COFS: A Tool for File System Virtualization

Abstract

Current file systems are not adequate to handle efficiently all kinds of loads: a parallel file system like GPFS can span across a high number of nodes providing a POSIX interface, but may behave poorly with small files or big directories with high inter-node sharing ratios; PVFS2 offers good performance for shared access, but only for MPI-IO applications; Hadoop DFS offers efficient streaming access to large files, but has a relaxed semantics and offers almost no support for shared updates. . . As no single file system fits all needs, data centers usually end up with more than one file system type, or at least with more than one file system configuration (with different number of servers, block and cache sizes, redundancy policies, etc.)

The multiplicity of file systems and the exposure of their internal structure and layout pushes the responsibility of placing the files in the right place onto the users who, instead, should be able to organize data in a sensible way, regardless of low level features or resources. To solve this situation, this paper proposes using a virtualization layer between the user and the raw file system(s) that is able to take sensible decisions on where and how files should be placed to take advantage of underlying file system(s) features. We present COFS (COmposite File System) as a framework providing this virtualization. Finally, we also present and evaluate an example of use that boosts the performance of an existing parallel file system through the virtualization of its name space.

1 Introduction

High-performance computing is rapidly developing into large aggregations of computing elements in the form of clusters or grids. In the last few years, the size of such distributed systems has increased from tens of nodes to thousands of nodes, and the number is still raising; consequently, there is also a need for parallel file systems able to provide a sustained flux of data to the distributed computing elements.

At the same time, there is a widening range of scientific and commercial applications seeking to exploit these new computing facilities. The requirements of such applications are also heterogeneous, usually leading to dissimilar patterns of use of the underlying file systems. This trend is favored by the increasing number of high-end computing facilities that aim to be "general purpose", in contrast with the more focused systems designed to solve specific problems or run homogeneous workloads that, for example, populated the high ranks of the Top500 [3] list just a few years ago.

Nevertheless, typical distributed and parallel file systems are heavily oriented to provide performance for workloads with specific characteristics (reduced or low scale write shares, supporting mpi-io applications, streaming data, etc.), jeopardizing the ability to adapt to varying use patterns from different applications. They may also introduce additional restrictions which depend on the characteristics of the supporting platform (cluster size, interconnection network, number of servers, number of clients, etc.); the file system has then to be finely tuned (by means of cache sizes, timeouts, etc.) and decisions taken through these settings may alter factors such as the optimum number of entries in a directory, the maximum concurrency level when accessing the name space hierarchy or the most adequate pattern to access files.

Computation centers try to support the distinct requirements by providing several file systems (or at least different file system configurations), which are traditionally mounted on separate branches of a directory tree. The burden of file system complexities and restrictions is then explicitly passed to the user, who must keep in mind the nuts and bolts of each specific file system and its configuration in order to achieve good results. In particular, a user must:

- Know how the available file systems are organized, which services/features are available in each of them (block sizes, automatic backup services, high availability, etc.) and to which branch of the directory hierarchy each file system is attached.

- Explicitly choose which file system to use by placing the files in the corresponding branch of the directory tree (which may not correspond with a logical/functional/sensible organization of the files from a user's perspective.)

- Make sure that each file system instance is used according to specific rules (e.g. 100,000 files in one directory may be fine on a certain file system, but it may crash the whole system on a different one.)

Our proposal consists of using virtual layers to prevent the exposition of the file system internal structure, freeing the final user from the burden of handling it by interposing intelligent modules capable to take advantage of the underlying file system features. Our goal is to break the ties between the user-visible hierarchical organization and the underlying file system structure, not only by providing independence from the storage location of data blocks, but also by transparently re-structuring the actual directory and i-node information.

In order to demonstrate the feasibility of adding a new layer above raw file systems without harming the performance, as well as having a testbed for testing placement policies, we have developed the COmposite File System (COFS): a proof-of-concept framework based on decoupling the file system metadata and the name space hierarchy from the data handling, and making a heavy use of union file system concepts. As an application example, we have deployed the COFS framework on our test cluster, and have used it to boost the performance of its native parallel file system by transparently re-arranging the user file hierarchy without altering the user view.

The main contributions of this paper can be summarized as:

1. Proposing the use of virtualization techniques to address different file system issues and limitations, based on fully decoupling the metadata and name space from the actual file system structure.

2. Implementing a proof-of-concept framework and evaluating its ability to boost the performance of a parallel file system as a case study.

The rest of this paper is organized as follows: next section summarizes related work; in section 3 we discuss our working hypothesis and the potential applications of our virtual file system model; section 4 describe the implementation details of the COFS framework; a case study and performance results are shown in sections 5 and 6; finally, we outline our plans for future work.

2 Related work

It is widely known that current parallel and distributed file systems constitute a potential bottleneck for the high I/O demands of the applications running on large-scale computing clusters. Given the complexity of file systems, a lot of work has been done to address different issues from different perspectives. In this section we try to summarize some of the developments that constitute the foundation of our present work.

2.1 File system specialization

Modern file systems try to adapt to the new demands by means of specialization. These optimizations allow to achieve efficiency for a given set of workloads, but there is a cost in terms of lack of performance for non-optimized cases, or changes in semantics that make the system inadequate for non-targetted environments. In the following paragraphs we mention a few examples of these specializations and the corresponding trade-offs.

The Hadoop Distributed File System [6] is designed for streaming access to large data sets and fault tolerance; to this end, they use a very relaxed consistency model, assuming that files are rarely changed once written, which makes it inadequate for simultaneous parallel write accesses. The closely related Google File System [13] is also highly specialized on sequential accesses and "append-only" modifications.

On the other hand, "parallel" file systems do provide mechanisms for simultaneous parallel access (including writing) but, even then, they tend to specialize and favor specific workloads.

IBM's GPFS [22] provides POSIX semantics across very large-scale clusters and is specially optimized for large, contiguous I/O operations; nevertheless, the complexity of the coherence mechanisms may hinder the performance for high ratios of write sharing.

Lustre [7] also offers a POSIX interface and tries to simplify coherence handling by centralizing metadata management. It achieves good behavior for parallel metadata operations and distributed I/O requests; on the other hand, some studies show that its data striping policies are not adequate for MPI-IO implementations [30].

Finally, PVFS2 [4,8] has evolved to overcome some of the parallel performance issues by going further into specialization and focusing on specific parallel access I/O models such as MPI-IO; as a counterpart, it offers a relaxed semantics, and performance is poor for POSIX-like parallel operations.

Recently, some studies tried to experimentally determine the strong and weak points of different parallel file systems and how they behave under different workloads [9, 23]. Another evaluation, focused on PVFS2, aimed to isolate the different causes of performance losses and determine their impact [16]. Additionally, efforts are being directed to mitigate some of the issues and limitations, at least for specific cases (e.g. hierarchical striping for Lustre [30] or specific file creation strategies for parallel I/O [10].)

Our proposal aims to combine the different optimizations by transparently redirecting requests to the file system which is most suitable for the current needs. For cases where no single file system fits well, our COFS framework also allows combining different file systems or transforming the requests using solutions designed for specific situations.

2.2 Global and virtual name spaces

Most of current file systems offer a "global" name space, where "global" means that each object has an identifier which is valid across the whole system and can be used to refer to that object from any place. Usually, this concept is mapped into a unique hierarchical directory tree where files are grouped inside directories and identified by its directory "path" from the root and a unique name inside the directory.

This view of a "path" from a common root is also useful when data spans across several storage nodes, partitions or even file systems.This unified hierarchy is sometimes refereed to as a "virtual" name space.

IBM's GPFS [22] distributes the name space-related metadata across several nodes much in the way it distributes data files (directories are essentially treated as special files). Other file systems, such as Lustre [7], Ceph [26] or PVFS2 [4] decouple metadata and name space management from data accesses, having specialized services and distinct storage mechanisms for name space handling.

Mechanisms to maintain name space coherence also vary with file systems: GPFS uses distributed locking techniques; PVFS2 prevents consistency problems by distributing metadata with a "no shared data, no cached data" policy [23]; and Lustre uses a single metadata server to avoid conflicts. Panasas [28] (closely related to Lustre) organizes the directory tree into separated "volumes", having a single metadata server per volume. Volumes appear as directories in the name space root, which act as "mount points" in a classical Unix file system.

The approach used by ONTAP GX [12] is also based on volumes; nevertheless, they tend to be small and the "mount points" (called *junctions*) may be located anywhere on the name space. Additionally, a virtualization layer makes easy to physically re-locate or replicate a whole volume for performance or capacity reasons, in a transparent way from the user's point of view.

The file systems mentioned above use explicit partitioning via name space: when a user decides to place files in a directory path crossing a certain mount point or junction, all those files will essentially be handled in the same way, regardless of how they are to be used.

An alternative to the explicit name space partitioning is based on stackable file systems. Stackable file systems are a well-known technology that can be used to extend the functionality of a file system [31]. In particular, fan-out stackable file systems (or *union* file systems) can combine the contents of several directories (possibly in different file systems) into a unified"virtual" view [20, 21, 29]. Essentially, each operation is forwarded to the corresponding objects in all the underlying file systems and the result is a sensible composition of the operation results for each layer, including file attributes and other metadata.

RAIF [15] uses this mechanism to combine several file systems (*branches*) into a single virtual name space. Internally, the directory structure is replicated on all branches, and regular files are placed (possibly striped) in specific branches according a set of rules matched against file names. In other respects, RAIF is very tied to the low-level underlying file systems for conventional file metadata management, so it may suffer from operations that require synchronous accesses to all branches.

Our proposal combines several aspects of the different approaches: explicit file system boundaries are avoided (i.e. no mount points: any file in any part of the name space can be on any file system), and metadata and name space management are decoupled from data handling (as in PVFS2 or Lustre), but also from the underlying file systems (no need to replicate the user view of the directory hierarchy into the the low-level file systems.)

2.3 Other file system virtualizations

The use of virtualization techniques applied to file systems is not a new topic. They have been used at different levels of the file system to hide complexities and facilitate the storage management.

Acting on the lowest device level, Parallax [25] uses a virtual machine to offer a block-based interface hiding a distributed storage environment. In a similar way, Peabody [14] uses a software-based iSCSI target to provide virtual disk images that a local file system can then use as a backing store.

At the infrastructure level, ONTAP GX [12] virtualizes the file system servers and network interfaces used to access the file system data, offering a single virtual large server with multiple interfaces and allowing transparent reprovisioning of physical servers.

Virtualization is also used at the name space level. ONTAP GX provides a virtual layer allowing volumes (directory subtrees) to be transparently re-located or replicated. On the other hand, RAIF [15] uses virtualization to divert the target file system for individual files, though the directory tree itself is not virtualized, but directly mapped into the underlying file systems.

The prototype presented in this paper (COFS) exploits virtualization at the name space level, using single file granularity and virtualizing also the directory hierarchy. Working at file level makes it easier to determine use patterns and select the most appropriate layout for the file. Of course, this does not coerce the use of additional virtual layers by the low level file underlying file systems.

3 Rationale

3.1 The price of optimizations

There is a widening gap between the computing power of current large scale platforms and the performance that file systems deliver. Current trends in file systems try to overcome this gap by optimizing the file system internals and its configuration to satisfy the most common demands from the high level applications.

This has a price. Providing certain features or optimizing certain operations may strain the requirements on other components beyond practical limits (e.g. a directory structure may be designed to contain an extremely large number of entries; however, if strictly consistent parallel accesses is required, then the cost of coherence handling may penalize such large directories.) As a result, optimized file systems bring up a set of restricted conditions under which they can operate with efficacy. Such file systems may be very efficient in their domains, but performance drops when limits are forced.

On the other hand, the spread and availability of large computing facilities has increased the diversity of applications targetted to run on them, bringing different I/O needs; so, each time is more difficult for a single file system to match all needs. At the end, some applications end up paying the price of features and optimizations that they cannot use [23].

3.2 Dealing with multiple file systems

The first approach to solve this situation consists of providing several file systems with different settings on the same computing center. This has several drawbacks:

1. The application must be adapted to a specific computing facility and moving it will have a higher cost (as other computing premises may use different file systems configured in different ways.)

2. The user should know the low level details of the system (this may be arbitrarily complex, ranging from optimum sizes for reads and writes up to how many directory entries fit in a cache block.)

3. Changes in the computing facility (reprovisioning, upgrades, etc.) may alter the optimum file system parameters, requiring a new optimization process.

4. Last but not least, the user may not have the opportunity, the capability or the time to carry out such adaptations.

The final cause of these issues is that the internal structures of file systems are exposed to the application level. While users think in terms of abstractions like files logically organized in hierarchical trees, file systems have a quite rigid map between such abstractions and the actual data layout; as a consequence, small changes in logical data organization may have significant (and possibly negative) impact on performance.

3.3 Advantages of high-level virtualization

Our proposal consists of taking advantage of a virtualization layer on top of conventional (and maybe optimized) file systems. Even if some control over physical layout is apparently lost, the multiplicity of layers of current file systems mitigates the negative effects [24] and may even improve some results [15].

We strongly believe that decoupling the file system metadata and name space from the actual data layout is a key factor. Metadata management at low-level file system layers is usually heavily tied to the low level data layout, aiming to efficiency; nevertheless, this makes the inner limitations and complexities visible to the applications.

On the contrary, we consider that the metadata and name service must provide a convenient view of the file system for the user, and transparently convert user requests into the appropriate low-level file system operations, adapting them to the advantages and limitations of the underlying file systems. The building blocks we plan to use are:

- Virtualization of the file system by decoupling the metadata and the name space hierarchy from the actual file layout in the low-level file system(s).

- File system composition to combine underlying file systems with different features and capabilities, unifying them under a common layer.

It is important to note that we do not intend to build a new file system from scratch, but to prove that it is possible to increase the overall performance by leveraging existing file systems and make a combined use of their strong points while mitigating the disadvantages.

Some concrete areas that may benefit from these virtualization techniques are:

- **Bypassing the limitations of current file systems.** A virtualization module can be aware of the optimization trade-offs of a particular file system. Thus, it can translate application needs into actual layouts which are able to take advantage from optimizations and tuned parameters. Being this transparent, the administrators have the freedom to change or adapt configurations without affecting application performance. As an example, a virtualization module could split a large user directory into smaller ones so that the number of entries is large enough to fill a data block (to improve cache usage), but small enough to prevent false-sharing conflicts and avoiding unnecessary synchronization traffic (handling low-level details that the user does not need to be aware of.) In a similar way, other improvements such as hierarchical striping [30] could be transparently applied when needed.
- **Automatically mapping files into the most adequate file system.** For data centers with several file systems it would be possible to use an intelligent module in the virtualization layer to decide the best location for each file: the user may have his own tree structure (which does not need to take into account file properties and intended usage) and the files will be transparently located in the file system that is best for them. Prediction algorithms and other heuristics can be used to guess the requirements of newly created files, using the name, logical location, the user or the job that has created the file as inputs for the prediction algorithms.
- **Replicating files into different file systems.** Virtualization at file system level can also be used to maintain replicas of the same data into different file systems, both for availability and load balancing [15]. Additionally, the replication mechanism can be used to provide extended features and allow different use patterns. For example, a large multimedia file can be located in a parallel file system, allowing efficient access from a distributed application while it is being generated and/or modified, and be replicated to a streaming file system to allow fast streaming sequential access for visualization.
- **Write off-loading to improve performance and bandwidth utilization.** It is frequent to see that writes from many clients (possibly to different files) are done simultaneously in burst. This may cause some servers saturate and prevent them from offering the desired performance. In this situation, write off-loading at block level has been proposed as a technique for both keeping the performance and reducing the power consumption [17]: block writes are temporarily diverted to a different server until the real target recovers its capacity. Using file system level virtualization, the same technique can be applied across file systems: when peaks occur, they could be detected and files would be written to a different file system. There are two main advantages: first, the pressure on the saturated file system is reduced, so it can recover faster; second, by using additional file system servers, storage-related network traffic has more chances to use a bigger portion of the available bandwidth, possibly reducing the overall response time.
- **Extended features.** Having a decoupled metadata and name service allows for feature extensions that can be activated per-file (and not globally at file-system level). Such extensions may include security and cryptographic modules, versioning, different coherence models, etc. Virtualization can provide these specific services by simply diverting the specific file to a file system supporting the desired features, or by having a module implementing them at high level (e.g. replication could be provided by a low-level file system using RAID mechanisms, or by the virtualization layer by diverting writes to two or more different file systems.)

In the following sections we present our framework to explore these possibilities. Our performance results focus on the first item: using virtualization to improve the behavior of an underlying file system for non-optimum workloads. The COFS prototype shows that the potential benefits are higher than the costs of virtualization.

4 COFS implementation

We have developed the "Composite File System" (COFS), a proof-of-concept prototype aimed towards the virtualization of file systems which helps us understanding how the file systems behave at low level, and provides a framework for testing and evaluating our ideas. Its main goals are:

- Decoupling the user view of file hierarchy from the actual layout, so the latter can be optimized for the underlying file system(s).
- Being able to divert files from the same directory into different file systems, transparently to the user. This, together with the previous item, provides a virtualized global name space.

Additionally, COFS has been conceived as a tool allowing us to track operations at file system interface (to obtain a detailed trace of the system behavior), to explore different mechanisms for metadata handling and assessing its impact on performance, and to test different policies and parameters for laying out data in the underlying file system(s).

Despite being a proof-of-concept prototype, having a reasonable performance is a major requirement, as we

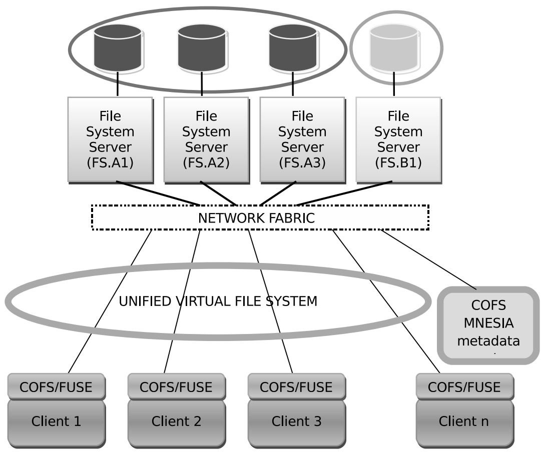

File System Server (FS.A1)
File System Server (FS.A2)
File System Server (FS.A3)
File System Server (FS.B1)
NETWORK FABRIC
UNIFIED VIRTUAL FILE SYSTEM
COFS MNESIA metadata
COFS/FUSE
Client 1
COFS/FUSE
Client 2
COFS/FUSE
Client 3
COFS/FUSE
Client n

Figure 1: A multi-file system architecture, augmented with COFS-based virtualized file system

pretend to show that the mechanism does not have a significant overhead and, on the contrary, may boost performance in several situations.

Also, POSIX compliance was a strong design requirement. Apart from the fact that this is still the dominant model for most applications, our goal was reducing the operating restrictions of underlying file systems; so, limiting the semantics was not an option: if POSIX semantics is required and the underlying file system supports it, then COFS should also be able to deal with it. The current implementation fully supports POSIX except for some functionalities which were not relevant for our present work (namely named pipes.)

It is important to mention that the framework does not directly deal with data. There is no management of disks, blocks or storage objects: COFS simply forwards data requests to the underlying file systems and indicates an appropriate low level path when a file is created. Then it is up to the underlying file system to take the decisions on low-level data server selection, striping, block/object placement, etc. In this sense, COFS is not a complete file system, but a tool to leverage the capabilities of underlying file systems.

On the contrary, COFS does take the responsibility on high level metadata management. By metadata we specifically mean access control (owner, group and related access permissions), symbolic and hard link management, directory management (both the hierarchy and the individual entries) and size and time data for non-regular files (sizes and access time management for regular files rely on the underlying file system.)

4.1 Architecture

Fig. 1 shows an example of how the COFS prototype integrates in a multi-file system environment. File system A is served by 3 file servers, while file system B has a single file server front-end; *n* clients are able to contact the servers through the network and, in a typical setup, they would mount A and B on separate branches of their local directory trees.

COFS introduces an extra layer on each client providing a unified virtual view of the multiple file systems, while metadata information is handled by an additional node. It is important to mention that even though we use a single metadata server in the current stage of implementation, this is not forced by design, and the framework also admits a distributed metadata service (we will be back on this issue later.)

The COFS layer on each node offers a file system interface, so it can be mounted as any other file system. The implementation is based on FUSE (Filesystem in USErspace [2]), which provides a kernel module that exports VFS-like callbacks to user-space applications. The decision to use FUSE was driven by both portability and level of support, as well as easy of development. FUSE is a standard component of current linux kernels (also available for other operating systems) and provides a stable platform for implementing a fully functional file system in a userspace program. Considering our experimental goals, the downside of missing some kernel-level information that is not exported or forwarded to user level, and possibly minor efficiency losses, is largely compensated by having a drop-in environment that can be used in most linux boxes without requiring specific kernel modifications.

Once intercepted by FUSE, file system requests are internally diverted by COFS into two different modules (the data and metadata drivers) with well-defined interfaces (Fig. 2). The data driver is responsible for mapping

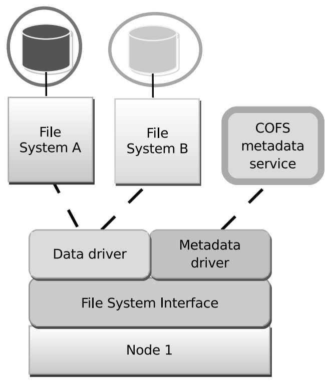


Figure 2: Detail of COFS client architecture and interfaces

the regular files into the underlying file system(s), while the metadata driver takes care of hard and symbolic links, directories, and generic attributes. Some operations need the collaboration of both drivers: for example, creating a file involves creating an actual regular file on a convenient location (a data driver responsibility) and updating the proper entries in the directory (done by the metadata driver). So, an interface is also defined for communication between both drivers.

4.2 The data driver

The COFS modular design allows us to easily test different policies for both data and metadata handling. By now, we have two different drivers for the data module: the *sparse* driver and the *null* driver (which redirects all data-related requests to `/dev/null` and is used for testing and evaluation purposes.)

The *sparse* driver implements a very simple policy to re-organize the user's file hierarchy into something adequate to take advantage of the underlying file system. When a new file is created, its actual location is not the path indicated by the user, but an automatically generated path that depends on the node issuing the create request, the parent directory in the user's view, and the process creating the file.

Several parameters can be tuned for the *sparse* driver: the depth of the actual directory tree, the maximum number of files and subdirectories in an actual directory, and a randomization factor. The entries per directory affect the way in which caches can be exploited, while the tree depth can mitigate false-sharing accesses to the same directory. The randomization factor can help to balance the pressure in situations where a single process creates lots of files that are then accessed by multiple clients. The overall result of the *sparse* policy is distributing the files across several directories, while keeping them loosely grouped by creator and proximity in the user's view.

4.3 The metadata service and driver

For COFS, we decided to take a conceptually centralized approach for metadata because of its simplicity. We dealt with scalability concerns by leveraging the well-know technology of distributed databases: metadata can be seen as small set of tables having information about the files and directories and, in case of need, it could be distributed into several servers by the database engine itself (without the need of explicit partitions.)

To this end, we chose the Mnesia database, which is part of the Erlang/OTP environment [1]. Mnesia provides a database environment optimized for simple queries in soft real time distributed environments (with built-in support for transactions and fault tolerance mechanisms). Additionally, the Erlang language has proven to be a good tool for fast prototyping of highly concurrent code (the language itself internally deals with thread synchronization and provides support for transparently distributing computations across several nodes.)

The current COFS prototype uses a single metadata server running an Erlang node with an instance of the Mnesia database. It also keeps the current working set of metadata information as a cache of active objects, using a concurrent caching mechanism similar to the one described by Jay Nelson [18]. The client's metadata driver simply forwards requests to the server, and handles metadata caching and leases.

5 Case study

One of the motivations of our work was the performance drop observed in some of our large production clusters. Such clusters have GPFS file systems, and an heterogeneous workload comprising both large parallel applications spanning across many nodes, and large amounts of relatively small jobs.

Our observation indicates that typical modus operandi ends up creating large amounts of files in the same directory. Large parallel applications usually create per-node auxiliary files, and generate checkpoints by having each participating node dumping its relevant data into a different file; not unlikely, applications place these files in a common directory. On the other hand, smaller applications are typically launched in large bunches, and users configure them to write the different output files in a shared directory, creating something similar to a file-based results database; the overall access pattern is similar to that from a parallel application: lots of files are being created in parallel from a large number of nodes in a single shared directory.

Very large directories, specially when populated in parallel, require GPFS to use a complex and costly locking mechanism to guarantee the consistency, resulting in far-from-optimal performance. The overhead is not limited to the "infringing" applications, but affects the whole system, as file servers are busy with synchronization and all file system requests are delayed.

Virtualization techniques based on decoupling the name space from the actual low-level file system layout could mitigate the issue, offering large directory views to the user while internally splitting them to reduce the synchronization pressure on the GPFS servers.

Next section presents a preliminary evaluation of our COFS prototype in order to assess its ability to mitigate this issue and boost a single GPFS file system.

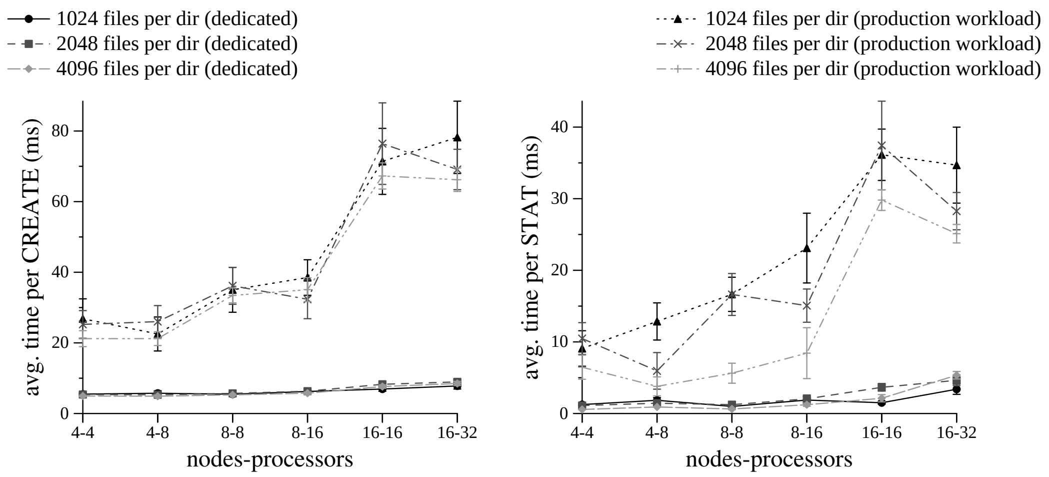


Figure 3: Effect of external workload on GPFS efficiency (with different scales)

6 Performance results

6.1 Experimental setup

Our test environment consisted of a cluster of 70 IBM JS20 blades, with 2 processors each (PPC970FX) and 4GB of RAM. Blades were distributed in several *blade centers*: each blade center had an internal 1Gb switch, and different blade centers were interconnected through non-uniform network links (so that available bandwidth is not the same for all blades.) We conducted our tests on a GPFS file system, based on two external Intel-based servers connected to the cluster by 1 Gb link each.

The current COFS prototype deals with metadata operations, and simply forwards data requests. So, we used *metarates* [5] as a benchmark. *Metarates* was developed by UCAR and the NCAR Scientific Computing Division, and measures the rate at which metadata transactions can be performed on a file system. It also measures aggregate transaction rates when multiple processes read or write data concurrently. We used this application to *create*, *stat*, *utime*, a number of files from the same directory in parallel. Directories were populated with different amounts of files (from 256 to 32,768) and accesses were done simultaneously from 4 to 64 nodes. As an addition to the original code, we also measure the time needed to *open* and *close* a file.

Our testbed cluster was not exclusively dedicated to benchmarking: unless explicitly noted, our measurements were done while the rest of the cluster was in production. This gave us the opportunity to see how the unrelated (though usual) workload affected the file system behavior. Even if we observed that the overall effect of the production workload is quite homogeneous along time, special care was taken to detect and avoid deviations due to particular situations. The methodology employed to obtain the measures minimized the impact of punctual variations of system behavior, getting acceptable statistical confidence levels.

6.2 Base system behavior

Understanding under which conditions a file system has a good performance, and which are the factors that negatively affect it, is the starting point for boosting it.

Our first experiment consisted of determining the effect of the production workload on the file system behavior. To that end, we executed the *metarates* benchmark using the cluster in dedicated mode, and then compared the results with the cluster shared with a production workload.

Fig. 3 shows the differences observed for *stat* and *create* operations with and without the production background. The *x* axis shows the number of nodes and processes used. It can be appreciated that the cluster activity reduces the file system performance by about an order of magnitude. Considering that such operations have a very small payload (only metadata information), we may speculate that a large portion of the delay is caused not by the bandwidth needed for the actual information, but by consistency-related traffic (even when each process works on a different set of files.) That assumption would give our virtualization system enough margin to obtain a good speed-up by reorganizing the file layout and reduce the synchronization needed among GPFS clients.

The performance drop when using 16 nodes (clearly visible for the create - Fig. 3) is due to the testbed network topology: when using up to 8 nodes, blades are located in "close" blade centers; for 16 nodes and above, some blades are allocated in "far" locations, so that the available bandwidth is shared with more nodes and the effect of possible interferences from the produc-

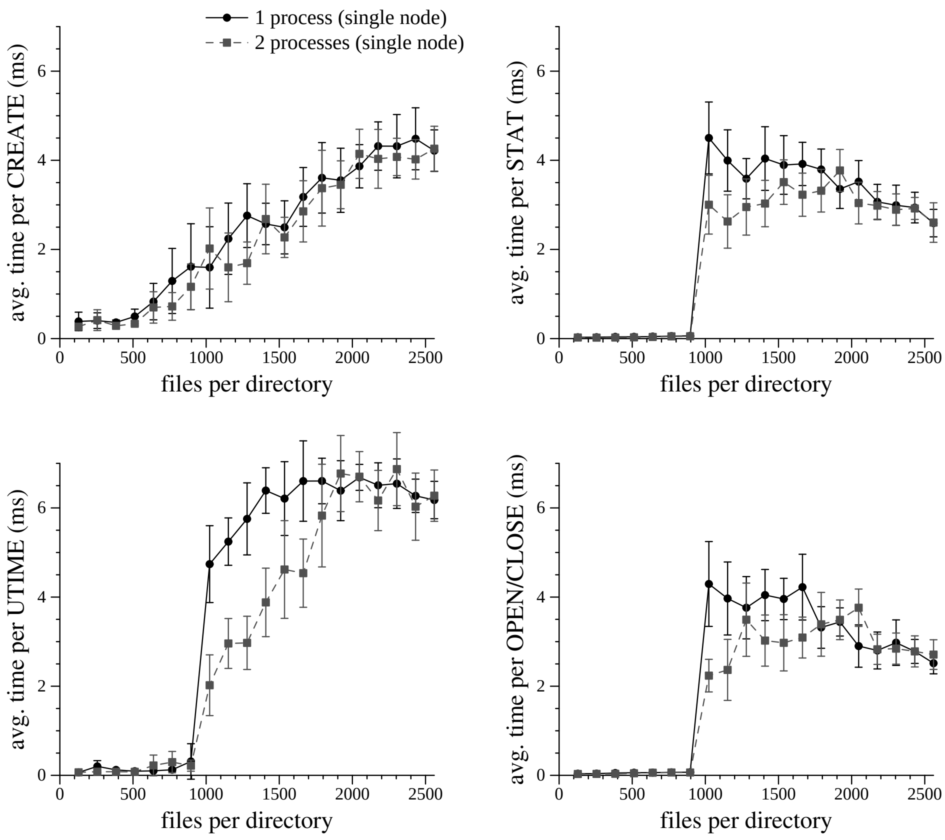


Figure 4: Effect of the number of entries in a directory on GPFS

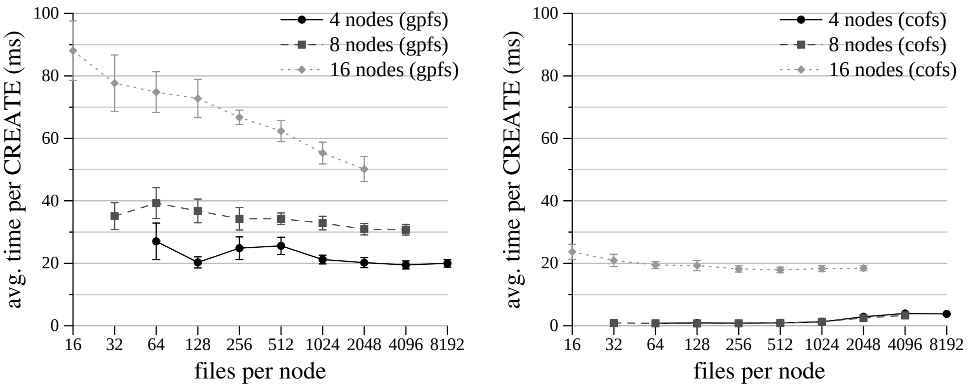
tion workload is more noticeable. Still, there is a big gap from the base-line of the dedicated executions that can be reduced by a more appropriate file layout.

Another interesting observation is how the *stat* time increases as the number of files in the directory decreases. Being GPFS a black box, we can speculate that this is related to distributed locking granularity: a *stat* response from the servers contains information about several entries to take advantage of bandwidth; but the less files we have the higher the probability of locking conflict between a larger number of nodes. Note that conflict-free access is not possible, because “the user” has decided to put all the files in the same directory and that forces the file system to handle a shared common structure (the directory.)

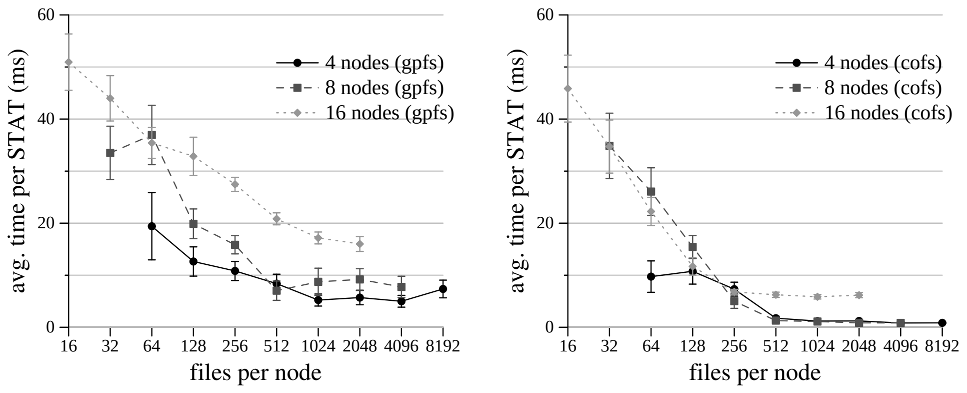
The behavior of the system for 2048 files per directory deserves a special comment, as the performance varies depending on the number of processes per node. Apparently, GPFS is able to coalesce the requests inside a client and this coalescing crosses a boundary that makes possible for GPFS to take advantage of the situation and improve the performance (reducing the inter-node conflicts, and being able to handle intra-node sharing efficiently). Ideally, a user could use this knowledge and tune an application to take advantage of it; however, parameters are closely related to specific datacenter configuration, so that user-specified per-application tuning would be impractical. On the contrary, a virtualization layer as the one provided by COFS may have a module with this kind of information and exploit it transparently.

Running the *metarates* benchmark on a single node reveals another boundary value: Fig. 4 shows that GPFS has an extremely good behavior for single node *stat*/*utime*/*open*/*close* operations on directories below 1024 entries (comparable to local file system rates.) We can speculate that this is caused by the ability of GPFS to delegate full control to clients under certain circumstances (e.g. single-node access and data present on local cache). Beyond that size, performance drops to network-compatible rates (though having two processes seems to slightly compensate - only up to 2048 entries). Note that the pattern is completely different for *create*: there is no local information to exploit (as we are creating new entries) and time per operation follows a steady linear increase above 512 entries.

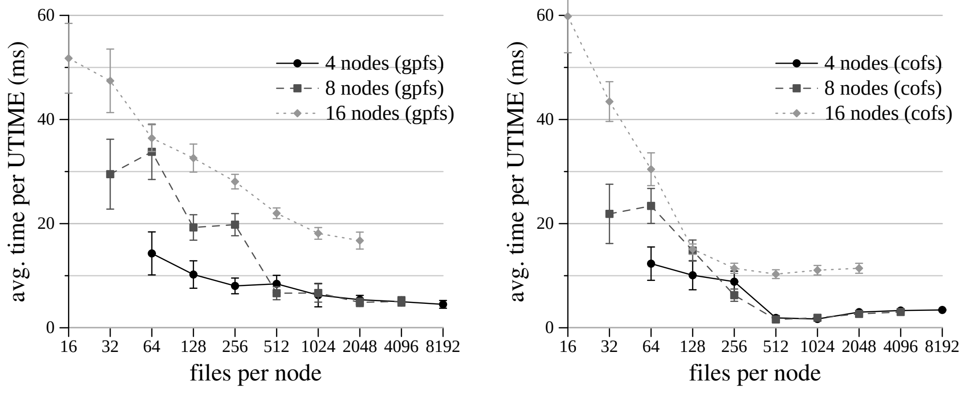
These observations inspired the COFS *sparse* data driver module described in section 4.2. Its goal is splitting large shared user directories into non-shared directories small enough to enable delegation and local access, so that GPFS can improve its performance. Next subsec-



(a) Create times (pure GPFS vs. COFS virtualization over GPFS)



(b) Stat times (pure GPFS vs. COFS virtualization over GPFS)



(c) Utime times (pure GPFS vs. COFS virtualization over GPFS)

Figure 5: Operation time vs. files per node accessed in a shared directory (*different scale for creation*)

tion shows the results obtained using the COFS virtualization layer to transparently alter the file system layout.

6.3 Virtualization results

Unless stated otherwise, COFS measurements in this section have been done using the *sparse* data driver, limiting the low level directory size to 512 entries, and using 1 random bit as randomization factor (that randomly distributes related entries into two separate directories.) We have also coalesced results per node in both COFS and GPFS (merging results using 1 and 2 processors per node) as we have observed that trends are determined by nodes as a whole, and not individual processors (slight differences exist: GPFS seems to be able to parallelize some requests when using 2 processors and take certain advantage, while the FUSE component in COFS apparently serializes them - nevertheless, differences are marginal compared with overall values.)

Fig. 5(a) shows the benefits of breaking the relationship between the virtual name space offered by the COFS framework (exporting a single shared directory to application level) and the actual layout of files in the underlying GPFS file system. By redistributing the entries into smaller low level directories, COFS allows GPFS to fully exploit its parallel capacity by converting a shared parallel workload into multiple local sections that do not require global synchronization.

The scaling overhead of moving from 4 to 8 nodes in GPFS disappears when COFS is used. For 16 nodes there is also a big improvement, due to the elimination of synchronization needs, but there is a performance barrier slightly below 20 ms. Some network measurements confirmed that this was not related to COFS itself, but to the particular topology of our testbed (as already mentioned in section 6.2.)

Fig. 5(b) and 5(c) shows the average time for *stat* and *utime* operations as a function of the number of files in a shared directory accessed by each node (figures for *open/close* operations are not shown as they closely resemble the *stat* figures for both GPFS and COFS.)

Overall, we can see that all the figures follow a similar pattern: there is a first phase of large operation times when only a few files per node are accessed, that converges when the number of files per node increases. It is worth noting that the stable times for *stat* and *utime* are considerably lower than *create* times; this is mainly due to an effective use of caches for larger number of files (which cannot be exploited for *create* operations.)

Times for *utime* operations are slightly higher than times for *stat* operations. One of the main reasons for these are i-node cache invalidations caused by the modification of data. On this respect, we must mention that we have observed noticeable false-sharing effects inside the i-node structure: some fields are rarely modified (type and permissions) but frequently used, and they are affected by modifications to more volatile fields (such as access times) which are not so often needed. Despite of this fact, i-nodes are usually handled and cached atomicly (including the current version of COFS - nevertheless, we plan to evaluate the effects of field-level caching in next versions of COFS.)

The COFS layer remarkably reduces the *stat* time when a directory grows beyond 512 entries per node (from approx. 7ms. to 1ms. for 8 nodes; and from 5ms. to 1ms. for 4 nodes). The variability caused by network noise is also eliminated, and Fig. 5(b) shows that COFS operation times for 4 and 8 nodes are nearly identical. Even for 16 nodes, where the network bandwidth is reduced, the speed-up for large directories is noticeable (from approx. 27ms. to 6 ms. per operation.)

The *utime* time is also improved with COFS (from about 5ms. to 1–3ms. for 4 and 8 nodes.) Again, the effect is clearer for 16 nodes (from approx. 18ms. to 11ms, as seen in Fig. 5(c).)

Below 128 operations per node, the COFS setup is only able to slightly reduce the high access cost for small shared directories (but it does not incur in any additional overhead.) Even then, results for 16 nodes show a better behavior than pure GPFS (less variability and faster convergence to reduced operation times.) That makes us expect that the benefits of COFS will be more noticeable for a larger number of nodes (as we will show in section 6.4.)

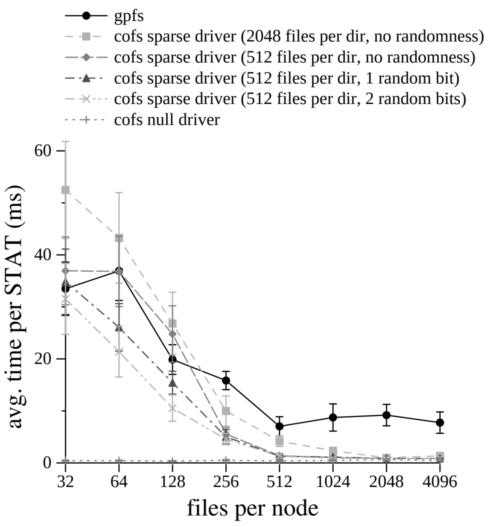


Figure 6: Stat times for several COFS configurations (8 nodes)

In order to thoroughly understand the causes of COFS benefits, we conducted some additional experiments. Fig. 6 shows the results for 8 nodes: we compared pure GPFS behavior with the results of COFS using different sets of parameters for the *sparse* data driver (increasing the number of entries per directory in the underlying file system from 512 to 2048, and modifying the layout randomness factor); we also executed the tests using the COFS *null* data driver (which diverts all operations to `/dev/null`) in order to distinguish pure COFS behavior from that related to the underlying file system.

The cost of accessing few files is mostly due to GPFS (as the overhead does not appear with the *null* data driver.) And this cost is partially related to synchronization: increasing the “random bits” in the *sparse* layout effectively distributes the files into a wider number of underlying directories, and we can observe that this helps to mitigate the problem (by reducing the chances of simultaneous colliding accesses from different nodes - as a matter of fact, we are reducing the level of false sharing inside the back-end directories.)

The effect of layout randomization is complemented by the limitation of the directory size. When the maximum number of entries is increased up to 2048 entries, the synchronization overhead is also increased, resulting

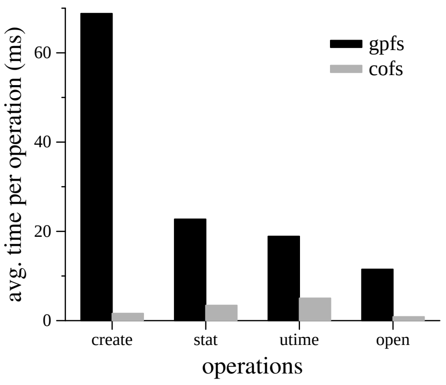


Figure 7: Operation times for 256 operations per node on 64 nodes

in larger operation times (to better understand this effect, it is worth noting that the benchmark creates all the files to be "stated" from a single node in such a way that files to be accessed by node 1 are created first, then those for node 2, and so on; beyond the directory size limit, the COFS data driver manages to put each node's files into different directories, effectively converting shared accesses into non-shared accesses). This way, access time approaches the *null* driver results. On the contrary, pure GPFS is unable to reduce the operation time.

In summary, our measurements show that using the virtualization of the name space provided by the COFS framework can drastically boost our underlying GPFS file system for file creations on shared parallel environments (with speed-up factors from 2 to 10, as shown in Fig. 5(a)). For the rest of metadata operations, performance is also boosted (though the speed-ups are more moderated), and the overhead and variability for parallel access to small-sized directories is reduced.

6.4 Scalability

One of our concerns was to check that our proof-of-concept prototype was able to deal with much larger environments than our testbed. So, we conducted some additional experiments to probe possible scalability limits in two different aspects: number of files in the file system and number of nodes in the distributed environment.

In order to check how many entries was able to handle a single COFS metadata server, we artificially populated the metadata database with a large directory tree (with a proportion of 10% directories and 90% files), and then re-run the experiments on it. Results show no significant degradation on create operations up to 8,000,000 entries (which is one order of magnitude larger than the actual file system in our testbed.) With respect to other metadata operations (*stat*, *utime*, *open*), results were stable up to 10,000,000 entries (where we started hitting physical memory limits - the metadata server uses a JS20 blade with 4GB RAM). Further scalability should not be a problem, as mechanisms to deal with creations in larger tables in Mnesia are discussed in Erlang-related literature [1], and node physical limits can be bypassed by using Mnesia distribution mechanisms [18]. Also, several works on file system metadata focus on partitioning techniques to distribute metadata and name space information [11, 19, 27] to multiple nodes.

Regarding the number of nodes in the cluster, we had the opportunity to deploy our prototype onto a dedicated cluster with an architecture very similar to our original testbed (for both blade centers and file system servers.) By running the *metarates* benchmark on this testbed we were able to confirm that the benefits of virtualization are not only maintained but increased in larger scales. Fig. 7 shows the comparison of metadata operation times on 64 nodes, accessing 256 files per node on a shared directory (results with other directory sizes are omitted due to lack of space, but they show similar trends).

As we may observe, the base-line given by pure GPFS shows considerably high operation times for 64 nodes, because the inter-node conflicts when accessing a shared directory increase with the number of nodes and they are the main component of the operation cost. On the contrary, COFS is able to mitigate such conflicts, allowing GPFS to obtain remarkable speed-ups. We expect to observe the same trend for a larger number of nodes.

7 Conclusions and future work

We propose the use of virtualization techniques to both take advantage of the different file system optimizations, while eliminating the burden from the high level applications. Name space virtualization is a key element, as it allows decoupling the high level file organization from the underlying file system configuration details.

The proof-of-concept prototype we implemented (COFS) shows that this is a feasible approach, as the theoretical cost of the virtualization layer is largely compensated by its benefits. Specifically, we have used the prototype to boost a single GPFS file system, significantly improving the metadata performance for shared parallel workloads.

In the near future we will focus on tuning the prototype (using autonomic techniques when possible), extending it to support the additional functionalities described in section 3.3 and exploring the different ways to increase scalability.

References

[1] Erlang/OTP. Web site: http://www.erlang.org.

[2] FUSE: Filesystem in userspace. Web site: http://www.fuse.org.

[3] Top 500 supercomputing sites. Web site: http://www.top500.org.

[4] Parallel Virtual File system, version 2. Online document, http://www.pvfs.org/cvs/pvfs-2-7-branch-docs/doc//pvfs2-guide.pdf, September 2003.

[5] Metarates. Web site: http://www.cisl.ucar.edu/css/software/metarates/, 2004.

[6] The Hadoop distributed file system: Architecture and design. Online document, http://hadoop.apache.org/core/docs/current/hdfs_design.html, 2007.

[7] Lustre file system. high-performance storage architecture and scalable cluster file system. White paper, http://www.sun.com/software/products/lustre/docs/lustrefilesystem_wp.pdf, December 2007.

[8] CARNS, P. H., WALTER B. LIGON, I., ROSS, R. B., AND THAKUR, R. PVFS: a parallel file system for linux clusters. In *ALS'00: Proceedings of the 4th conference on 4th Annual Linux Showcase & Conference* (Berkeley, CA, USA, 2000), USENIX Association, pp. 28–28.

[9] COPE, J., OBERG, M., TUFO, H. M., AND WOITASZEK, M. Shared parallel filesystems in heterogeneous linux multi-cluster environments. In *Proceedings of the 6th LCI International Conference on Linux Clusters: The HPC Revolution* (2005).

[10] DEVULAPALLI, A., AND WYCKOFF, P. File creation strategies in a distributed metadata file system. In *IPDPS'07: Proceedings of the 21th International Parallel and Distributed Processing Symposium* (March 2007), IEEE, pp. 1–10.

[11] DOUCEUR, J. R., AND HOWELL, J. Distributed directory service in the Farsite file system. In *OSDI'06: Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 321–334.

[12] EISLER, M., CORBETT, P., KAZAR, M., NYDICK, D. S., AND WAGNER, J. C. Data ONTAP GX: A scalable storage cluster. In *FAST'07: Proceedings of the 5th USENIX Conference on File and Storage Technologies* (2007), USENIX Association.

[13] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *SOSP 2003: Proceedings of the 19th ACM Symposium on Operating Systems Principles* (October 2003).

[14] III, C. B. M., AND GRUNWALD, D. Peabody: The time travelling disk. In *MSST'03: Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies* (Washington, DC, USA, 2003), IEEE Computer Society, p. 241.

[15] JOUKOV, N., KRISHNAKUMAR, A. M., PATTI, C., RAI, A., SATNUR, S., TRAEGER, A., AND ZADOK, E. RAIF: Redundant Array of Independent Filesystems. In *MSST'07: Proceedings of 24th IEEE Conference on Mass Storage Systems and Technologies* (San Diego, CA, September 2007), IEEE, pp. 199–212.

[16] KUNKEL, J. M., AND LUDWIG, T. Performance evaluation of the PVFS2 architecture. In *PDP'07: Proceedings of the 15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing* (2007), IEEE.

[17] NARAYANAN, D., DONNELLY, A., AND ROWSTRON, A. Write off-loading: practical power management for enterprise storage. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies* (2008), USENIX Association, pp. 253–267.

[18] NELSON, J. Concurrent caching. In *Proceedings of Erlang'06* (September 2006), ACM.

[19] PATIL, S. V., GIBSON, G. A., LANG, S., AND POLTE, M. Giga+: scalable directories for shared file systems. In *PDSW '07: Proceedings of the 2nd international workshop on Petascale data storage* (New York, NY, USA, 2007), ACM, pp. 26–29.

[20] PENDRY, J.-S., AND MCKUSICK, M. K. Union mounts in 4.4BSD-lite. In *TCON'95: Proceedings of the USENIX 1995 Technical Conference* (Berkeley, CA, USA, 1995), USENIX Association, pp. 3–3.

[21] PIKE, R., PRESOTTO, D., THOMPSON, K., TRICKEY, H., AND WINTERBOTTOM, P. The use of name spaces in Plan 9. *SIGOPS Oper. Syst. Rev. 27*, 2 (1993), 72–76.

[22] SCHMUCK, F., AND HASKIN, R. GPFS: A shared-disk file system for large computing clusters. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, January 2002), IBM, USENIX Association, pp. 231–244.

[23] SEBEPOU, Z., MAGOUTIS, K., MARAZAKIS, M., AND BILAS, A. A comparative experimental study of parallel file systems for large-scale data processing. In *LASCO'08: Proceedings of First USENIX Workshop on Large-Scale Computing* (June 2008).

[24] STEIN, L. Stupid file systems are better. In *HOTOS'05: Proceedings of the 10th Conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2005), USENIX Association, pp. 5–5.

[25] WARFIELD, A., ROSS, R., FRASER, K., LIMPACH, C., AND HAND, S. Parallax: managing storage for a million machines. In *HOTOS'05: Proceedings of the 10th Conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2005), USENIX Association, pp. 4–4.

[26] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: a scalable, high-performance distributed file system. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 307–320.

[27] WEIL, S. A., POLLACK, K. T., BRANDT, S. A., AND MILLER, E. L. Dynamic metadata management for petabyte-scale file systems. In *SC'04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing* (Washington, DC, USA, 2004), IEEE Computer Society, p. 4.

[28] WELCH, B., UNANGST, M., ABBASI, Z., GIBSON, G., MUELLER, B., SMALL, J., ZELENKA, J., AND ZHOU, B. Scalable performance of the panasas parallel file system. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies* (2008), USENIX Association, pp. 17–33.

[29] WRIGHT, C. P., DAVE, J., GUPTA, P., KRISHNAN, H., QUIGLEY, D. P., ZADOK, E., AND ZUBAIR, M. N. Versatility and Unix semantics in namespace unification. *Trans. Storage 2*, 1 (2006), 74–105.

[30] YU, W., VETTER, J., CANON, R. S., AND JIANG, S. Exploiting lustre file joining for effective collective io. In *CCGrid'07: Proceedings of the International Conference on Cluster Computing and Grid* (2007), IEEE Computer Society.

[31] ZADOK, E., IYER, R., JOUKOV, N., SIVATHANU, G., AND WRIGHT, C. P. On incremental file system development. *ACM Transactions on Storage (TOS) 2*, 2 (May 2006), 161–196.