients have to issue read requests that produce large head seeks forwards and, what is worse, backwards, incurring in large latencies. This behavior is more noticeable as directory objects grow, and specially in ReiserFS, where batchops perform 60% worse than regular requests.

In the unlink test, we have two different behaviors depending on the file system (see Figure 9.(c)). Ext4 benefits from batchops for 4 and 8 OSD+s in any case, while ReiserFS only benefits for 4 and 8 OSD+s when there are 200000 files and 400000 file per OSD+. As before, performance downgrades as the size of the directory grows. Several factors are intervening here, particularly the type of workload, which mixes reads (similar to those issued by stat) and writes. As we have just seen, both Ext4 and ReiserFS downgrade performance with batchops for some configurations in stat, and this problem also affects reads in this unlink test.

However, when it comes to writes, Ext4 benefits from batchops, since many disk blocks are entirely modified in a short time (e.g., blocks full of i-nodes of files of the same client), and are usually written to disk only once despite the frequent commits in Ext4. This reduces the duration of the test, which in turn also reduces the chance of a block of being modified several times and, therefore, it reduces (again) the amount of writes. Without batchops, a disk block (e.g., a block with i-nodes from different clients) can be modified at different moments, and written to disk several times. This increases the number of writes and head seeks, so the test takes longer.

ReiserFS also reduces the number of write requests, but its performance is significantly determined by its behavior for read requests, as that seen in the stat test. Therefore, batchops achieve better results when directories are smaller (200000 files per OSD+). ReiserFS uses a B-tree+ to store the directory and, apparently, that tree

produces a more random pattern to place files on disk, which later produces a worse use of caches.

Figure 10 shows that for HDD-OSD+s scalability is super-linear, and usually better with batchops than with regular requests. Directory size impacts performance, therefore for non-distributed configurations performance with batchops significantly downgrades. Scalability for ReiserFS is usually smaller than for Ext4, because ReiserFS is less sensitive to the directory size. For example, Ext4 achieves a scalability higher than 30 for unlink, while ReiserFS slightly exceeds 20. Also, in the create test, Ext4 achieves larger speed-ups than ReiserFS when the shared directory is large (that is, when there are 8 OSD+s).

5.2.2 SSD-OSD+

Results for SSD-OSD+s and a single shared hugedir are depicted in Figures 11 and 12. Now, batchops perform better than regular operations for all the tests.

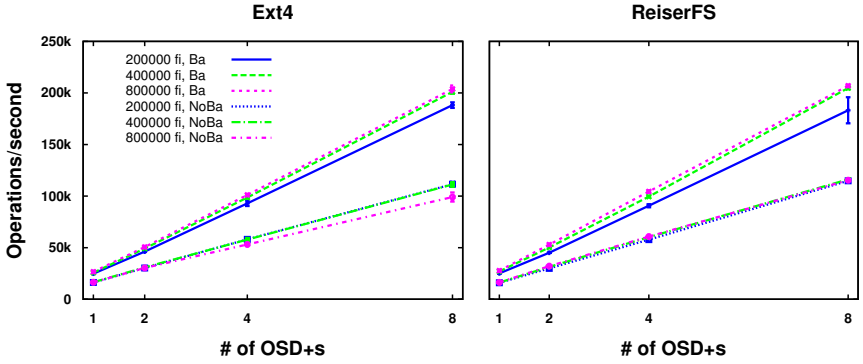
Batchops are specially helpful for the create test, because they reduce the network traffic. Batchop significantly increases the number of requests per second for each OSD+ by sending more requests to each server in each message, and sending them in parallel to all the servers too. Thanks to batchops, FPFS is always able to improve performance by 50% at least, doubling the number of operations per second in some cases of the create test.

Ext4 takes more advantage of batchops with SSD-OSD+ devices than with HDD-OSD+s in the create test. For instance, with 400 000 files per OSD+, batchops increase the number of files created per second by 30% for SSD-OSD+s and Ext4 with respect to the results obtained for HDD-OSD+s, while they only improve the results by 5% when the backend file system is ReiserFS.

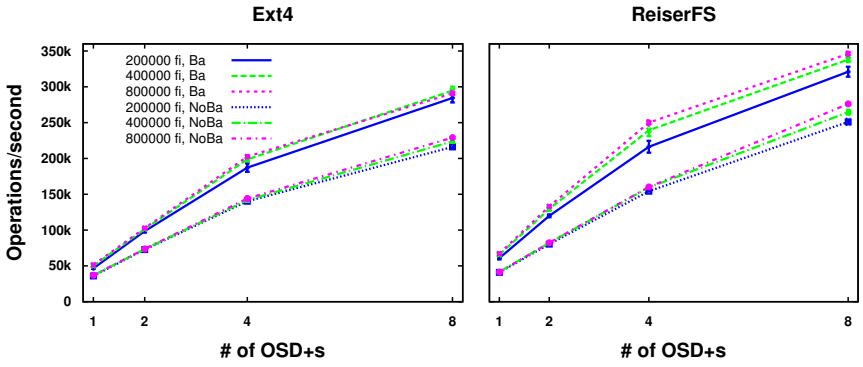
In the stat test, for both Ext4 and ReiserFS, batchops improve performance by, at least, 25%. The improvement is smaller than in the create test because the reduction in network traffic is smaller too, since stat already produces half the network traffic than create.

In the unlink test, the backend file system determines the results to a large extent, being Ext4 the file system that better leverages batchops. Specially for large directories, Ext4 performs a 60% better with than without batchops, while ReiserFS achieves a 23% of improvement. This is because batchops cause a better use of the different caches when Ext4 is the local file system. Batchops allow the serving threads in the storage nodes to carry out a request immediately after the previous one, without waiting for a new request from a client after serving a request. This specially helps Ext4 which reads and writes more blocks than ReiserFS. For 800 000 files, Ext4 exceeds RAM capacity, and using batch helps reducing the number of written blocks. By writing less, we also improve the reads performance, since there is less competition for disk. In the case of ReiserFS, it does not exceed the maximum capacity of RAM for our tests. Batchops still provide some benefits, but they are less noticeable.

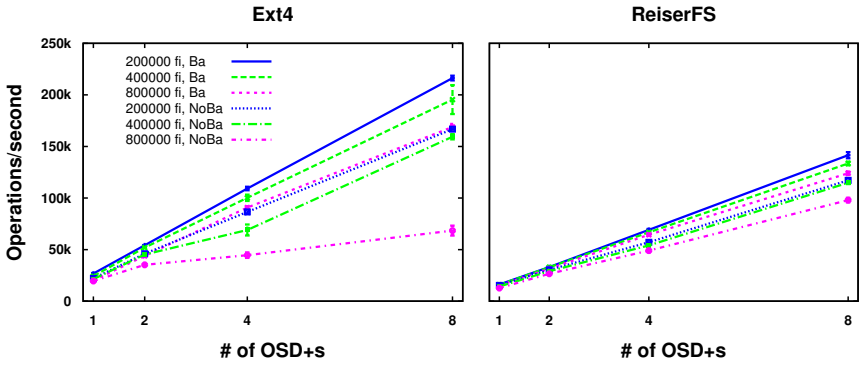
Therefore, with batchops, disk blocks in the buffer cache, fetched during the processing of a request, are likely to be used in the next request of the same thread before being evicted by requests of other threads. For ReiserFS, batchops provide a smaller benefit. Since ReiserFS produces a quite “random” access pattern from a cache’s



(a) create

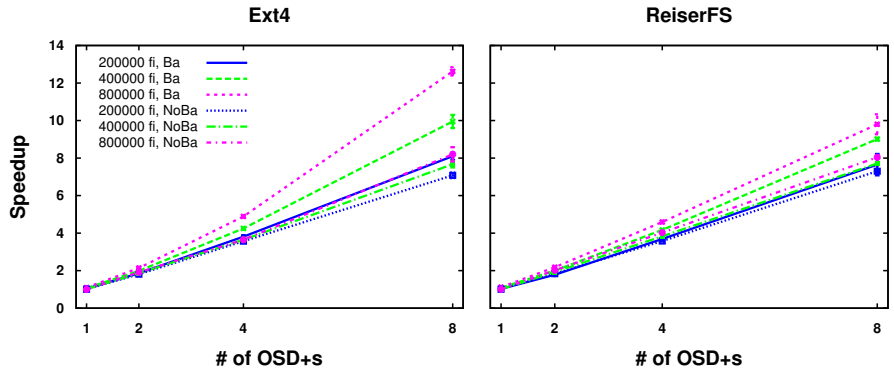


(b) stat

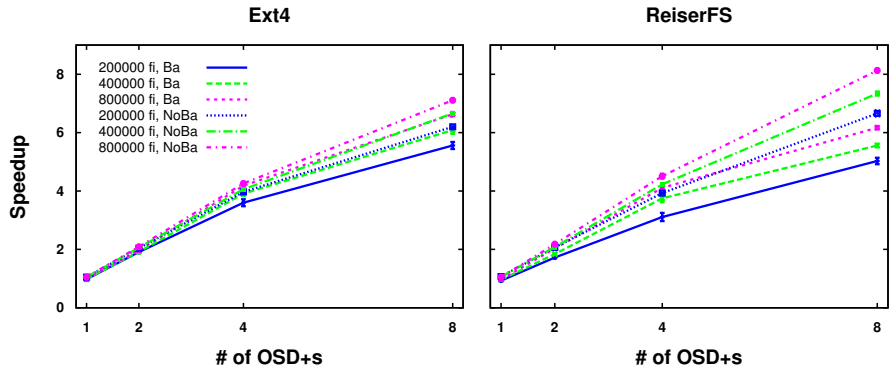


(c) unlink

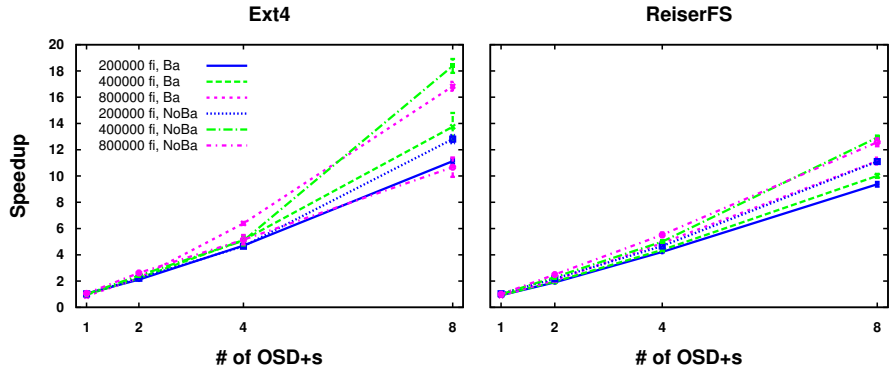
Fig. 11 Operations per second obtained by FPFS with SSD-OSD+s when using one shared hugedir. Graphs on the left show results for Ext4 and those on the right for ReiserFS. Note that the range of the Y axis can change from one test to another.



(a) create



(b) stat



(c) unlink

Fig. 12 Scalability obtained by FPFS with SSD-OSD+s when using one shared hugedir. Graphs on the left show results for Ext4 and those on the right for ReiserFS. Note that the range of the Y axis can change from one test to another.

Table 3 Performance obtained by FPFS on SSD-OSD+ devices with Ext4 and ReiserFS, when 8 hugedirs are accessed concurrently and no batch operations are used.

(a) Ext4

		1 client/directory			16 clients/directory			32 clients/directory		
Test	#OSD+	Never(s)	Dyn(%)	Alw(%)	Never(s)	Dyn(%)	Alw(%)	Never(s)	Dyn(%)	Alw(%)
create	1	138.54	0.09	0.43	81.39	-0.15	-0.56	84.68	-0.35	-0.11
	2	121.82	14.37	13.28	39.94	11.81	10.47	42.17	9.09	6.27
	4	98.60	19.82	20.95	24.50	15.11	3.74	25.28	1.05	0.43
	8	87.47	23.10	20.59	17.35	13.02	5.85	17.70	-2.00	-1.71
stat	1	78.87	0.12	-0.66	33.13	0.08	-0.17	34.43	-1.28	-0.75
	2	73.04	4.14	4.06	18.65	-2.84	-3.25	19.46	-6.94	-8.63
	4	68.33	5.35	5.88	13.37	-4.23	-7.75	12.98	-15.95	-13.74
	8	66.93	5.56	5.14	11.63	-4.26	-5.42	10.62	-3.80	-3.62
unlink	1	122.82	1.12	-0.10	181.86	-2.12	-2.35	191.35	-0.13	2.10
	2	75.69	10.86	10.84	52.63	50.63	38.98	53.59	83.65	60.73
	4	65.79	19.15	18.75	24.59	91.18	65.92	25.23	95.39	81.80
	8	56.31	24.29	22.06	14.57	5.19	3.88	14.57	25.36	5.29

(b) ReiserFS

		1 client/directory			16 clients/directory			32 clients/directory		
Test	#OSD+	Never(s)	Dyn(%)	Alw(%)	Never(s)	Dyn(%)	Alw(%)	Never(s)	Dyn(%)	Alw(%)
create	1	132.05	-0.92	-0.94	110.58	1.13	-0.44	113.07	1.66	0.57
	2	119.86	9.13	8.50	51.14	8.30	5.91	53.81	8.15	6.43
	4	99.84	10.34	10.50	25.61	9.40	6.47	26.07	9.05	6.19
	8	88.52	10.94	11.28	16.46	3.19	-0,67	16,39	-4,87	-8,44
stat	1	76.46	-0.18	-0.49	41.49	0.09	-0.26	42.84	-0.38	-0.09
	2	70.89	4.87	3.90	20.16	4.41	4.21	21.10	3.43	1.45
	4	68.08	3.90	4.35	11.68	1.00	1.70	11.69	-4.78	-4.84
	8	66.64	4.39	5.67	10.57	1.11	0.25	9.77	1.82	2.07
unlink	1	160.64	-0.58	-0.61	173.02	-1.10	1.10	190.02	-1.93	-1.06
	2	84.53	1.88	1.68	82.98	5.02	2.79	86.80	12.23	11.26
	4	67.13	7.53	9.42	40.43	5.11	5.76	40.99	17.83	15.03
	8	62.70	5.89	6.08	20.93	5.17	3.08	21.55	10.21	10.18

point of view [22], the improvement that can be obtained from the "aggregate disk cache" is limited.

Figure 12 shows that batchops hardly affect scalability. For the create test, the most noticeable change is for 800 000 files per OSD+, and 8 OSD+s, where scalability is super-linear. In the stat test, however, batchops slightly reduce the scalability for ReiserFS, and for unlink it remains super-linear for both Ext4 and ReiserFS. With batchops, we significantly reduce the amount of network traffic, specially when the directory is not distributed. Therefore, when we distribute a directory with batchops, the network reduction is not as high as the one achieved with no-batchops.

These results diverge from the ones with HDD-OSD+s, where batchops significantly increased the scalability. While, with batchops and HDD-OSD+s, the directory size significantly affected several tests, with SSD drives, again, we remove all the head-seeks that provoked this increment.

5.3 Multiple Hugedirs

Distribution is beneficial for a single hugedir accessed by hundreds or thousands of clients. However, results can be rather different when several hugedirs are con-

Table 4 Performance obtained by FPFS on SSD-OSD+ devices with batchops, and Ext4 and ReiserFS, when 8 hugedirs are accessed concurrently and batch operations are deployed.

(a) Ext4

		1 client/directory			16 clients/directory			32 clients/directory		
Test	#OSD+	Never(s)	Dyn(%)	Alw(%)	Never(s)	Dyn(%)	Alw(%)	Never(s)	Dyn(%)	Alw(%)
create	1	74.32	-2.14	-2.14	75.28	-2.40	-2.31	74.10	-4.34	-3.75
	2	37.44	6.31	1.78	37.27	15.06	11.63	37.15	17.52	33.45
	4	23.01	-4.55	-10.35	20.97	26.05	21.72	21.38	24.75	21.45
	8	16.27	-33.73	-34.73	13.52	13.78	9.12	13.63	18.96	10.97
stat	1	28.29	-0.24	-0.61	17.96	3.74	4.13	18.58	2.39	7.80
	2	24.74	-8.44	-9.03	10.77	-3.11	0.61	12.35	-7.71	-6.73
	4	22.04	6.37	7.43	9.23	-5.80	-1.53	9.68	-5.35	-9.98
	8	19.69	56.67	55.74	8.28	-5.28	-3.49	8.69	-2.83	-6.43
unlink	1	133.07	-1.89	-4.21	201.93	-12.36	-11.80	214.32	-9.46	-7.33
	2	33.84	56.10	46.47	53.04	6.13	8.27	58.63	15.72	8.39
	4	20.08	23.33	3.73	20.53	2.02	-1.09	21.27	19.41	13.85
	8	13.12	-26.83	-30.60	12.63	-26.01	-26.09	12.79	-28.34	-29.10

(b) ReiserFS

		1 client/directory			16 clients/directory			32 clients/directory		
Test	#OSD+	Never(s)	Dyn(%)	Alw(%)	Never(s)	Dyn(%)	Alw(%)	Never(s)	Dyn(%)	Alw(%)
create	1	85.54	0.22	-0.33	94.57	-3.41	-2.35	100.71	-11.75	-11.41
	2	42.29	-0.71	-4.97	43.91	18.67	14.06	40.81	24.78	16.95
	4	24.68	-15.96	-19.14	20.59	35.66	32.90	20.01	14.55	25.46
	8	16.61	-32.26	-35.82	13.20	18.61	13.98	12.89	7.29	-6.86
stat	1	25.01	-1.03	0.45	23.83	-0.62	-0.64	23.08	1.86	2.00
	2	21.57	-4.80	-5.87	11.69	3.17	10.65	12.60	-1.41	2.68
	4	19.33	17.54	17.33	8.23	2.45	0.19	8.24	-1.18	5.72
	8	18.38	64.06	65.24	6.78	16.83	18.65	8.00	1.34	1.77
unlink	1	141.00	-1.98	-2.66	144.73	0.41	-1.48	150.40	-3.10	-3.38
	2	70.91	-5.65	-6.44	71.51	-1.62	-4.76	71.94	-3.65	-4.28
	4	32.16	2.90	2.97	34.64	0.60	0.22	35.11	-3.48	-4.31
	8	18.33	-8.13	-7.57	18.06	-2.87	-5.96	18.31	-5.76	-4.00

currently accessed by a few clients. In this section we analyze the performance of batchops when several huge directories are concurrently accessed by a few clients by using SSD-OSD+ devices. The following tests use 8 directories (each containing 320 000 files) accessed by 1, 16 and 32 clients per directory. Note that, 1 client per directory is an example for non-shared directories, and with 32 clients per directory, there are 256 clients altogether. Once again, each batchop request includes 1 000 operations.

Tables 3 and 4 show, for each number of processes per directory, absolute application times when hugedirs are never distributed in the first column. The column labelled *Dyn* gives relative application-time variations, in percentage, with respect to the absolute times, when hugedirs are distributed dynamically (i.e., when a directory exceeds 8 000 files). The column labelled *Alw* also gives relative application-time variations, in percentage, with respect to the absolute times, but when any directory is always distributed (i.e., when threshold is 0). Confidence intervals (not showed) are smaller than 10% of the mean. A positive/negative percentage means an increase/decrease in time, and, hence, a worse/better performance. While Table 3 shows results for SSD-OSD+s without batch operations, Table 4 shows results for SSD-OSD+s with batchops.

The first thing we can observe is that, when comparing absolute times in columns *never*, batchops improve performance in general (for both Ext4 and ReiserFS), specially when there is one client per directory. When directories are distributed, results obtained by batchops are more variable, as it already happens with regular operations, and they also depend on the number of OSD+s, number of processes per directory and backend file system. However, there are some noticeable differences now with respect to a system without batchops.

For Ext4 and the create test, distribution and batchops improve results with respect to *never* when there is 1 client per directory, but slightly downgrade them when the number of clients per directory grows. However, absolute times are inferior now in any case. For the stat operations, results are comparable with those without batchops, except for 1 client per directory and 8 OSD+s, where the distribution with batchops increments the application time from 5% to 56%. For the unlink test, distribution with batchops behaves much better than without batchops, and now there is only a small increase in the application time. Moreover, with 8 OSD+s, batchops are able to significantly reduce the application time. Exception appears for 1 process per directory and 2 OSD+s, although, considering absolute times, batchops still reduce the application time considerably.

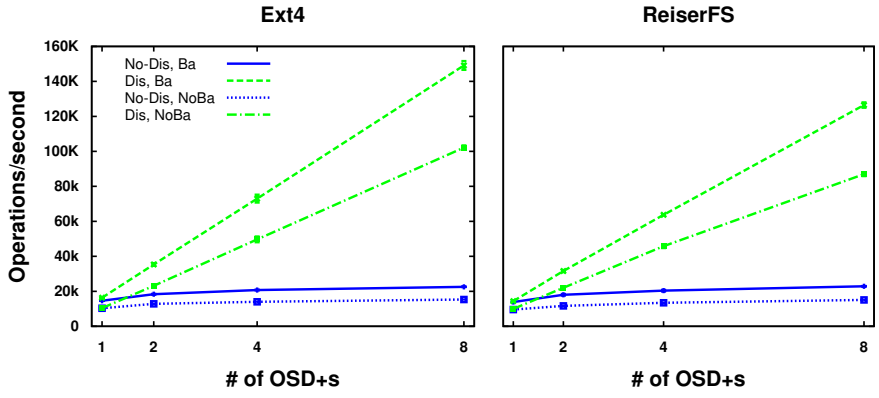
For ReiserFS and the create benchmark, the behavior is similar to that of Ext4. For the stat test and 32 clients per directory, results are comparable to those we have without batchops. For 1 and 8 clients per directory and for 8 OSD+s (and, sometimes, 4 OSD+s), distribution increments times respect to *never* more than when we do not have batchops. For the unlink case, differently to what happens with regular operations, distribution and batchops reduce the application time with respect to *never*.

In summary, although the distribution of hugedirs can downgrade the performance in some cases, results also show that batchops can help to reduce the possible negative effects caused by such distribution. We believe this is because the threads attending requests in the servers can process more requests in a shorter time. This improves caches' performance and reduces the overhead produced by disk contentions. The results obtained with hard drives (not included) confirm these findings.

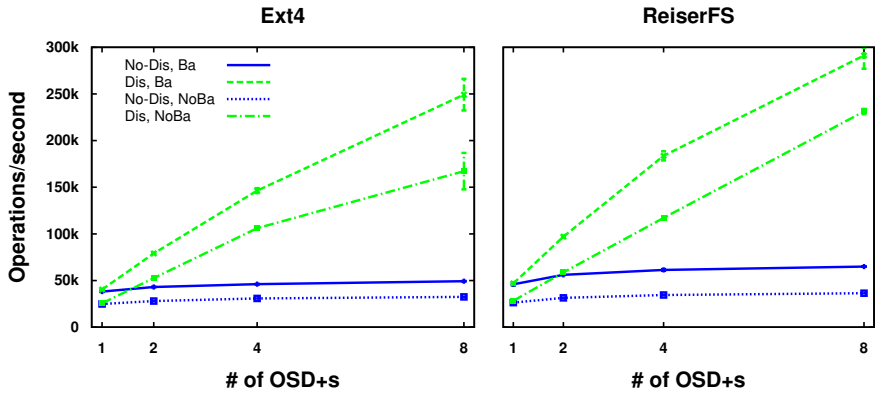
5.4 Mixed directories

Figure 13 depicts the throughput in operations/s achieved by FPFS with SSD-OSD+ devices when two hugedirs, a distributed one and a non-distributed one, are accessed at the same time by 128 clients each. Results are labelled as "Dis-Ba", "No-Dis-Ba", "Dis-NoBa", and "NoDis-NoBa", where "Dis" stands for distributed and "Ba" for batching. There are always 1 280 000 files per directory, evenly shared out among clients. Again, each batchop includes 1 000 operations. Batchops always improve the performance of both directories in all cases, and, as in a single shared directory, the reduction in network traffic and a better use of the caches explain the improvements.

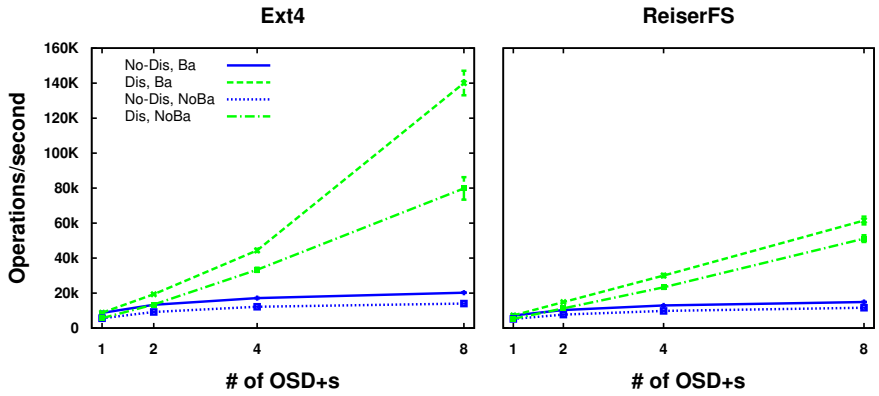
In the create test, batchops achieve an improvement of more than 30% for both the non-distributed and distributed directory, and both Ext4 and ReiserFS, due to the reduction in network traffic.



(a) create



(b) stat



(c) unlink

Fig. 13 Operations per second obtained by FPFS with SSD-OSD+s when a distributed hugedir and a non-distributed hugedir are concurrently accessed by 256 clients. Graphs on the left show results for Ext4 and those on the right for ReiserFS. Note that the range of the Y axis can change from one test to another.

Batchops obtain the best improvements in the stat test. For the non-distributed directory and Ext4, batchops improve the throughput by 34% at least, and by 44% with ReiserFS. In the case of the distributed directory and Ext4, batchops achieve a maximum improvement of 36%, and with ReiserFS the improvement reaches a 40%. Since this is a read-only test, which reads related directory entries and i-nodes, batchops allow servers to make a better use of caches and prefetching, because they process many requests in a row.

Finally, results in the unlink test are similar to those in Section 5.2.2 where Ext4 performs better than ReiserFS as the number of OSD+s increases. For the distributed directory, Ext4 achieves a 40% of improvement with 8 OSD+s, while ReiserFS gets 16%. For Ext4 and the non-distributed directory the improvement is around 30% and, in the case of ReiserFS, the improvement is around 25%.

6 Related work

To the best of our knowledge, batchops have not been proposed in the parallel/distributed file systems field, although some network file systems support similar ideas. For instance, NFSv4 [30] reduces latency for multiple operations by bundling different RPC calls into a single request. Operations `lookup`, `open`, `read` and `close`, for example, can be sent once over the wire, and the server can execute the entire compound call as a single entity. Version 2 of the Server Message Block (SMB2) [32] is also able to send an arbitrary set of commands in a single request, thereby improving the performance by reducing the number of network round trips. The compounding ability in SMB2 is very flexible; commands packed in a single request can be unrelated (executed separately, potentially in parallel) or related (executed in sequence, with the output of one command available to the next); responses can also be compounded or sent separately. Note that, for both network file systems, there exists a single server. Our approach, however, allows batch operations in a distributed multi-server environment.

Ideas similar to batchops have been used in many other different areas. For instance, Linux kernel 3.14 [50] includes a feature, called *automatic TCP corking*, to help applications to do small write()/sendmsg() on TCP sockets. Previous versions of Linux also allow the use TCP corking, although Linux kernel 3.14 is the first one to include automatic TCP corking. This feature allows to delay the dispatch of messages in a socket in order to coalesce more bytes in the same packet, thereby lowering the total amount of sent packets. However note that it is not a proper batch operation. This technique complements batchops, although a study of performance of both is postponed to the future. Also in the network area, IX [5] is an operating system that uses an adaptive batching in every stage of its network stack in order to improve performance on congestion. IX batches network requests in the presence of network congestion and allows application threads to issue batched system calls. Tyche [21] is a network storage protocol over raw Ethernet that uses an adaptive batching mechanism to achieve high link utilization under high degrees of I/O concurrency and small I/O requests. Tyche proposes a dynamic technique that varies the degree of batching depending on the throughput achieved. In Tyche, a batch message is composed

of several I/O requests, reads or writes, issued by the same or different application threads.

Similarly, but in the grid computing area, Chervenak *et al.* [10] use what they call *bulk operations* in the implementation of a *Replica Location Service* (RLS). RLS provides a mechanism for registering the existence of replicas and discovering them within a grid environment. They store *catalogs* that map logical names to target names. In turn, clients send queries to the servers in order to discover replicas associated with a logical name. Among the operations they support, they include bulk operations to add/delete entries and/or attributes to the catalogs, and to perform query operations on them. They include 1 000 requests per bulk operation. Their experiments show a significant performance improvement for a single client. However, as the number of clients increases, the performance advantage of bulk queries decreases. We obtain a similar behavior in our experimental results, although our improvement with batchops versus regular operations is much higher than theirs, and batchops still provide noticeable benefits with a large number of clients.

OpenStack Swift also includes in its Object Storage API two bulk operations: delete [38] and archive extraction [37]. Bulk delete can remove up to 10 000 objects or containers (configurable) in one request. The archive extraction allows to expand a tar file into a Swift account in a single request. Only regular files are uploaded; empty directories, symlinks, etc. are not uploaded.

Another area where reducing the number of requests is especially useful is Internet. The next major version of HTTP (HTTP/2 [6]) will use bulk operations to accelerate communications. Currently, services like Google or Facebook also try to reduce the number of HTTP requests by batching operations together. Google [27] uses batch requests in Google Base [24], Google Spreadsheet [23], Google Calendar [25] and Google Cloud Storage API [26]. Specifically, the Google Cloud Storage API provides with batch requests to bundle API calls together and reduce the number of HTTP connections clients have to make. In a similar vein, Facebook provides its Ads API [15] and Graphics API [16] with batch requests to send several requests of the same type in a single HTTP request. Depending on the type of operation, the maximum number of requests per batch operation varies.

7 Conclusions

In distributed or parallel file systems, workloads that perform the same operation on multiple files, such as the migration of a directory, the creation of a set of files in a directory, or the removal of all the files in a directory, usually incur in large amounts of network traffic. In order to deal with these workloads in a more efficient way, we present the design and implementation of operations that embed hundreds or thousands of operations of the same type into a single message. These operations are possible in FPFS because its namespace distribution is based on directories, which usually contain related files. With these operations, that we call batchops, we significantly reduce the amount of network messages and, therefore, network delays and round-trips. We also manage to reduce the overall network congestion, making a better use of the available I/O and processing resources.

We add the management of batchops to FPFS by including specific operations to create (`openv` and `closev`), get the status (`statv`) and unlink (`unlinkv`) files in a batch fashion. For each operation, we modify the message format to include a list of entries within the same directory. Our batch operations include semantics to specify the behavior in case of failure of an operation in the batchop. The implementation also supports huge directories in a transparent way; clients do not need to differentiate between distributed and non-distributed directories when issuing batchops.

The experiments show that batchops help us to reduce the network overhead, and increment the number of operations/s in OSD+s, improving FPFS performance. Specifically, in tests that make a more intensive use of the network, such as the creation of a single shared directory, performance improves by a 50% at least, reaching a 100% in some cases. In the case of stat, the improvement is always around 25%. Finally, for the unlink test, which issues both read and write requests, the backend file systems determine results to a large extent, being Ext4 the one that better leverages batchops with an improvement of 60%, while ReiserFS obtains a 23% when using this kind of operations.

Thanks to batchops, FPFS can create, stat and delete around 200 000, 300 000 and 200 000 files per second, respectively, with just 8 SSD-OSD+ devices and a regular Gigabyte network.

Finally, our experiments also show that, while batchops are usually beneficial with SSD-OSD+s, there are some cases where they downgrade the performance when HDD-OSD+ devices are used. The problem is that batchops affect the way files are allocated on disk. For HDD-OSD+s, this different layout increases the I/O time in some cases due to more head seeks and less efficient disk caches.

Although some common file operations can already take advantage of batchops (e.g., `ls -l` and `rm -rf`), as future work, we plan to identify specific HPC applications and scenarios that can benefits from our proposal.

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

References

1. 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), pp. 175–184 (2008)
2. Artiaga, E., Cortes, T.: Using filesystem virtualization to avoid metadata bottlenecks. In: Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 562–567 (2010)
3. Avilés-González, A., Piernas, J., González-Férez, P.: A metadata cluster based on OSD+ devices. In: Proceedings of the 23rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 64–71 (2011)
4. Avilés-González, A., Piernas, J., González-Férez, P.: Scalable huge directories through OSD+ devices. In: Proceedings of the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2013, Belfast, United Kingdom, pp. 1–8 (2013)
5. Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C., Bugnion, E.: IX: A protected dataplane operating system for high throughput and low latency. In: Proceedings of 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), pp. 49–65 (2014)
6. Belshe, M., Twist, Peon, R., Thomson, M.: Hypertext transfer protocol version 2 (2015). URL http://datatracker.ietf.org/doc/draft-ietf-httpbis-http2

7. 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'09), pp. 1–12 (2009)
8. Braams, P.J.: High-performance storage architecture and scalable cluster file system (2008). URL http://wiki.lustre.org/index.php/Lustre_Publications
9. 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), pp. 290–298 (2003)
10. Chervenak, A.L., Palavalli, N., Bharathi, S., Kesselman, C., Schwartzkopf, R.: Performance and scalability of a replica location service. In: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC'04), pp. 182–191 (2004)
11. Cray Inc.: HPCS-IO (2012). URL http://sourceforge.net/projects/hpcs-io
12. Dilger, A.: Lustre future development (2012). URL http://storageconference.us/2012/Presentations/M04.Dilger.pdf. Symposium at the 28th IEEE Conference on Massive Data Storage (MSST'12)
13. Dilger, A.: Lustre metadata scaling (2012). URL http://storageconference.us/2012/Presentations/T01.Dilger.pdf. Tutorial at the 28th IEEE Conference on Massive Data Storage (MSST'12)
14. Dunn, M.P.: A new I/O scheduler for solid state devices. Master's thesis, Texas A&M University (2009)
15. Facebook Inc.: Batch requests. URL https://developers.facebook.com/docs/reference/ads-api/batch-requests
16. Facebook Inc.: Making multiple API requests. URL https://developers.facebook.com/docs/graph-api/making-multiple-requests/
17. Fikes, A.: Storage architecture and challenges. In: Google Faculty Summit 2010 (2010). URL http://research.google.com/university/relations/facultysummit2010/storage_architecture_and_challenges.pdf
18. Freitas, R., Slember, J., Sawdon, W., Chiu, L.: GPFS scans 10 billion files in 43 minutes. Tech. Rep. RJ10484, IBM Almaden Research Center (2011). URL http://www.almaden.ibm.com/storagesystems/resources/GPFS-Violin-white-paper.pdf
19. 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), pp. 1–17 (1997)
20. Gibson, G.A., Nagle, D., Amiri, K., Butler, J., Chang, F.W., Gobioff, H., Hardin, C., Riedel, E., Rochberg, D., Zelenka, J.: A cost-effective, high-bandwidth storage architecture. In: Proceedings of the international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98), pp. 92–103 (1998)
21. González-Férez, P., Bilas, A.: Reducing CPU and network overhead for small I/O requests in network storage protocols over raw Ethernet. In: Proceedings of the IEEE 31st Conference on Mass Storage Systems and Technologies (MSST) (2015)
22. González-Férez, P., Piernas, J., Cortés, T.: Evaluating the Effectiveness of REDCAP to Recover the Locality Missed by Today's Linux Systems. In: Proceedings of the IEEE/ACM International Symposium on Modeling, Analysis, and Simulation Computer and Telecommunication Systems (MASCOTS'08), pp. 1–4 (2008)
23. Google Inc.: Google spreadsheet (2013). URL https://developers.google.com/chart/interactive/docs/spreadsheets
24. Google Inc.: Google base (2014). URL http://www.google.com/merchants/default
25. Google Inc.: Google calendar (2014). URL https://www.google.com/calendar
26. Google Inc.: Google cloud storage: Sending batch requets (2014). URL https://developers.google.com/storage/docs/json_api/v1/how-tos/batch
27. Google Inc.: Using batch operations (2014). URL http://code.google.com/p/gdata-python-client/wiki/UsingBatchOperations
28. Kim, J., Oh, Y., Kim, E., Choi, J., Lee, D., Noh, S.H.: Disk Schedulers for Solid State Drivers. In: Proceedings of the 7th ACM International Conference on Embedded Software, pp. 295–304 (2009)
29. Lin, W., Wei, Q., Veeravalli, B.: WPAR: A weight-based metadata management strategy for petabyte-scale object storage systems. In: Proceedings of the 4th International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI'07), pp. 99–106 (2007)
30. MacDonald, A.: Nfsv4. ;login: **37**(1), 28–35 (2012)
31. Mesnier, M., Ganger, G.R., Riedel, E.: Object-based storage. IEEE Communications Magazine **41**(8), 84–90 (2003)

32. Microsoft Inc.: Server Message Block (SMB) Version 2.0 Protocol Specification (2007). URL https://msdn.microsoft.com/en-us/library/cc212614.aspx
33. Miranda, A., Effert, S., Kang, Y., Miller, E.L., Brinkmann, A., Cortes, T.: Reliable and randomized data distribution strategies for large scale storage systems. In: Proceedings of 18th IEEE International Conference on High Performance Computing (HiPC'11), pp. 1–10 (2011)
34. Morrone, C., Loewe, B., McLarty, T.: mdtest HPC Benchmark (2014). URL http://sourceforge.net/projects/mdtest
35. Newman, H.: HPCS mission partner file I/O scenarios, revision 3 (2008). URL http://wiki.old.lustre.org/images/5/5a/Newman_May_Lustre_Workshop.pdf
36. OpenSFS, EOFS: The Lustre file system (2015). URL http://www.lustre.org
37. OpenStack Foundation: Archive auto extraction (2014). URL http://docs.openstack.org/developer/swift/middleware.html#module-swift.common.middleware.bulk
38. OpenStack Foundation: Bulk delete (2014). URL http://docs.openstack.org/api/openstack-object-storage/1.0/content/bulk-delete.html
39. 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), pp. 15–30 (2011)
40. 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), pp. 1–6 (2012)
41. Polyakov, E.: The Elliptics network (2009). URL http://reverbrain.com/elliptics
42. Ren, K., Patil, S., Gibson, G.: A case for scaling HPC metadata performance through de-specialization. In: Proc. of the 7th Petascale Data Storage Workshop Supercomputing (PDSW), pp. 30–35 (2012)
43. Seagate Inc.: Kinetic open storage (2013). URL https://developers.seagate.com/display/KV/Kinetic+Open+Storage+Documentation+Wiki
44. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The Hadoop distributed file system. In: Proceedings of the 26th IEEE Conference on Massive Storage Systems and Technologies (MSST'10), pp. 1–10 (2010)
45. Sinnamohideen, S., Sambasivan, R.R., Hendricks, J., Liu, L., Ganger, G.R.: A transparently-scalable metadata service for the Ursa Minor storage system. In: Proceedings of USENIX Annual Technical Conference (ATC'10), pp. 1–14 (2010)
46. Skeen, D., Stonebraker, M.: A formal model of crash recovery in a distributed system. IEEE Transactions on Software Engineering **9**(3), 219–228 (1983)
47. Sun-Oracle: Lustre tunning (2010). URL http://wiki.lustre.org/manual/LustreManual18_HTML/LustreTuning.html
48. SwiftStack Inc.: Kinetic motion with Seagate and OpenStack Swift (2013). URL https://swiftstack.com/blog/2013/10/22/kinetic-for-openstack-swift-with-seagate/
49. The PVFS Community: The Orange file system (2015). URL http://orangefs.org
50. Torvalds, L., et al.: Linux 3.14 features (2014). URL http://kernelnewbies.org/Linux_3.14
51. Wang, F., Xin, Q., Hong, B., Brandt, S.A., Miller, E.L., Long, D.D.E., McLarty, T.T.: 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), pp. 139–152 (2004)
52. Weijia, L., Wei, X., Shu, J., Zheng., W.: Dynamic hashing: Adaptive metadata management for petabyte-scale file systems. In: Proceedings of the 23rd IEEE Conference on Massive Storage Systems and Technologies (MSST'06), pp. 159–164 (2006)
53. 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), pp. 307–320 (2006)
54. Wheeler, R.: One billion files: Scalability limits in Linux file systems. In: LinuxCon'10 (2010). URL http://events.linuxfoundation.org/slides/2010/linuxcon2010_wheeler.pdf
55. Zhu, Y., Jiang, H., Wang, J.: Hierarchical Bloom Filter Arrays (HBA): A novel, scalable metadata management system for large cluster-based storage. In: Proceedings of IEEE International Conference on Cluster Computing (Cluster'04), pp. 165–174 (2004)