Using Filesystem Virtualization to Avoid Metadata Bottlenecks

Ernest Artiaga
Barcelona Supercomputing Center (BSC-CNS)
Jordi Girona 31, Barcelona, E-08034 Spain
Email: ernest.artiaga@bsc.es

Toni Cortes
Barcelona Supercomputing Center (BSC-CNS), and
Dept. of Computer Architecture
Technical University of Catalonia (UPC)
Jordi Girona 1-3, Barcelona, E-08034 Spain
Email: toni@ac.upc.edu

***Abstract*—Parallel file systems are very sensitive to adverse conditions, and the lack of synergy between such file systems and some of the applications running on them has a negative impact on the overall system performance. Our observations indicate that the increased pressure on metadata management is one of the relevant causes of performance drops. This paper proposes a virtualization layer above the native file system that, transparently to the user, reorganizes the underlying directory tree, mitigating bottlenecks by taking advantage of the native file system optimizations and limiting the effects of potentially harmful application behavior. We developed COFS (COmposite File System) as a proof-of-concept virtual layer to evaluate the feasibility of the proposal.**

I. Introduction

Nowadays, computing clusters are available everywhere from small data centers with a few nodes to large high-performance facilities with thousands of nodes. Consequently, the heterogeneity of the applications running on these platforms has also increased, ranging from parallel applications that coordinately use the computing resources spanning across the available nodes, to workloads composed of loosely coupled processes that simply use local resources to produce results which are later to be gathered and analyzed.

Parallel file systems have emerged to provide a storage solution for this kind of environments, providing mechanisms for distributing data across a range of storage servers and making it readily available to the computing elements, so that the applications may access the data essentially in the same way they did when file systems were only locally accessible on each individual machine.

Unfortunately, the mechanisms to coordinate the parallel view and provide consistency across several nodes are complex and expensive. Parallel file systems try to keep pace with the increasing demands for performance by incorporating optimizations to improve specific workloads, but usually at the cost of degraded performance when an unexpected access pattern appears. Additionally, file system overheads tend to affect the whole system (not only the "infringing" applications), as file servers are kept overloaded and all requests are delayed.

The observations made in our own production clusters suggested that an excessive pressure on the file system metadata mechanisms was behind some important performance drops, and that the pressure was caused by some applications using inadequate file and directory layouts which were pushing the file system out of the "optimized grounds."

Our proposal consists of using a virtual layer above the native file system to decouple the name space and metadata management from the underlying directory tree, so that applications can use their desired layout, which is transparently converted into a file organization that avoids synchronization conflicts and improves the overall system performance. We also designed and implemented a proof-of-concept framework to evaluate the feasibility of the proposal and its ability to boost the performance of an existing file system.

II. Case Study

This work was motivated by the performance drops observed in our production clusters. Such clusters use a GPFS [1] file system and have a very heterogeneous workload corresponding to different projects, comprising both large parallel applications spanning across many nodes, and large amounts of relatively small jobs.

Large parallel applications usually create per-node auxiliary files and/or generate checkpoints by having each node dump 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 also in a shared directory; the overall access pattern is then similar to that from parallel applications: lots of files being created in parallel in a single shared directory.

The problem is that this convenient file layout (from the applications perspective) is in conflict with the internal file system structure. In classical local file systems, a directory was used as a hint to indicate "affinity"; trying to exploit it, file systems tended to pack together the management information about files in the same directory (access permissions, statistics and also the physical location of the file's data). As most of the modern parallel file systems evolved from the classical local ones, this approach is still present. Thus, parallel file systems may end up trying to keep consistent a relatively small pack of miscellaneous management information corresponding to files that, despite being in the same directory, are probably unrelated and used in an independent way.

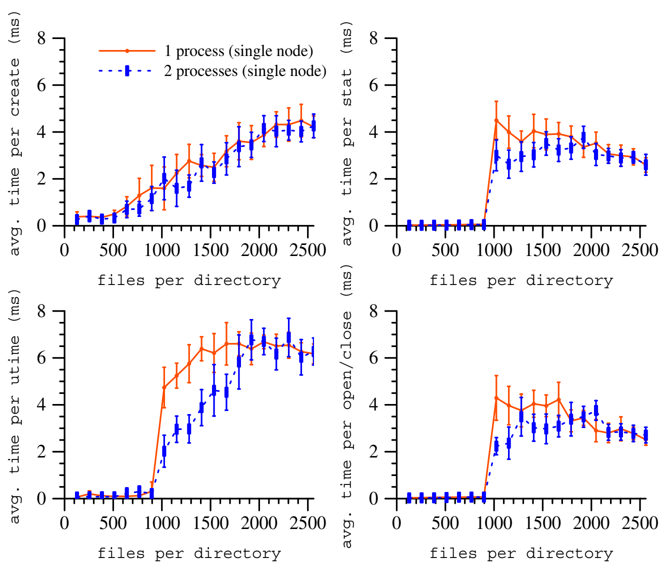

Fig. 1. Effect of the number of entries in a directory in GPFS

Preliminary observations in our clusters indicated that, indeed, some of the global performance drops seemed to be related to periods when an application was involved in heavy metadata activity (e.g. parallel file creation or large directory traversals). This is consistent with the fact that large directories, specially when populated in parallel, require GPFS to use a complex synchronization mechanism to guarantee the consistency [1], resulting in a far-from-optimal performance.

In this situation, a virtual layer able to transparently decouple the application view from the underlying file layout should reduce the pressure on GPFS metadata system and minimize the synchronization costs (leading to an overall performance improvement). This also avoids the need for having to change how the individual applications work (which is difficult or even impossible for some legacy codes).

To verify this assumption and evaluate its impact, we conducted a series of tests under a controlled environment in a small cluster. The rest of this section describes the testbed and shows the metadata behavior on a bare GPFS file system.

A. Testbed

Our environment consisted of a cluster of IBM JS20 *blades*, with 2 processors (PPC970FX) and 4GB of RAM each, running SUSE linux with kernel version 2.6.16-45. Unless stated otherwise, our test *blades* where grouped in a *blade center* with an internal 1GB switch. The file system was GPFS v3.1.0-15, based on two external Intel-based servers connected to the blade center by 1 GB link each.

The situation we wanted to evaluate essentially involved parallel metadata operations, so we used *metarates* as the main 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 metadata concurrently.

The operations exercised are *create*, *stat* and *utime*; additionally, we also included code for *open/close* sequences. The four measurements are taken consecutively: first all files are created in parallel, and then deleted; for each of the other operations, the first node sequentially creates all files, which are then accessed (*stat'd*, *utime'd* or *open/close'd*) in parallel, and then deleted again by the first node.

B. 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. The, our first task consisted of measuring the performance of metadata operations on the bare GPFS file system in our testbed.

Fig. 1 shows average times for GPFS metadata operations in a single node (using 1 and 2 processes in the same node). It reveals an extremely good behavior for *stat*, *utime* and *open/close* when the directory size is below 1024 entries (with a performance comparable to local file system rates).

The low operation times are probably caused by the ability of GPFS to delegate control to clients under certain circumstances (e.g. single-node access and data present on local cache). Beyond 1024 entries, performance drops to network-compatible rates. Having two processes seems to slightly compensate, as some fetched data can be re-used by the second process (though this effect disappears beyond 2048 entries).

The pattern is different for *create* operations: there is no local cache to exploit (as we are creating new entries) and operation time follows a steady increase above 512 entries.

Fig. 2 shows some of the results obtained running the *metarates* benchmark on 4 and 8 nodes (using a single process per node). The first observation is the large increase of the *create* time when operating in parallel. For a directory with 1024 entries, it goes from slightly less than 2 ms in a single node to more than 20 ms in 4 nodes and more than 30 ms for 8 nodes. As such operations have a very small payload (only i-node like information) we may assume that most part of the large operation time is consumed by communications and consistency-related traffic (even when each participating process works on a different set of files). It is also worth mentioning that the number of files in the directory seems to have little impact for *create*, compared to the number of nodes.

The effect on the rest of operations is more moderated for 4 nodes, but still noticeable: for 1024 files, it increases from 4 ms (*stat* and *open*) and 6 ms (*utime*) to about 10 ms. Moving

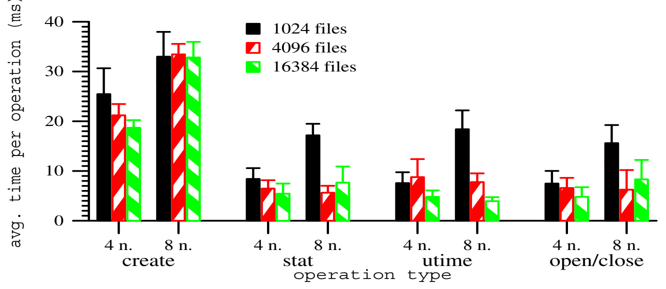

Fig. 2. Parallel metadata behavior of GPFS

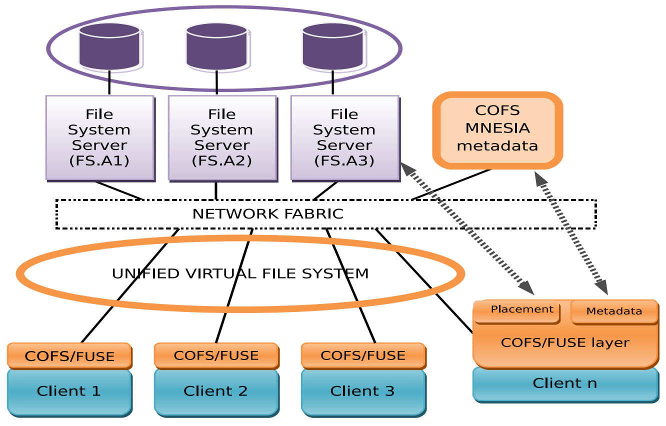

Fig. 3. File system architecture with COFS-based virtualization layer

to 8 nodes has a notable impact for 1024 files (time goes up to 15-20 ms), but not for larger directories. This effect is related to distributed locking granularity: for example, a *stat* response from the servers will contain data about several entries in the same directory to take advantage of bandwidth and hide network latencies; but this also introduces a risk of "false-sharing" between nodes, and the less files we have, the higher the probability that two nodes want to access information that has been packed together (producing a locking conflict).

C. Lessons Learned

We learned some facts from the experiments:

- the best sequential behavior is obtained for small directories (below 1024 entries for *stat*, *utime* and *open/close* and about 512 entries for *create*); the optimization disappears for large directories and/or parallel accesses;
- parallel creations on shared directories have an important overhead;
- parallel non-*create* operations on shared directories are also likely to produce locking conflicts between nodes.

Decoupling the name space from the actual file system layout should mitigate the issues. A virtual layer could offer large shared directory views while internally splitting them to reduce synchronization on the GPFS servers, and keep the use patterns within the optimized boundaries. The COmposite File System (COFS) was developed to validate this approach.

III. COFS Design and Implementation

The goal of the current COFS prototype is to be able to decouple the user view of the file hierarchy, the metadata handling and the physical placement of files. This allows us to offer a virtual view of the directory tree, while the actual layout can be optimized for the underlying file system.

COFS is implemented as a user-level FUSE (Filesystem in USErspace [2]) daemon, and it is independent from the underlying file system. The reasons for this are two-fold. First, we believe that similar issues affect several file systems; so, the solving mechanism must be as generic as possible. Second, our future plans include deploying our framework in production-grade clusters, and having a userland drop-in package without hard requirements on kernel modifications or configurations makes it much easier to have access to such environments.

Despite being a proof-of-concept prototype, having a reasonable performance is a must, as we pretend to show that the mechanism does not have a significant overhead and, on the contrary, can boost performance in several situations. Additionally, we have been careful to not assume simplifications that could overlook complex aspects of actual file systems (in particular, the COFS prototype is POSIX compliant, and we have leveraged the underlying technologies to provide an adequate level of security for file system operations).

A. Architecture

Fig. 3 shows the COFS prototype integrated in a file system environment with 3 file servers contacted by several clients. COFS introduces an additional node to handle metadata information and an extra layer on each client providing a virtual view of the file system layout.

The extra layer on each node is based on FUSE [2], and provides a file system interface which can be mounted as any other file system. All file system requests are then diverted via VFS-like callbacks to two userspace modules: the *placement* and *metadata* drivers (see client *n* in Fig. 3). When needed, requests are eventually forwarded to the actual file system.

B. The Placement Driver

The *placement driver* maps the regular files in the user view into the underlying file system layout. The current version was specifically designed to mitigate the issues observed in our GPFS clusters (see section II) and to prevent situations that could harm the overall system performance. Nevertheless, different mapping polices could be easily implemented.

The currently implemented policy computes the underlying path name at creation time from a hash function applied to a combination of the following parameters: the node issuing the creation request, the parent directory in the virtual view of the file hierarchy, and the process creating the file. The result is that files tend to be organized in different directories for different nodes (in order to avoid inter-node conflicts) while still being loosely coupled according the user view.

To account for cases where files are created from the same node and then accessed in parallel, a randomization factor is used, resulting in files being further distributed in a subdirectory level below the path determined by the hash function (reducing the chances of simultaneous parallel access to the same underlying directory).

Finally, we applied a limit of 512 entries to the underlying directory size. As our goal is to convert most parallel accesses into conflict-free local accesses, we could benefit from the highly optimized GPFS behavior for local small directories, masking the possible overheads caused by the virtual layer.

C. The Metadata Driver and Service

The *metadata driver* takes care of requests related to hard and symbolic links, directories and regular file attributes (essentially type, owner, group and access permissions) and forwards them to a centralized *metadata service*.

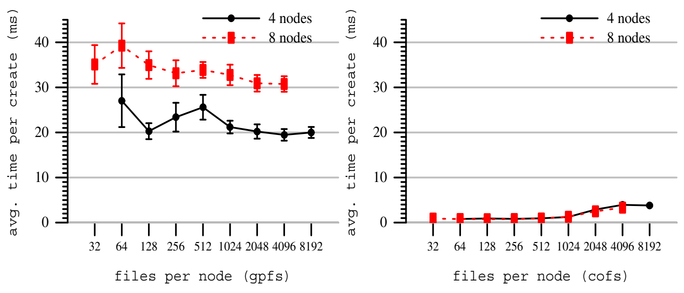

Fig. 4. Create time (pure GPFS vs. COFS over GPFS)

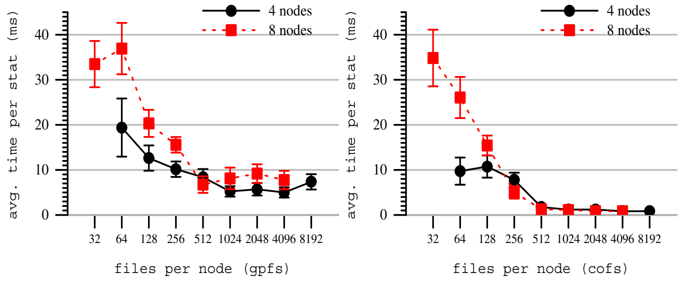

Fig. 5. Stat time (pure GPFS vs. COFS over GPFS)

The *metadata service* is the responsible for maintaining the virtual view of the file system hierarchy (as seen by the applications) and also keeping the metadata information.

The reason for not delegating metadata handling to the underlying file system is to further reduce the possibility of unnecessary access conflicts. For example, a typical i-node structure contains the file's type, but also pointers to its physical data blocks. If a file is being written from a node while its parent directory is being listed from a different one, there will be a synchronization issue (even when different fields from the i-node are being used).

The COFS *metadata service* mitigates these conflicts by decoupling the metadata information from its implementation and packing in the underlying file system. Instead, metadata is maintained as a small set of database tables having the information about files and directories, and pure metadata operations are translated to the appropriate database queries. Only requests related to file contents are forwarded to the underlying file system.

The implementation of the *metadata service* is based on the Mnesia database from the Erlang/OTP environment [3]. Mnesia is optimized for simple queries in soft real time distributed environments (with support for transactions and fault tolerance mechanisms). The Erlang language internally deals with thread synchronization and provides support for transparent distributed computing.

D. Data Handling

COFS does not deal with physical data storage, and it does not keep any information about block locations for files or directories: this is delegated to the underlying file system once a virtual file name is mapped into the low-level structure.

Accordingly, the *read* and *write* operations intercepted by FUSE are directly forwarded to the actual underlying file, without any intervention beyond the necessary buffer copies.

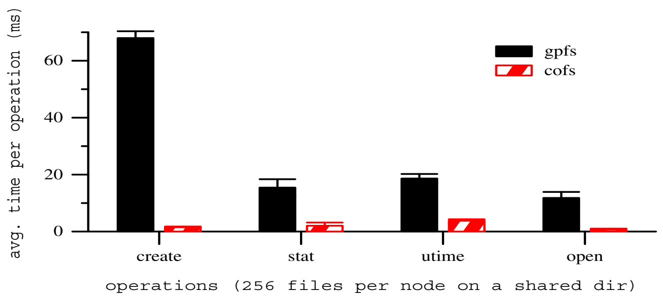

Fig. 6. Operation times on 64 nodes

IV. Evaluation and Performance Results

The measurements in this section have been taken in the same testbed described in section II. When using COFS, one of the *blades* (with 2 PPC970FX processors and 4GB of RAM) was used to host the COFS *metadata service*; the backend for the Mnesia database was a 25 GB disk locally attached to that node and formatted with the *ext3* file system.

Metarates results have been coalesced (merging results for 1 and 2 processes per node) as we have observed that trends are determined by nodes as a whole, and not individual processes (differences are marginal compared to overall values). Also, figures are related to the number of files accessed per node, instead of the total number of files used in the benchmark. All files are created in a single shared directory.

Additionally, we also used the *IOR* (Interleaved Or Random) benchmark to measure data I/O performance for GPFS with and without the COFS virtual layer. Even if COFS does not deal with data I/O, we wanted to verify that the hierarchy re-organization had no negative impact in this aspect. *IOR v2* was developed at LLNL and provides aggregate I/O data rates for both parallel and sequential *read/write* operations to shared and separate files in a parallel file system. The benchmark was executed using the POSIX interface with aggregate data sizes of 256MB, 1GB and 4GB. (The individual file size when using separate files is the aggregated data size divided into the number of participating processes.)

A. Virtualization Results

Fig. 4 shows the benefits of breaking the relationship between the virtual name space offered by COFS (exporting a single shared directory to the application level) and the actual layout of file 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 improvement translates into a reduction of the average create time from more than 20 ms for GPFS to between 2 and 5 ms when using COFS over GPFS on 4 nodes. The scaling overhead when moving from 4 to 8 nodes in pure GPFS (operation time increases from 20 ms to 30 ms) is also eliminated with the use of COFS.

TABLE I
IMPACT OF COFS ON DATA TRANSFERS, DEPENDING ON USE PATTERN

	Separate files per process	**Single shared file**
Sequential read	COFS performance comparable to GPFS except for small files (<32MB per node), where COFS suffers an important slowdown	COFS performance comparable to GPFS
Random read	COFS performance comparable to GPFS except for small files (<32MB per node), where COFS suffers an important slowdown	COFS performance comparable to GPFS
Sequential write	COFS performance drawback for single node, and improved performance of COFS over GPFS as the number of nodes increases	COFS performance drawback for single node, and comparable performance for multi-node
Random write	COFS performance comparable to GPFS except for small files (<32MB per node), where COFS suffers a slight slowdown	COFS performance comparable to GPFS

Fig. 5 shows the average time for *stat*. Overall, we can see that 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.

COFS reduces the *stat* time when a directory grows beyond 512 entries per node (from approx. 7 ms to 1 ms for 8 nodes; and from 5 ms to 1 ms for 4 nodes). Even for smaller directories, where conflicts are more difficult to avoid, the performance is comparable to pure GPFS and even slightly better (showing that the benefits of the virtualization layer compensate for the introduced overhead).

Figures for *open/close* and *utime* are not shown as they follow a pattern closely resembling the *stat* behavior. Times for *utime* in pure GPFS stabilize about 6-7 ms, compared to 4 ms when using COFS; values obtained for *open/close* are very similar to *stat* results, for both pure GPFS and COFS.

Additionally, we conducted an experiment to see how the system behaved in a larger cluster. For this, we added additional *blade centers* to our initial cluster up to 64 nodes. Unfortunately, the available connectivity was limited, and the network topology had to be hierarchical, meaning that some *blades* needed to cross several switches to reach the original blade center, which was directly connected to the file servers.

Despite not being directly comparable to the original testbed (due to bandwidth limitations), the results confirm that the benefits of virtualization are not only maintained but increased in larger scales. As an example, Fig. 6 shows the comparison of metadata average operation times on 64 nodes, accessing 256 files per node in a shared directory (results with other directory sizes show similar trends). Pure GPFS shows considerably higher operation times due to inter-node conflicts when accessing a shared directory, while COFS seems to be able to avoid such conflicts. We expect to observe the same trend for even larger number of nodes.

In summary, our measurements show that the virtualization of the name space provided by the COFS framework can drastically boost our base GPFS file system for file creations on shared parallel environments (with speed-up factors from 5 to 10, as shown in Fig. 4, and even more when the number of participating nodes is increased). 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.

B. Impact on Data Transfer Bandwidth

After verifying that the benefits obtained by our prototype regarding metadata handling are promising, and that it effectively mitigates the issues motivating the present work, we also wanted to measure the possible impact of the virtualization environment on read/write operations on file contents.

Altering the file hierarchy could lead the underlying file system to modify the actual location of data, impacting negatively on read/write bandwidth; additionally, we wanted to be sure that COFS infrastructure was not adding an unacceptable overhead to data transfer operations. Possible causes would be FUSE's double buffer copying, round-trips to the metadata service or caching issues.

Table I summarizes the results obtained with the *IOR* I/O benchmark. Overall, COFS over GPFS is usually able to obtain a data transfer performance similar to native GPFS. The only noticeable exceptions occur when each node access independent small files.

For operations on small separate files (less than 32MB), pure GPFS is able to exploit its optimizations and the cache by locally keeping both the metadata and the file contents for read operations (files were created and written in the same node they were accessed). Additionally, the total benchmark times for such small files are about a few milliseconds, which is comparable, for example, to the extra round-trips needed by COFS to access its metadata server. In this circumstances, COFS is paying the cost of its infrastructure. The case of writes is slightly different: not being a pure local cache operation (as data has to be eventually sent to file servers) GPFS cannot apply all of its optimizations; consequently, COFS benefits have room to partially mask the infrastructure costs, resulting in only slightly lower performance. The performance penalties disappear with larger file sizes, as transfer times become dominant compared with COFS infrastructure costs.

Noticeably, we also observed a positive effect of COFS when writing sequentially to separate files. In this case, GPFS bandwidth suffers a degradation as the number of participating nodes was increased, while COFS was able to neutralize this effect. A closer look revealed that, for a larger number of nodes, the increased cost of the parallel open operation was "serializing" the data transfers (as the last node was able to open the file only much later than the first one, it also started to transfer data later); as a result, the use of the available data bandwidth was reduced. On the contrary, COFS reduced the open time to a minimum, allowing all nodes to start transferring data in parallel and achieving a much better use of the network bandwidth.

In summary, we did not observe a remarkable global impact of the COFS virtualization layer on data transfer rates. The punctual performance drops affect only the GPFS highly optimized cases (local accesses to independent small files) where there is little room for improvement. Even then, the nature of the cases would make it possible to reduce the differences by adding the same aggressive caching and delegation techniques for strictly local accesses to the COFS framework.

V. Related Work

The notion of virtualization is not new to file systems, and it is commonly used at different levels to hide complexities and extend functionalities. At operating system level, mechanisms such as the Linux "Virtual File System" (VFS) provide a common layer aimed to facilitate the integration of different file systems into the OS internal structure [4]. Instead of altering such low level structures, COFS uses FUSE to modify the file system requests, so that the current internal structures used by regular file systems can be used with greater efficacy.

Virtualization is also used at the name space level. Commercial systems like ONTAP GX [5] provide a virtual layer allowing volumes (directory subtrees) to be transparently relocated or replicated to improve performance. COFS works at a much finer level (file-based, instead of directory based) so that user file organization does have an impact on performance.

RAIF [6] also uses virtualization techniques to divert individual files to different target file systems; nevertheless, the directory tree itself is not virtualized and must be replicated in all target underlying file systems.

Some object-based systems like Ursa Minor [7] avoid the the conflicts related to inadequate directory trees by eliminating the hierarchical name space (data objects have an identifier in a flat name space and are accessed by a non-standard protocol). Other systems like PVFS2 [8] also provide an alternate ad-hoc library with non-standard semantics to let the user bypass the parallelization conflicts. The approach used in COFS is different, as the transformation of the name space is done transparently, providing the user with standard semantics and a classical directory layout.

Regarding the metadata service, several file systems such as Lustre [9], Ceph [10] or PVFS2 [8] decouple metadata and name space management from data accesses, having separate services to handle them. The pNFS [11] decoupled architecture also follows this principle. In all these cases, nevertheless, the metadata services also include information to reach the physical location of file contents.

The COFS metadata service, on the contrary, does not keep any information about how and where file data is physically stored: data distribution and handling is responsibility of the underlying file system. This allows us tho have a lighter server with a reduced load (e.g. there is no need to contact the COFS metadata server if a file is written or resized).

VI. Conclusion

Our experimental results show that breaking the ties between the user view and the underlying file system organization helps to minimize the impact of synchronization conflicts on parallel metadata operations.

Virtualization techniques have been very valuable as a tool to decouple the name space and metadata from the low-level data handling. The proof-of-concept prototype we implemented (COFS) shows that virtualization is a feasible approach, as its theoretical overhead is largely compensated by its benefits. As a proof, we have successfully used the prototype to boost a GPFS file system by improving its metadata performance for shared parallel workloads.

Unlike usual file system optimizations, which introduce specialized features to improve the response in very specific situations, COFS allows for a complementary approach: to automatically re-organize requests to avoid the situations which are harmfull for the overall performance of the file system.

Acknowledgment

This work was partially supported by the EU IST program as part of the XtreemOS project (contract FP6-033576), by the Spanish Ministry of Science and Technology under the TIN2007-60625 grant, and by the Catalan Government under the 2009-SGR-980 grant.

References

[1] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *FAST'02: Proc. of the 1st USENIX Conf. on File and Storage Technologies*, Jan 2002.

[2] "FUSE: Filesystem in userspace," http://www.fuse.org.

[3] "Erlang/OTP," http://www.erlang.org.

[4] E. Zadok, R. Iyer, N. Joukov, G. Sivathanu, and C. P. Wright, "On incremental file system development," *Trans. Storage*, vol. 2, no. 2, pp. 161–196, May 2006.

[5] M. Eisler, P. Corbett, M. Kazar, D. S. Nydick, and J. C. Wagner, "Data ONTAP GX: A scalable storage cluster," in *FAST'07: Proc. of the 5th USENIX Conf. on File and Storage Technologies*, 2007.

[6] N. Joukov, A. M. Krishnakumar, C. Patti, A. Rai, S. Satnur, A. Traeger, and E. Zadok, "RAIF: Redundant array of independent filesystems," in *MSST'07: Proc. of 24th IEEE Conf. on Mass Storage Systems and Technologies*, Sep 2007.

[7] M. Abd-El-Malek, I. W. V. Courtright, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie, "Ursa Minor: versatile cluster-based storage," in *FAST'05: Proc. of the 4th USENIX Conf. on File and Storage Tech.*, 2005.

[8] "Parallel Virtual File System, version 2," http://www.pvfs.org/cvs/pvfs-2-7-branch-docs/doc/pvfs2-guide.pdf, PVFS2 Development Team, 2003. [Online]. Available: http://www.pvfs.org/cvs/pvfs-2-7-branch-docs/doc/pvfs2-guide.pdf

[9] "Lustre file system: High-performance storage architecture and scalable cluster file system," White paper, Sun Microsystems, Sun Microsystems, Dec 2007. [Online]. Available: http://www.sun.com/software/products/lustre/docs/lustrefilesystem_wp.pdf

[10] 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'06: Proc. of the 7th Symp. on Op. Systems Design and Impl.*, 2006.

[11] D. Hildebrand and P. Honeyman, "Exporting storage systems in a scalable manner with pNFS," in *MSST'05: Proc. of the 22nd IEEE / 13th NASA Goddard Conf. on Mass Storage Systems and Technologies*, 2005.