Striping without Sacrifices:
Maintaining POSIX Semantics in a Parallel File System

Jan Stender[1], Björn Kolbeck[1], Felix Hupfeld[1]
Eugenio Cesario[2], Erich Focht[3], Matthias Hess[3],
Jesús Malo[4], Jonathan Martí[4]

[1]*Zuse Institute Berlin (ZIB), Takustr. 7, 14195 Berlin, Germany*
[2]*Institute High Performance Computing and Networks of the National Research Council of Italy (ICAR-CNR),*
DEIS-UNICAL, P. Bucci 41-C, 87036 Rende, CS, Italy
[3]*NEC HPC Europe GmbH, Hessbruehlstr. 21b, 70656 Stuttgart, Germany*
[4]*Barcelona Supercomputing Center (BSC), c/ Jordi Girona 31, Barcelona, Spain*

Abstract

Striping is a technique that distributes file content over multiple storage servers and thereby enables parallel access. In order to be able to provide a consistent view across file data and metadata operations, the file system has to track the layout of the file and know where the file ends and where it contains gaps. In this paper, we present a light-weight protocol for maintaining a consistent notion of a file's layout that provides POSIX semantics without restricting concurrent access to the file. In an evaluation, we show that the protocol scales and elicit its corner cases.

1 Introduction

Parallel file systems reach their superior I/O performance and decent scalability with the help of *striping*. Instead of storing a file only on one storage server, a parallel file system splits up the data of a file into chunks and distributes these chunks across multiple storage servers (see Fig. 1). Because clients can retrieve or calculate the location of the chunks on storage servers, they can access chunks directly in parallel.

With this direct, parallel access to storage, clients can execute operations on the file in parallel on several storage servers with the aggregated bandwidth and resource capacity of all servers. The independence of storage resources also helps when the system needs to be extended. In order to handle increasing storage or bandwidth demands, the parallel file system can be scaled by adding more storage servers.

Ideally, a parallel file system would completely hide the complexity of distributed storage and exhibit a behavior as it is specified by POSIX [11] and closely followed by local file systems. Then users could run any application on top of a parallel file system without any modifications or caveats. The distributed nature of a parallel file system, however, requires file system designers
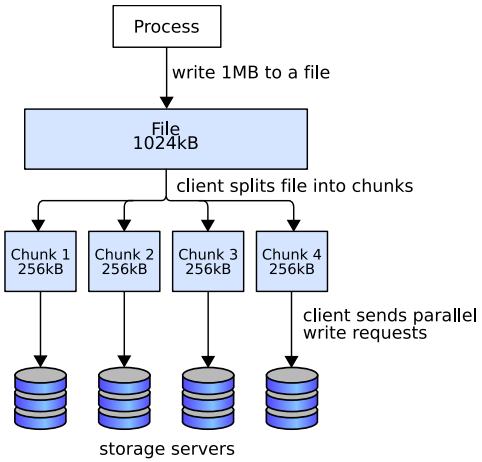

Figure 1: In parallel file systems, file data is split into chunks which are distributed among a set of storage servers.

to take means to ensure the consistency of the file abstraction as it is presented to clients.

A particular problem of striping is ensuring a consistent view on the size of the file. When a file is split into chunks, distributed over multiple servers and accessed in parallel, the file size is conveyed by the length of the last chunk of the file that defines the end-of-file (EOF) marker. However, with multiple clients modifying the chunks in parallel, no entity in the file system knows the location of this last block and thus neither metadata servers, storage servers nor clients know the current file size [8, 12].

This problem becomes worse when files can contain *gaps*. POSIX allows a client to seek to any offset, even beyond EOF, and write data there. When a client reads data from unwritten parts in-between that are within the EOF bound, the file system has to return zeros. These gaps in the file can be implemented efficiently as *sparse files*: the file system does not allocate space for these

unwritten parts of the file and pads any reads with zeros on demand. In order to correctly identify the absence of file data on disk as a gap or the end of file, the file system needs to know the current file size.

In lack of better techniques, current parallel file systems usually either sacrifice consistency of concurrent file access in order to avoid this problem altogether, or do not fully implement this case. Similar to networked file systems like NFS or AFS, they require developers to be aware of the idiosyncrasies of each file system.

In this paper we show that it is possible to support a consistent notion of a file's size even with sparse files, without having to make sacrifices for consistency, concurrency or performance. We present a striping protocol for a parallel file system with semantics that are compatible with local file systems, and which does not restrict access or hinder scalability. Section 2 gives a detailed treatment of the problem of striping in parallel file systems and names the challenges to solve. We then present our protocol for maintaining a consistent view on file size and data in Sec. 3, demonstrate its scalability in Sec. 4, give an overview of how the problem is tackled by other parallel file systems in Sec. 5 and conclude in Sec. 6.

2 File System Semantics

To simplify the development of applications for UNIX-like operating systems, application developers expect the behavior of file systems to be uniform and independent of their underlying implementations. With the aim of enforcing a uniform behavior across the multitude of different file systems, POSIX [11] defines certain semantics that file systems have to offer.

In particular, applications can expect the file system to provide a consistent notion of a file's size, even under concurrent access by multiple applications. This file size has to be consistent to what has been actually written. Primarily, this means that reads beyond the current end-of-file (EOF) must be answered accordingly.

The size of a file is also present in the result of a `stat` call. Although POSIX does not make statements about the consistency of `stat` information and the actual location of the end-of-file marker, applications and users of the file system expect this information to be updated at least periodically when the file is opened and under change. For a closed file, the information should correctly reflect the file size as it is on disk.

POSIX (and all common local file systems) support the creation of so-called *gaps*. Applications can `seek` to arbitrary offsets in a file and write data there. Seeking and writing is not restricted to the current bounds of the file. A write beyond the current EOF creates a region between the former EOF and the write offset to which no

(a) Process A creates a gap

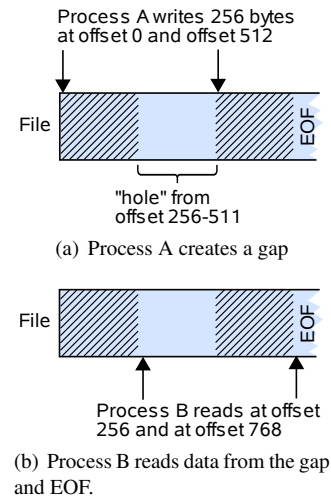(b) Process B reads data from the gap and EOF.

Figure 2: (a) shows how process A creates a gap by writing 256 bytes at offset 0 and 512. (b) shows process B reading 256 bytes from the same file at offset 256 (gap) and 768 (EOF). Process B will receive a zero-padded buffer of 256 bytes for the former, and an empty (zero length) buffer for the latter read.

data has been explicitly written. Such a region is generally referred to as a *gap* (see Fig. 2(a)).

POSIX requires that data within a gap is read as zeros (see Fig. 2(b)). This behavior can be implemented by filling the gaps with zeros, or, more efficiently, by storing the file as a *sparse file*. For sparse files, the file system does not allocate space for gaps, which creates *holes* in the sequence of file blocks on disk. Any reads that encounter such a hole are padded with zeros. When reading beyond EOF, POSIX requires that the range of requested bytes is pruned such that only those bytes between the read offset and the last byte of the file are returned. In turn, an application is able to detect the EOF when it receives a smaller amount of bytes than it requested.

Ensuring a POSIX-compliant behavior with respect to gaps and EOFs is not an issue in local file systems, because the access to file data and metadata can be serialized without having a noticeable effect on performance. In a parallel file system, however, file data is distributed across multiple storage servers. This involves that in some cases, file data stored on a single server is insufficient to distinguish between a gap and the EOF.

The problem is illustrated in Fig. 3. A process writes to an empty file at two different offsets. The file system client maps these writes to their corresponding chunks and sends them to the responsible storage servers. With a sparse file implementation, this operation necessarily creates a hole at the storage server that did not receive any data. If a second process tries to read data at an offset

inside the gap, the responsible storage server finds a hole and needs to decide whether it should respond with zero-padded or pruned data.
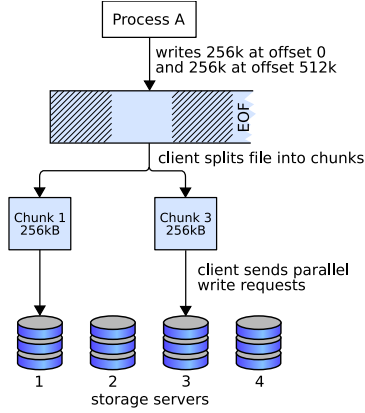
(a) Process A implicitly creates a hole.
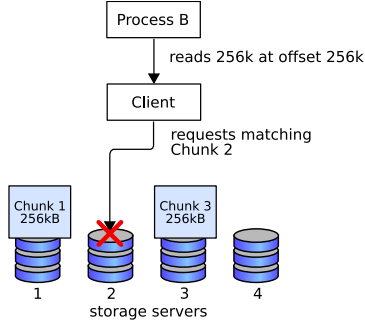
(b) Process B reads data from the hole.

Figure 3: In (a), process A writes two separate blocks of data to a file. The file system client translates these blocks into chunks that reside on storage servers 1 and 3. On storage server 2, no chunk is written, i.e. a hole exists. In (b), Process B tries to read data from the same file at the offset of the hole. Storage server 2 cannot decide if this missing chunk indicates a gap or EOF.

3 Striping Protocol

To implement full POSIX compliance in presence of gaps, the striping protocol has to make sure that a storage server can distinguish between a hole and the EOF when it receives a request for a missing chunk. To decide for one of the two cases, storage servers require knowledge about the current file size.

It is neither feasible to store the file size on a central server, nor to maintain synchronously updated copies of it across all servers. The former approach would require locking mechanisms that prevent multiple clients from concurrently changing the file size, which greatly restricts access parallelism on a single file. The latter approach would require some sort of synchronization across all storage servers with each file size change, which would significantly slow down frequent operations like append writes.

With the aim of making append operations as efficient as possible, our protocol relies on loosely-synchronized file size copies maintained by each storage server. When files grow, file size updates are disseminated between storage servers in an asynchronous, best-effort manner. When missing chunks are requested, storage servers attempt to distinguish between holes and EOFs by means of their local views on the current file size. Only if this turns out to be impossible, which may be the case when file size updates were lost or the requested chunk is outside the file boundaries, file size views have to be explicitly synchronized.

3.1 Definitions

We assume that the file system manages a set of files F. Each file is striped across an individual, immutable set of storage servers $S_f = \{s_1, \ldots, s_n\}$. Per file f, each storage server $s \in S_f$ holds a mutable, potentially empty set of locally managed chunks $C_{s,f} = \{c_n, \ldots, c_m\}$ that contain the file data. Chunks are numbered; n and m refer to chunk numbers that grow monotonously with their offsets in f. We further assume that clients can determine the storage server that is responsible for a certain chunk.

In $gmax_{s,f} \in \mathbb{N}$, each server s locally stores information about the globally largest chunk number of a file f, i.e. the number of the last chunk of f. This number reflects a server's local view on the current file size.

3.2 Protocol

To simplify our presentation, we assume that read and write operations always refer to entire chunks. We also assume that the size of a file grows monotonously, i.e. once created, a file will never shrink. How the protocol can be enhanced to take care of arbitrary file size changes will be described in Sec. 3.3.

Write. The `write` operation adds a new chunk to a file, or changes the content of an existing chunk. When receiving a write request, a storage server locally writes the chunk. In a second step, it checks whether the previously written chunk is outside the former boundaries of the file, according to $gmax$. If so, it communicates a new value for $gmax$ to all storage servers, which in turn update their local $gmax$ values if they are smaller. As long as no update messages are lost, all storage servers

have a synchronized view on $gmax$ as the number of the last chunk of the file.

```
BEGIN write(f, c_n)
   – write the chunk
   C_{s,f}  ←  C_{s,f}  ∪  {c_n}
   – if a new chunk was created, send a gmax update message
   – to all storage servers (including the local one)
   IF  n  >  gmax_{s,f}  THEN
      SEND  SET_GMAX(f, n)  TO  S_f
   END IF
END

UPON SET_GMAX(f, n)
BEGIN
   – if a larger gmax was received, replace the current gmax
   IF  n  >  gmax_{s,f}  THEN
      gmax_{s,f}  ←  n
   END IF
END
```

Read. The `read` operation returns the content of a chunk. A storage server receiving a read request first checks if the chunk is stored locally. If so, it simply returns the data. Otherwise, it has to find out whether the missing chunk corresponds to a gap, or is outside the current file boundaries. As file sizes grow monotonously, $gmax$ always refers to a chunk number that defines a lower bound for the file size. If the chunk number is smaller, it is therefore safe to say that the missing chunk is inside the file boundaries and refers to a gap. Otherwise, it is necessary to ensure that $gmax$ in fact refers to the last chunk of the file, as $gmax$ might be outdated due to the fact that update messages may have been lost. This is done by retrieving $gmax$ values from all storage servers holding chunks of the file via RPCs and replacing $gmax$ with the largest such value. Finally, the chunk number is compared to the previously synchronized $gmax$; any larger chunk number refers to an EOF, whereas any smaller chunk number refers to a gap.

```
BEGIN read(f, n)
   IF  c_n  ∈  C_{s,f}  THEN
      – chunk exists locally → return chunk
      RETURN c_n
   ELSE IF  n  <  gmax_{s,f}  THEN
      – chunk does not exist locally, but chunk with higher
      – numbers than n definitely exist → gap, return padded
      – chunk
      RETURN zero-filled chunk
   ELSE
      – not clear if chunk with higher numbers than n exist:
      – initiate a broadcast to update gmax_{s,f}
      SEND GET_GMAX(f)  TO  S_f
      M  ←  RECEIVE all responses FROM S_f
```

```
      gmax_{s,f}  ←  max{m,  m ∈ M}
      IF  n  >  gmax_{s,f}  THEN
         – the chunk is the last one → EOF
         RETURN EOF
      ELSE
         – the chunk is not the last one → gap, return zero-
         – filled chunk
         RETURN zero-filled chunk
      END IF
   END IF
END

UPON GET_GMAX(f)
BEGIN
   RETURN  gmax_{s,f}
END
```

Stat. The `stat` operation returns all metadata that is associated with a file, which includes information about the file's size. This file size can be determined by fetching $gmax$ values from all storage servers and selecting the largest of them. This procedure can be avoided by introducing a designated metadata server that is also responsible for tracking file sizes. Such a metadata server can cache the current file size and serve `stat` requests from its cache. Depending on the required consistency between the file size returned by `stat` and the actual end of file, it can be updated with $gmax$ values either synchronously with each write request, in certain periods, or when the file is closed.

3.3 Arbitrary File Size Changes

As part of the POSIX interface definition, the `truncate` operation allows a process to explicitly set the size of a file to an arbitrary value. Such an operation violates our assumption that file sizes increase monotonously. The problem can be solved by maintaining a truncate operation counter for each file. We refer to such a counter as a *truncate epoch*. The necessary enhancements to the protocol can be sketched as follows:

- For each file, one particular storage server (*"truncate server"*) is designated for the execution of truncate operations. This ensures an ordering of multiple concurrently initiated truncate operations.

- In addition to the number of the last chunk, $gmax$ is enhanced by a *truncate epoch*.

- Each time a truncate operation is invoked, the truncate server increments its *truncate epoch* and propagates updates for the new $gmax$ to all storage servers. In contrast to the write operation, it waits for acknowledgments from all servers. This ensures that after the truncate operation has completed, all servers are aware of the new *truncate epoch*.

- When receiving a $gmax$ update, the local $gmax$ is replaced either if the received *truncate epoch* is greater than the locally known one, or both epochs are equal and the number of the last known chunk has increased.

The protocol enhancement ensures that truncate epochs as well as file sizes referring to a certain truncate epoch grow monotonously. This relationship allows for a correct ordering of $gmax$ updates between storage servers (and a metadata server, if necessary), which preserves the guarantees our protocol offers with respect to gaps and EOFs. Since truncate operations occur rather infrequently [1, 6, 7], it is acceptable to execute them in a coordinated fashion.

4 Evaluation

We demonstrate the scalability and the characteristics of our protocol with two experiments.

The first experiment investigates the read and write performance. We show the throughput observed by a single client when reading and writing a 4GB file using one to 29 storage servers.

In a second experiment, we elicit the corner cases; we compare the duration of a regular read, the read of a gap and the read beyond the end-of-file.

We implemented our striping protocol as part of XtreemFS [4], a distributed, parallel, object-based file system [5, 3]. File system metadata is stored on a dedicated metadata server, and chunks are stored on a set of object storage devices (OSDs). A client component translates application requests into sequences of interactions with the metadata server and the OSDs.

Setup. For both experiments, we used a cluster of 30 nodes connected via 4xDDR InfiniBand. Each node has two 2.5GHz Xeon CPUs with four cores, 16GB RAM, and a local 80GB SATA hard disk.

We measured the hardware limits of the network and hard disk with Iperf and IOzone, respectively. Iperf reported a maximum TCP throughput between any two nodes of approx. 1220 MB/s. For the local hard disks, IOzone measured a read performance around 57 MB/s and a write performance of 55 MB/s, when using synchronous writes and direct I/O for reads.

The OSDs do not cache objects and use synchronous writes and direct I/O to circumvent the system's page cache.

Experiment 1: Scalability. We used a client application that first writes and then reads a 4GB file linearly
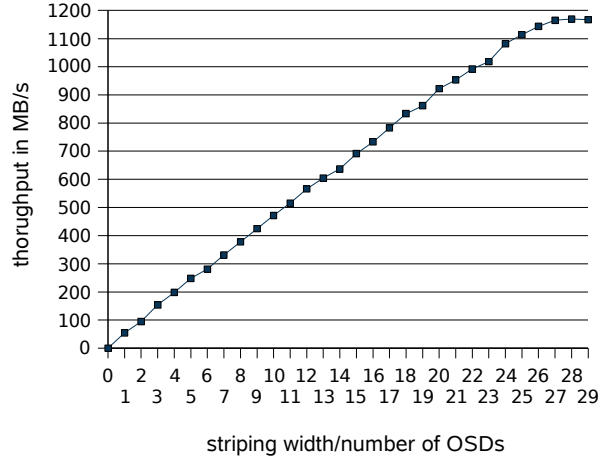

Figure 4: Throughput for writing a 4GB file from a single client on one to 29 OSDs.

with a chunk size of 1 MB. The application uses asynchronous writes and read-ahead to achieve maximum throughput with a single thread.

The throughput for writing a 4 GB file from a single client node onto one to 29 OSDs is shown in Figure 4. The throughput with a single OSD is approx. 40 MB/s and is limited by the maximum bandwidth of the local hard disk. The maximum throughput of the network is reached with 27 OSDs; as expected, adding more OSDs does not further increase the throughput.

Reading (Fig. 5) shows a similar linear increase until the maximum read throughput of the client application is reached at approx. 740 MB/s. As the limiting factor, we identified our single threaded HTTP parser which was not able to parse all responses with the maximum throughput of the network.

The experiment verified that our protocol and implementation scale with an increasing number of OSDs.

Experiment 2: Request Duration. A second client application is used which first writes chunks no. 1 and 3 and then requests data for chunks no. 2 (a gap), 3 (regular data chunk) and 4 (beyond EOF). We measured the duration of these read requests without striping and with striping over 5, 15 and 25 OSDs.

The results in Figure 6 illustrate the characteristics of our protocol as expected. Reads of gaps and of regular chunks take approximately the same amount of time. Reading beyond the end-of-file takes the longer the more OSDs have to be queried for the file size, since overhead is caused by sending $gmax$ RPCs to a growing number of OSDs.

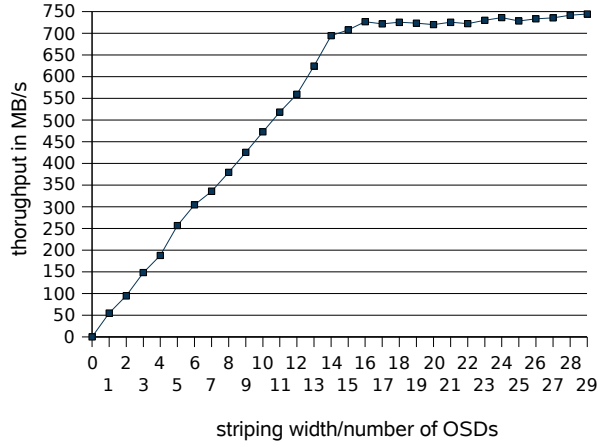Each request contains a list of all OSDs used for strip-

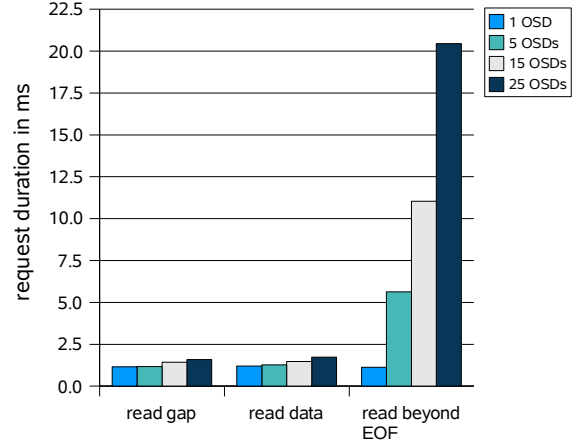Figure 5: Throughput for reading a 4GB file from one to 29 OSDs with a single client.

Figure 6: Duration of reading data, a gap or beyond the end-of-file on one, five, 15 and 25 OSDs.

ing. This information is necessary so that an OSD can contact all other OSDs to query them for the file size. The longer the list, the more addresses of OSDs have to be parsed and resolved. This explains the slight increase in the duration of regular reads of both gaps and data with a growing number of OSDs.

5 Related Work

The problem of ensuring a consistent notion of a file's size in presence of gaps and concurrent access is common to all parallel file systems and an important element of POSIX compliance. However, the available literature is light on details on how this problem has been handled in various file systems.

GPFS [8] is a block-based cluster file system that coordinates parallel and concurrent access to shared disks (such as a SAN). Because of the passive storage devices, clients are key elements of the architecture and participate actively in coordinating concurrent accesses. For each file, one of the clients acts as a dedicated *metanode* and is exclusively allowed to change the file's metadata (that includes the file size) on the shared disks.

Theses file size updates are tightly coupled with file locks to ensure a consistent view on the file size. Normally, all clients have a shared write lock and are allowed to change the file. Any changes that affect the file size are cached locally and sent to the metanode periodically. If a client operation relies on a correct file size, i.e. a `stat` call or a read past the EOF, the client has to ask the metanode for the definitive file size. The metanode in turn revokes the shared write lock, which triggers all other clients to send their pending file size updates. The

metanode can now infer the correct file size, answer the query and allow the client to proceed.

With this protocol, GPFS is able to guarantee POSIX compliant behavior. In addition to the query for the current file size, file-size sensitive changes require a presumably complex lock revocation procedure.

The Lustre file system is structured as an object-based architecture. Its storage servers take an active role in the file system architecture and contain dedicated disks. Unfortunately, its handling of file size-relevant updates is not well covered in the literature. The existing documentation [10] suggests that the metadata server hands out EOF locks to clients that convey the right to define the file size. When a storage server needs file size information, it checks whether one of its client holds the EOF lock and asks it for the file size. If no such lock exists, the information in the metadata server is up-to-date and the storage servers resort to its information. Like GPFS, Lustre implements POSIX compliant handling of file size and couples it with a lock mechanism. The responsibility of a single client for defining the end of file can become a performance bottleneck.

PVFS [2] is a parallel file system for high-performance computing applications. Its file size handling is not well documented either. It seems that PVFS clients decide locally whether unfulfilled read requests mean a gap or the end of file. The client first estimates a local minimum file size, e.g. by checking for existing data in subsequent objects. If that fails, it queries the metadata server for an up-to-date file size. The problem of outdated information on the metadata server seems not to be handled. With this protocol, PVFS might not be able to give the guarantees that are required by POSIX.

CEPH [12] is an object-based parallel file system. The file size, along with other metadata is separated from file content and stored on a dedicated metadata server. Because file data and metadata are separated, changes to the file size have to be propagated from the storage servers to the metadata server. When opening the file, a CEPH client retrieves the current file size from the metadata server and caches it. Any file size changes are reported back to the metadata server when the client closes the file. All changes to the file size that take place between opening and closing a file are applied to the cached value, so the client can tell an EOF apart from a gap by checking the cache.

With respect to gaps and EOFs, correctness is only guaranteed as long as no more than one client at a time has an open state for the file. Thus, developers must ensure that a file is properly closed before being opened by another process. Enforcing such a behavior to an application does not comply with POSIX file system semantics.

DPFS [9] is a parallel file system that supports different striping patterns. It uses an SQL database for storing the metadata. The transaction mechanism of the SQL database is used to handle all concurrency issues, including the updates of file sizes, and is likely to limit the performance.

6 Conclusion

We have presented a light-weight protocol that coordinates file size updates among storage servers in a parallel file system. The protocol ensures POSIX-compliant behavior in the face of concurrent accesses to a sparse file by allowing storage servers to distinguish between gaps and the end of file. With this protocol, we have tried to show that POSIX compliance for sparse files is possible with a relatively simple mechanism. Our experiments show that the protocol allows a parallel file system to scale freely without sacrificing performance or access semantics.

The protocol does not make use of any client-side locks that are usually required for a client-side caching mechanism. As these locks restrict the degree of access parallelism on a file for the sake of reaching a better performance on a single client, we assume that the striping protocol can be further optimized for this case.

Acknowledgments

References

[1] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. *SIGOPS Oper. Syst. Rev.*, 25(5):198–212, 1991.

[2] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, 2000. USENIX Association.

[3] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran. Object storage: The future building block for storage systems. In *2nd International IEEE Symposium on Mass Storage Systems and Technologies*, 2005.

[4] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario. XtreemFS: a case for object-based storage in Grid data management. In *3rd VLDB Workshop on Data Management in Grids, co-located with VLDB 2007*, 2007.

[5] M. Mesnier, G. Ganger, and E. Riedel. Object-based storage. *IEEE Communications Magazine*, 8:84–90, 2003.

[6] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A trace-driven analysis of the unix 4.2 bsd file system. *SIGOPS Oper. Syst. Rev.*, 19(5):15–24, 1985.

[7] D. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Conference*, pages 41–54, June 2000.

[8] F. Schmuck and R. Haskin. Gpfs: A shared-disk file system for large computing clusters. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, page 19, Berkeley, CA, USA, 2002. USENIX Association.

[9] X. Shen and A. Choudhary. A high-performance distributed parallel file system for data-intensive computations. *J. Parallel Distrib. Comput.*, 64(10):1157–1167, 2004.

[10] Sun Microsystems, Inc. The Lustre Operations Manual, 2008.

[11] The Open Group. The Single Unix Specification, Version 3.

[12] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI*, pages 307–320, 2006.