SEVENTH FRAMEWORK PROGRAMME
Research Infrastructures

INFRA-2011-2.3.5 – Second Implementation Phase of the European High Performance Computing (HPC) service PRACE

PRACE-2IP

PRACE Second Implementation Project

Grant Agreement Number: RI-283493

D12.4
Performance Optimized Lustre
Draft/Final

Version: 1.0
Author(s): Ernest Artiaga, Alberto Miranda, Barcelona Supercomputing Center
Date: 25.08.2012

Project and Deliverable Information Sheet

PRACE Project	Project Ref. №: RI-283493	
	Project Title: PRACE Second Implementation Project	
	Project Web Site: http://www.prace-project.eu	
	Deliverable ID: D12.4	
	Deliverable Nature: Report	
	Deliverable Level: PU	Contractual Date of Delivery: 31/08/2012
		Actual Date of Delivery: 31/08/2012
	EC Project Officer: Leonardo Flores Añover	

* - The dissemination level are indicated as follows: **PU** – Public, **PP** – Restricted to other participants (including the Commission Services), **RE** – Restricted to a group specified by the consortium (including the Commission Services). **CO** – Confidential, only for members of the consortium (including the Commission Services).

Document Control Sheet

Document	Title: Performance Optimized Lustre	
	ID: D12.4	
	Version: <1.0>	Status: ***Draft/Final***
	Available at: http://www.prace-project.eu	
	Software Tool: Microsoft Word 2007	
	File(s): D12.4.docx	
Authorship	Written by:	Ernest Artiaga, Alberto Miranda, Barcelona Supercomputing Center
	Contributors:	Jonathan Martí, BSC Toni Cortes, BSC Jan Christian Meyer, SIGMA
	Reviewed by:	D. Erwin, FZJ; Jerry Erikson, UMU
	Approved by:	MB/TB

Document Status Sheet

Version	Date	Status	Comments
0.1	20/July/2012	Draft	
0.2	24/July/2012	Draft	
1.0	25/August/2012	Final version	

Document Keywords

Keywords:	PRACE, HPC, Research Infrastructure, Lustre, GPFS, Metadata, data, COFS

Table of Contents

List of Figures

List of Tables

References and Applicable Documents

[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*, 2002.

[2] A. Devulapalli and P. Wyckoff, “File Creation Strategies in a Distributed Metadata File System,” in *IPDPS '07: Proceedings of 21st IEEE International Parallel and Distributed Processing Symposium*, Long Beach, CA, USA, 2007.

[3] W. Yu, J. Vetter, R. S. Canon and S. Jiang, “Exploiting Lustre File Joining for Effective Collective IO,” in *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, 2007.

[4] P. J. Braam, “Lustre File System: High-Performance Storage Architecture and Scalable Cluster File System (White Paper),” Sun Microsystems, Inc., 2007.

[5] “Metarates,” University Corporation for Atmospheric Research (UCAR) and NCAR Scientific Computing Division, 2004. [Online]. Available: http://www.cisl.ucar.edu/css/software/metarates/.

[6] “FUSE: Filesystem in Userspace,” [Online]. Available: http://www.fuse.org.

[7] “Erlang/OTP (Open Telecom Platform),” [Online]. Available: http://www.erlang.org.

[8] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin and R. Panigrahy, “Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web,” in *Proceedings of the 29th ACM Symposium on Theory of Computing (STOC)*, El Paso, Texas, USA, 1997.

[9] A. Miranda and T. Cortes, "Analyzing Long-Term Access Locality to Find Ways to Improve Distributed Storage Systems," in *Proceedings of the 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, Garching, Germany, 2012.

[10] M. Mense and C. Scheideler, "Spread: An adaptive scheme for redundant and fair storage in dynamic heterogeneous storage systems," in *Proceedings of the 19th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, San Francisco, California, USA, 2008.

[11] R. J. Honicky and E. L. Miller, "Replication Under Scalable Hashing: A Family of Algorithms for Scalable Decentralized Data Distribution," in *Proceedings of the 18th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[12] A. Brinkmann, S. Effert, F. Meyer auf der Heide and C. Scheideler, "Dynamic and Redundant Data Placement," in *Proceedings of the 27th IEEE International Conference on Distributed Computing Systems (ICDCS)*, Toronto, Canada, 2007.

[13] G. R. Ganger, B. L. Worthington and Y. N. Patt, "The DiskSim simulation environment version 2.0 reference manual," Carnegie Mellon University/University of Michigan, 1999.

[14] D. Ellard, J. Ledlie, P. Malkani and M. Seltzer, "Passive NFS tracing of email and research workloads," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, 2003.

[15] A. Verma, R. Koller, L. Useche and R. Rangaswami, "Srcmap: Energy proportional storage using dynamic consolidation," in *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, 2010.

List of Acronyms and Abbreviations

AAA	Authorization, Authentication, Accounting.
ACF	Advanced Computing Facility
ADP	Average Dissipated Power
AMD	Advanced Micro Devices
APGAS	Asynchronous PGAS (language)
API	Application Programming Interface
APML	Advanced Platform Management Link (AMD)
ASIC	Application-Specific Integrated Circuit
ATI	Array Technologies Incorporated (AMD)
BAdW	Bayerischen Akademie der Wissenschaften (Germany)
BCO	Benchmark Code Owner
BLAS	Basic Linear Algebra Subprograms
BSC	Barcelona Supercomputing Center (Spain)
CAF	Co-Array Fortran
CAL	Compute Abstraction Layer
CCE	Cray Compiler Environment
ccNUMA	cache coherent NUMA
CEA	Commissariat à l'Energie Atomique (represented in PRACE by GENCI, France)
CGS	Classical Gram-Schmidt
CGSr	Classical Gram-Schmidt with re-orthogonalisation

CINECA	Consorzio Interuniversitario, the largest Italian computing centre (Italy)
CINES	Centre Informatique National de l'Enseignement Supérieur (represented in PRACE by GENCI, France)
CLE	Cray Linux Environment
COFS	Composite File System, a metadata virtualization file system layer
CPU	Central Processing Unit
CSC	Finnish IT Centre for Science (Finland)
CSCS	The Swiss National Supercomputing Centre (represented in PRACE by ETHZ, Switzerland)
CSR	Compressed Sparse Row (for a sparse matrix)
CUDA	Compute Unified Device Architecture (NVIDIA)
DARPA	Defense Advanced Research Projects Agency
DDN	DataDirect Networks
DDR	Double Data Rate
DEISA	Distributed European Infrastructure for Supercomputing Applications. EU project by leading national HPC centres.
DGEMM	Double precision General Matrix Multiply
DIMM	Dual Inline Memory Module
DMA	Direct Memory Access
DNA	DeoxyriboNucleic Acid
DP	Double Precision, usually 64-bit floating point numbers
DRAM	Dynamic Random Access memory
EC	European Community
EESI	European Exascale Software Initiative
EoI	Expression of Interest
EP	Efficient Performance, e.g., Nehalem-EP (Intel)
EPCC	Edinburg Parallel Computing Centre (represented in PRACE by EPSRC, United Kingdom)
EPSRC	The Engineering and Physical Sciences Research Council (United Kingdom)
eQPACE	extended QPACE, name of the FZJ WP8 prototype
Erlang/OTP	Erlang-based Open Telecom Platform
ETHZ	Eidgenössische Technische Hochschule Zuerich, ETH Zurich (Switzerland)
ESFRI	European Strategy Forum on Research Infrastructures; created roadmap for pan-European Research Infrastructure.
EX	Expandable, e.g., Nehalem-EX (Intel)
FC	Fiber Channel
FFT	Fast Fourier Transform
FHPCA	FPGA HPC Alliance
FP	Floating-Point
FPGA	Field Programmable Gate Array
FPU	Floating-Point Unit
FUSE	Filesystem in Userspace, a framework for user-level file systems
FZJ	Forschungszentrum Jülich (Germany)
GASNet	Global Address Space Networking
GB	Giga (= 2^{30} ~ 10^9) Bytes (= 8 bits), also GByte
Gb/s	Giga (= 10^9) bits per second, also Gbit/s
GB/s	Giga (= 10^9) Bytes (= 8 bits) per second, also GByte/s
GCS	Gauss Centre for Supercomputing (Germany)
GDDR	Graphic Double Data Rate memory

GÉANT	Collaboration between National Research and Education Networks to build a multi-gigabit pan-European network, managed by DANTE. GÉANT2 is the follow-up as of 2004.
GENCI	Grand Equipement National de Calcul Intensif (France)
GFlop/s	Giga (= 10^9) Floating point operations (usually in 64-bit, i.e. DP) per second, also GF/s
GHz	Giga (= 10^9) Hertz, frequency =10^9 periods or clock cycles per second
GigE	Gigabit Ethernet, also GbE
GLSL	OpenGL Shading Language
GNU	GNU's not Unix, a free OS
GPFS	A commercial parallel file system
GPGPU	General Purpose GPU
GPU	Graphic Processing Unit
GS	Gram-Schmidt
GWU	George Washington University, Washington, D.C. (USA)
HBA	Host Bus Adapter
HCA	Host Channel Adapter
HCE	Harwest Compiling Environment (Ylichron)
HDD	Hard Disk Drive
HE	High Efficiency
HMM	Hidden Markov Model
HMPP	Hybrid Multi-core Parallel Programming (CAPS enterprise)
HP	Hewlett-Packard
HPC	High Performance Computing; Computing at a high performance level at any given time; often used synonym with Supercomputing
HPCC	HPC Challenge benchmark, http://icl.cs.utk.edu/hpcc/
HPCS	High Productivity Computing System (a DARPA program)
HPL	High Performance LINPACK
HT	HyperTransport channel (AMD)
HWA	HardWare accelerator
IB	InfiniBand
IBA	IB Architecture
IBM	Formerly known as International Business Machines
ICE	(SGI)
IDRIS	Institut du Développement et des Ressources en Informatique Scientifique (represented in PRACE by GENCI, France)
IEEE	Institute of Electrical and Electronic Engineers
IESP	International Exascale Project
IL	Intermediate Language
IMB	Intel MPI Benchmark
I/O	Input/Output
IOR	Interleaved Or Random
IPMI	Intelligent Platform Management Interface
ISC	International Supercomputing Conference; European equivalent to the US based SC0x conference. Held annually in Germany.
IWC	Inbound Write Controller
JSC	Jülich Supercomputing Centre (FZJ, Germany)
KB	Kilo (= 2^{10} ~10^3) Bytes (= 8 bits), also KByte
KTH	Kungliga Tekniska Högskolan (represented in PRACE by SNIC, Sweden)
LBE	Lattice Boltzmann Equation

LINPACK	Software library for Linear Algebra
LLNL	Laurence Livermore National Laboratory, Livermore, California (USA)
LQCD	Lattice QCD
LRZ	Leibniz Supercomputing Centre (Garching, Germany)
LS	Local Store memory (in a Cell processor)
Lustre	A parallel file system
MB	Mega (= 2^{20} ~ 10^6) Bytes (= 8 bits), also MByte
MB/s	Mega (= 10^6) Bytes (= 8 bits) per second, also MByte/s
MDT	MetaData Target
MFC	Memory Flow Controller
MFlop/s	Mega (= 10^6) Floating point operations (usually in 64-bit, i.e. DP) per second, also MF/s
MGS	Modified Gram-Schmidt
MHz	Mega (= 10^6) Hertz, frequency =10^6 periods or clock cycles per second
MIPS	Originally Microprocessor without Interlocked Pipeline Stages; a RISC processor architecture developed by MIPS Technology
MKL	Math Kernel Library (Intel)
ML	Maximum Likelihood
Mop/s	Mega (= 10^6) operations per second (usually integer or logic operations)
MoU	Memorandum of Understanding.
MPI	Message Passing Interface
MPP	Massively Parallel Processing (or Processor)
MPT	Message Passing Toolkit
MRAM	Magnetoresistive RAM
MTAP	Multi-Threaded Array Processor (ClearSpead-Petapath)
mxm	DP matrix-by-matrix multiplication mod2am of the EuroBen kernels
NAS	Network-Attached Storage
NCF	Netherlands Computing Facilities (Netherlands)
NDA	Non-Disclosure Agreement. Typically signed between vendors and customers working together on products prior to their general availability or announcement.
NoC	Network-on-a-Chip
NFS	Network File System
NIC	Network Interface Controller
NUMA	Non-Uniform Memory Access or Architecture
OpenCL	Open Computing Language
OpenGL	Open Graphic Library
Open MP	Open Multi-Processing
OS	Operating System
OSS	Object Storage Server
OST	Object Storage Target
PCIe	Peripheral Component Interconnect express, also PCI-Express
PCI-X	Peripheral Component Interconnect eXtended
PGAS	Partitioned Global Address Space
PGI	Portland Group, Inc.
pNFS	Parallel Network File System
POSIX	Portable OS Interface for Unix
PPE	PowerPC Processor Element (in a Cell processor)
PRACE	Partnership for Advanced Computing in Europe; Project Acronym
PSNC	Poznan Supercomputing and Networking Centre (Poland)
QCD	Quantum Chromodynamics

QCDOC	Quantum Chromodynamics On a Chip
QDR	Quad Data Rate
QPACE	QCD Parallel Computing on the Cell
QR	QR method or algorithm: a procedure in linear algebra to compute the eigenvalues and eigenvectors of a matrix
RAM	Random Access Memory
RDMA	Remote Data Memory Access
RISC	Reduce Instruction Set Computer
RNG	Random Number Generator
RPM	Revolution per Minute
SAN	Storage Area Network
SARA	Stichting Academisch Rekencentrum Amsterdam (Netherlands)
SAS	Serial Attached SCSI
SATA	Serial Advanced Technology Attachment (bus)
SDK	Software Development Kit
SGEMM	Single precision General Matrix Multiply, subroutine in the BLAS
SGI	Silicon Graphics, Inc.
SHMEM	Share Memory access library (Cray)
SIMD	Single Instruction Multiple Data
SM	Streaming Multiprocessor, also Subnet Manager
SMP	Symmetric MultiProcessing
SNIC	Swedish National Infrastructure for Computing (Sweden)
SP	Single Precision, usually 32-bit floating point numbers
SPE	Synergistic Processing Element (core of Cell processor)
SPH	Smoothed Particle Hydrodynamics
SPU	Synergistic Processor Unit (in each SPE)
SSD	Solid State Disk or Drive
STFC	Science and Technology Facilities Council (represented in PRACE by EPSRC, United Kingdom)
STRATOS	PRACE advisory group for STRAtegic TechnOlogieS
STT	Spin-Torque-Transfer
TARA	Traffic Aware Routing Algorithm
TB	Tera (= 240 ~ 1012) Bytes (= 8 bits), also TByte
TCO	Total Cost of Ownership. Includes the costs (personnel, power, cooling, maintenance, ...) in addition to the purchase cost of a system.
TDP	Thermal Design Power
TFlop/s	Tera (= 1012) Floating-point operations (usually in 64-bit, i.e. DP) per second, also TF/s
Tier-0	Denotes the apex of a conceptual pyramid of HPC systems. In this context the Supercomputing Research Infrastructure would host the Tier-0 systems; national or topical HPC centres would constitute Tier-1
UFM	Unified Fabric Manager (Voltaire)
UNICORE	Uniform Interface to Computing Resources. Grid software for seamless access to distributed resources.
UPC	Unified Parallel C
UV	Ultra Violet (SGI)
VHDL	VHSIC (Very-High Speed Integrated Circuit) Hardware Description Language

Executive Summary

This report documents the research and development carried on within Task 12.4. The main goal of the task is to identify and address some open issues in file systems for multi-petascale and exascale facilities, aiming to the development of solutions that can be applied to the Lustre file system.

The addressed issues can be classified into two main areas: metadata management and data management. Metadata handling involves dealing with huge numbers of files and their hierarchical organization according the user's view (including directory management and file attributes). Data handling deals with the storage of file contents and management data; this includes, in particular, techniques for automatic (self-tuned) placement of data on a system with many heterogeneous devices, aiming at maximizing bandwidth and minimizing response time.

The work carried on in the area of metadata management included the observation, measurement and study of a large scale system currently in production, in order to identify the key metadata-related issues; the development of a prototype aimed to improve the metadata behaviour in such system and also to provide a framework to easily deploy novel metadata management techniques on top of other systems; the measurement and study of specially deployed Lustre and GPFS prototypes to validate the presence of the metadata issues observed in current in-production systems; and finally the porting of the framework prototype to test novel metadata management techniques on the Lustre prototype facility.

In this line we have observed that in both Lustre and GPFS there are some scalability issues that reduce the performance of metadata operation when many files are used by the applications or when the number of accessing clients grows. The most important observation is that the number of files needed for the problem to appear is only a few hundreds and the number of clients a few dozens. This clearly shows that the problem needs to be addressed.

After our mechanism has been added to the GPFS system, we have observed that the decrease in performance that appears in the evaluated cases disappears making the system much more scalable with the number of files and clients. The main reason for this beneficial effect is that our middleware is able to convert not optimized cases (from GPFS point of view) into the optimized cases of GPFS.

Regarding data management, the work carried on in the present Task is based in the study of the limitations of current data distribution strategies, especially when dealing with sustained growth of storage capacity.

The results consist of a proposal for a novel data re-distribution technique for increased storage capacity, aimed to maximize bandwidth and responsiveness while minimizing the cost of data re-distribution. This technique that redistributes data using an approach that mixes the two current trends (deterministic and randomized placement) is able to achieve a perfect redistribution of data with minimal data movement (like the randomized approaches) and without the negative effects in metadata size or computing time of traditional randomized approaches.

It is also a contribution of this work a new level in the caching hierarchy that uses a tiny portion of all available storage systems to cache data and achieves performance results similar to the ones obtained when the data is perfectly well distributed, but starting from data very badly distributed.

1. Introduction

The purpose of this report is to document the results obtained in Task 12.4. The final goal of the task is to identify the open issues related to file systems for multi-petascale and exascale facilities, and propose novel solutions that can be applied to Lustre, enabling it to manage a huge number of files on a system with many heterogeneous devices while efficiently delivering huge data bandwidth and low latency, minimizing the response time.

The work of the task follows two main lines: metadata handling and data storage handling. Metadata management involves providing support for a large number of files, offering a consistent view of their attributes and organization, according to the semantics expected by users. On the other hand, data management focuses on optimizing the placement of data on a large set of heterogeneous storage devices, taking into account its dynamic nature (due to the addition of new devices to both increase the storage capacity and the replacement of old devices).

The issues we try to solve are not specific to Lustre: all modern file systems aiming to provide a storage solution for very large scale systems share similar goals and face related issues. For this reason, even if the final goal is to provide novel mechanisms to enrich the Lustre prototype, we have studied and taken advantage of the experience of large-scale production facilities also using different file systems (e.g. GPFS[1]), and we have also evaluated policies in simulation environments. Consequently, the results we have obtained could also be applied directly (or ported with minimum effort) to file systems other than Lustre.

In general, file systems try to adapt to the new demands by specializing and targeting specific kinds of workloads. As a counterpart, there is a cost in terms of performance for non-optimized cases.

A usual way to deal with non-optimized cases is to add even more specific optimizations, either by means of files system specific modifications (e.g. parallel creations policies in PVFS2 [2]), or by means of a middleware targeted to fulfil the needs of specific classes of applications (e.g. MPI-IO implementations using hierarchical striping for Lustre [3]). Unfortunately, this approach does not deal with performance penalties caused by unforeseen or inadequate access patterns from arbitrary applications, which are likely to occur in heterogeneous workloads.

Our approach to handle the metadata issues consists of placing a layer on top of the file system for decoupling the user view from the actual low-level file system organization, allowing us to convert the application access patterns into something that can be dealt with by underlying file system without harming the performance (instead of trying to adapt the file system to any possible workload). In other words, the mechanism consists of improving the file system behaviour by avoiding bottlenecks caused by application patterns, instead of focusing only on optimizing the file system for a very specific workload.

Running on top of the file system (instead of integrating new code into it) makes development much easier and simplifies the integration of other state-of-the-art technologies which may help to address the issues under study. Additionally, it allows us to easily adapt the same solutions to multiple file systems and environments.

On the other hand, regarding data, we attack the problem by proposing algorithms that can easily adapt their placement to the changing needs of the load and the changing resources. Both the new placement algorithm and the new hierarchy level are general enough to fit any parallel file system and should be implemented as part of the data placement modules of such systems.

Section 2 describes the identified metadata scalability issues. Section 3 describes the COFS (Composite File System) prototype and how it is used as a framework to integrate solutions for the scalability issues. Section 4 summarizes the data management issues in large-scale systems. Section 5 describes the proposals to reduce the cost of data allocation and re-distribution. Section 6 exposes some details about the evaluations and experiments performed, both in actual systems and in simulated environments. Finally, section 7 summarizes the conclusions.

This document is intended for the authors of PRACE deliverables and for parallel file system managers who can understand some limitations of their systems and see some mechanisms to mitigate such problems. Finally, and especially the second part of the deliverable, is aimed at parallel file system developers that may be interested in including the proposed simulated techniques into real file systems.

2. Metadata Management Issues

Parallel file systems usually keep pace with high performance computing clusters by incorporating optimizations targeted at specific workloads. Unfortunately, the growing number of large scale applications increases workload heterogeneity, generating a gap between how file systems work, and how users expect them to behave.

High performance computing is rapidly evolving into large aggregations of computing elements in the form of big clusters. 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 rising. Trying to keep pace with these developments, parallel file systems try to provide mechanisms for distributing data across a range of storage devices and making them readily available to the computing elements.

At the other end, the final user's view of the storage systems has not changed significantly: for the usual case, files are organized in a hierarchical name space, much in the same way as they were placed in a classical local file system using an attached disk in a single computer.

In this classical view, a directory was often tacitly used as a hint to indicate *locality*. Indeed, it is not surprising to find a directory containing files that are going to be used together, or in a very related way (for example, a directory containing a program's code, the corresponding executable, some configuration files and maybe the output of its execution).

Trying to exploit this affinity, file systems tended to group together the management information (the metadata) about directory contents. This was favoured by the fact that this information is relatively small, so that it is feasible to pack together information about access permissions, statistics, and also the physical location of the data, not only for a single file, but also for a set of related files (i.e. for files present in the same directory).

The approach of treating directory contents as a group of related objects with similar properties is still present in modern file systems. It is common to see parallel file systems to use directories as mount points to access different volumes or partitions with different functionalities or, in finer grain file systems, to be able to specify directory-wide rules that apply to directory contents.

The problem appears when, in large clusters with parallel file systems, directory contents are *semantically* related according user's view, without that meaning that they are going to be used together or in similar ways from the file system perspective. Reusing the example above, now the program code in a directory will be compiled and linked, probably in a single node, to generate a binary that will be read and executed simultaneously in 2,000 nodes, each of

them generating an output file to be left in the same directory, which will be collected and read by a single post-processing tool generating some summary information. Files are indeed related, but patterns of use are totally dissimilar.

Parallel file systems trying to keep close the apparently related fragments of metadata will end up trying to keep consistent a relatively small pack of miscellaneous information (which is not easy to distribute and share) while it is being simultaneously used, and probably modified, by a large number of nodes. As a result, the pressure on metadata handling in large scale systems will rise up, producing delays comparable to the actual data transfer times from and to the storage systems, and jeopardizing the overall system performance.

The metadata issue has been confirmed by observations in actual systems, showing that lack of synergy between file systems and the multiplicity of applications running on them is increasing the pressure on metadata management, up to the level of becoming a significant performance-killer.

The following subsections give more details about the observations of metadata behaviour in both GPFS and Lustre file systems.

2.1. Metadata Scalability Issues in GPFS

Parallel applications on large scale parallel systems expect to be able to perform I/O simultaneously from all the nodes as they would if it was a single node (i.e. efficiently, in parallel and keeping the consistency). Not many file systems are able to provide such support to applications in a reliable way. Together with Lustre, GPFS is another mature file system aimed to large distributed clusters which has been adopted at many high performance computing centers.

As Lustre, GPFS offers a standard POSIX interface, while having the possibility to use non-POSIX advanced features for increased performance (e.g. for MPI-IO). Nevertheless, their architectural approaches differ significantly. For this reason, studying its behaviour and comparing it with Lustre's provides useful insights regarding the different techniques available to deal with large cluster file systems and their impact on performance.

GPFS uses block-based storage (contrary to Lustre's Object Storage Devices [4]). A typical configuration consists of clients which access to a set of file servers connected to the storage devices via a Storage Area Network (SAN). Metadata is also distributed and consistency is guaranteed by distributed locking, with the possibility of delegating control to a particular client to increase performance in case of exclusive access.

We have been able to observe important performance drops in large production clusters using GPFS with heterogeneous workloads, and we have been able to track the causes back to metadata management issues that are related to the way in which applications use the file system:

- Large parallel applications usually create per-node auxiliary files, and/or 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.

In both cases, typical modus operandi ends up creating large amounts of files in the same directory; and very large directories, especially when populated in parallel, require GPFS to use a complex and costly locking mechanism to guarantee the consistency, resulting in far-from-optimal performance. For example, a parallel application spanning across a large number of nodes can use an important portion of its execution time creating and writing checkpoint files (significantly greater than what should be expected for the simple transfer of data.) To mention another example, collection software used by computer science researchers to obtain per-node application execution traces for performance analysis also suffer from this inadequate metadata handling.

As an additional concern, 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.

We have conducted a series of experiments in a GPFS cluster to confirm that metadata handling was a significant cause of performance drops. The situation we wanted to evaluate essentially involved parallel metadata operations, so we used *Metarates* [5] 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. We used this application to invoke *create*, *stat*, and *utime* on a number of files from the same directory in parallel. The issues identified in the production GPFS cluster have then been used to locate potential issues in the Lustre prototype.

The measurements for GPFS have been carried out in a cluster of IBM JS21 blades, with 2 dual core processors PPC970MP at 2.3 GHz and 8 GB of RAM per blade. The interconnection network is a 1Gb Ethernet.

The observations indicate that an important portion of the relatively large operation times is consumed not by actual information being transmitted from the server to the clients, but by consistency-related traffic (even when each process works on a different set of files). That assumption would give our virtualization layer enough room to obtain a good speed-up by reorganizing the file layout and reduce the synchronization needed among GPFS clients.

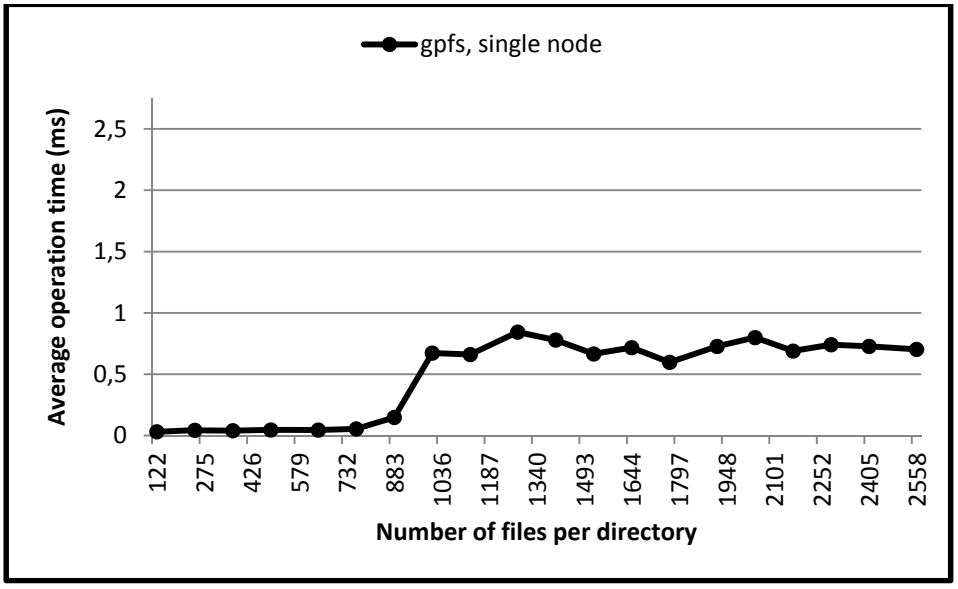


Figure 1 *Utime* cost for a single GPFS client node

Figure 1 shows the average time to fulfil an *utime* request (changing the timestamp indicating when the file was last modified) on a file in a directory accessed by a single client node. The low values for small directories (about 45 microseconds) are the cost of the request when the GPFS delegation mechanism is active (control is transferred to the client node, so that operations can be carried on locally, without server synchronization) and represent a lower boundary for the operation cost; beyond 1024 entries per directory, delegation is no longer active and the operation requires remote server intervention, resulting in larger operation times.

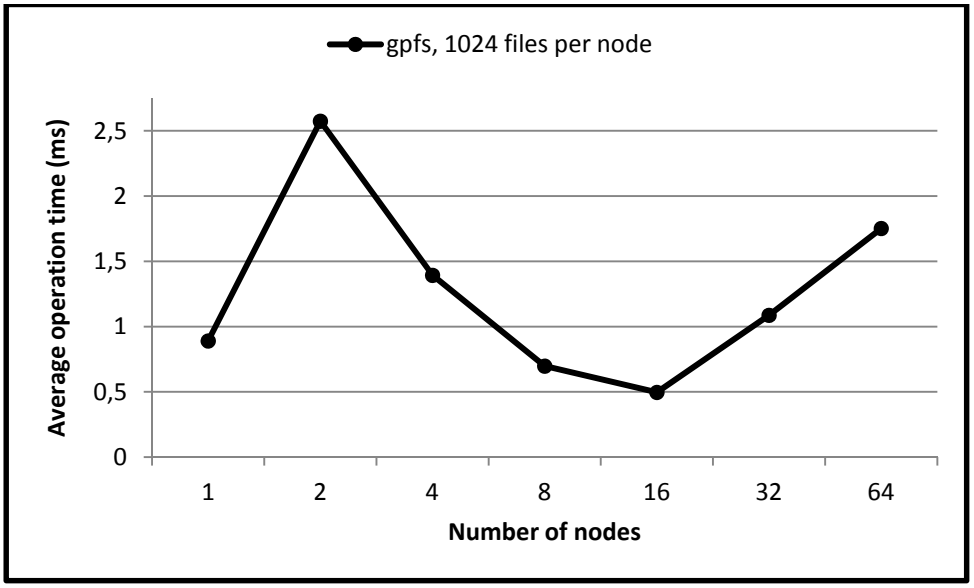


Figure 2 *Utime* cost in GPFS using multiple client nodes

The operation average times when performing simultaneous *utime* calls from different nodes are shown in Figure 2. Ideally, the cost should be around 0.75ms (the cost in a single node when a round-trip to the server is done). What we actually observe is that the cost with 2 nodes is much higher, and progressively decreases when we increase the parallelism, reaching an optimum value for 16 nodes. Beyond that point, the system does not scale anymore and increasing the number of nodes just increases the access conflicts, reducing the performance.

Therefore, one of the goals of our work is to reorganize accesses so that conflicts are avoided, and the operation times are kept near the optimum values, regardless of the number of nodes used or the size of the directories involved.

Another interesting observation from GPFS is that the average parallel creation time differs depending on the files being created on a single shared directory or on unique directories per processor. Figure 3 shows the large delays observed when creating files from different nodes in the same directory. It is important to mention that each node creates a disjoint set of files; so, the only cause of poor behaviour is the management of a shared metadata management structure: the directory. By changing the way the metadata is organized and handled, it should be possible to reduce this cost to levels similar to using unique directories per processor.

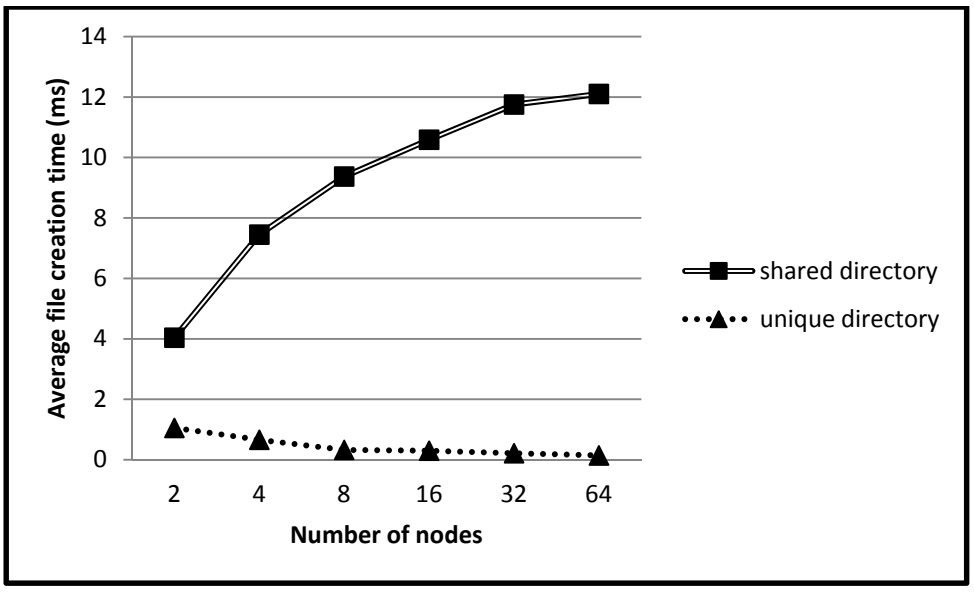


Figure 3 Average parallel creation times in GPFS (1024 files per node)

The experiences and observations in the GPFS cluster have served two purposes: on one side, they have guided the analysis and study of potential metadata-related performance issues in the Lustre prototype for exascale systems (that will be commented in the following subsection); on the other side, they have inspired the solutions developed to overcome metadata performance issues (explained in Section 3).

2.2. Metadata Scalability Issues in Lustre

Lustre [4] is a parallel file system able to offer a POSIX compliant interface and based on three types of components: the clients (nodes accessing the file system), the storage servers where the data resides (based on object storage devices) and a metadata server responsible for name space, access rights and consistency management.

One of the key characteristics of Lustre regarding metadata management is that it relies on a single metadata server (possibly replicated for failover replacement) to handle all metadata. This approach simplifies consistency management (compared to a fully distributed locking mechanism for metadata management – as in GPFS).

We have executed the Metarates benchmark also in the Lustre prototype in order to compare them with the observations in GPFS and, given the different strategies used by both systems, and obtained thorough information about the causes of metadata performance issues and which are the best ways to neutralize them. The client nodes of the Lustre prototype are Bullx Inca Nehalem-based blade nodes (8 X5560 processors at 2.80 GHz).

Figure 4 shows the average time spent in a file system request (*utime*) in a single Lustre client. Note that operation times are not directly comparable to GPFS due to differences in the hardware used (both the nodes and the network), but the observations regarding the trends are still valid.

One of the things to remark in comparison with GPFS is the absence of large differences in operation times depending on the number of entries of the directory (compare Figure 1 with Figure 4). This was expected, as Lustre does not use a delegation mechanism transferring the metadata control to the client (as GPFS does), so there is not a „local-like“ behaviour and all requests go to the remote metadata server.

As the multi-core hardware in the Lustre prototype allows us to efficiently execute several processes in the same node, we have executed the benchmarks using 1 and 8 processes per node. From the data in Figure 4 we can see that the Lustre client is able to parallelize requests from different processes, resulting in an increased performance.

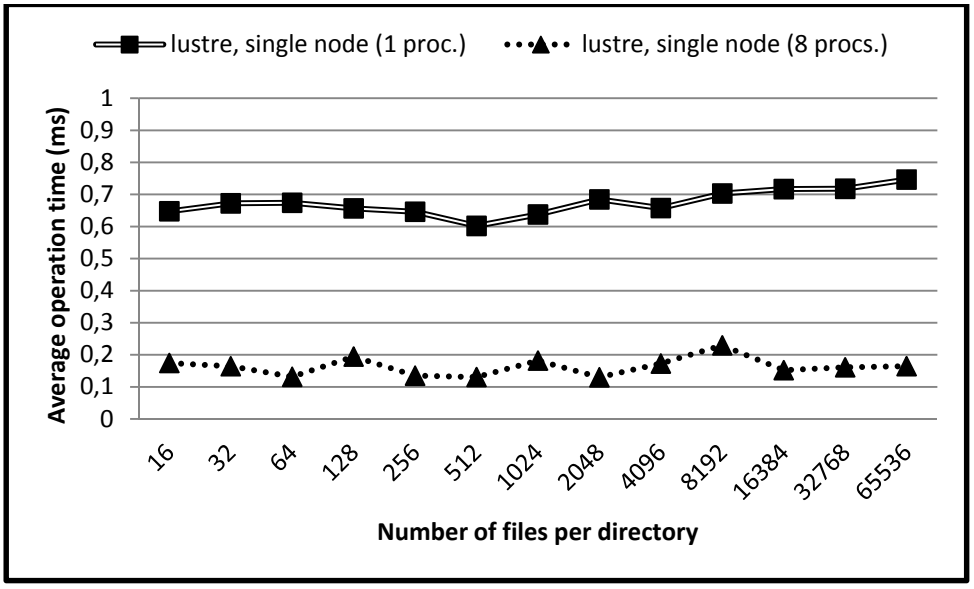


Figure 4 *Utime* cost for a single Lustre client node

The improved behaviour of Lustre when using multiple processes per node tends to converge with the single node behaviour when the number of nodes increases (see Figure 5).

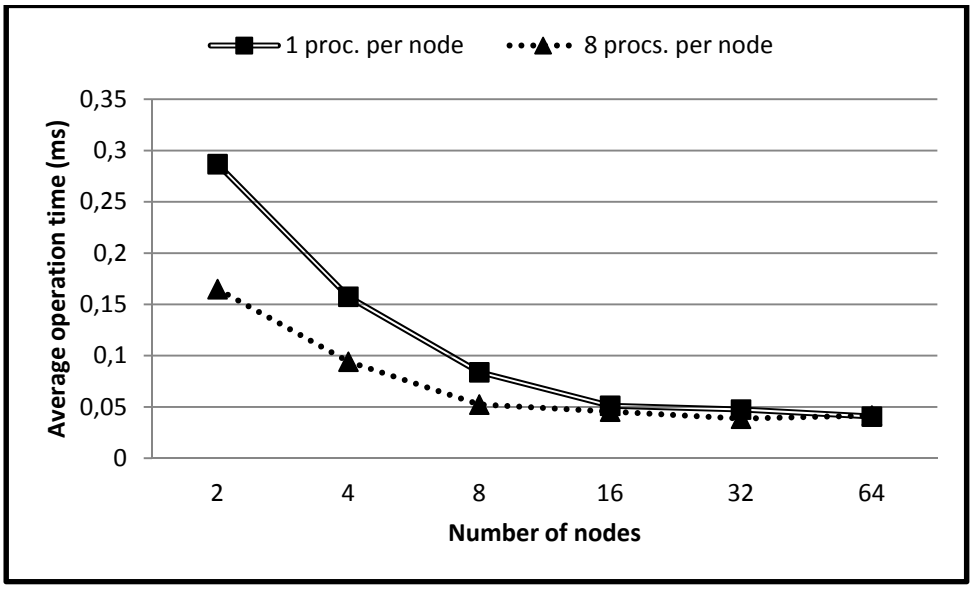


Figure 5 *Utime* cost for multiple Lustre client nodes (1024 files per node)

When comparing with the *utime* results in GPFS (see Figure 2), we see that the Lustre prototype does not seem to suffer from clear performance degradation when we increase the number of nodes (while GPFS does). We suspected that this difference was caused not only by the different file system, but also due to different hardware and network configuration that was pushing the issue beyond the number of nodes used in the experiments. We had the opportunity to validate this assumption by running the benchmarks in a Lustre system larger than our main Lustre prototype.

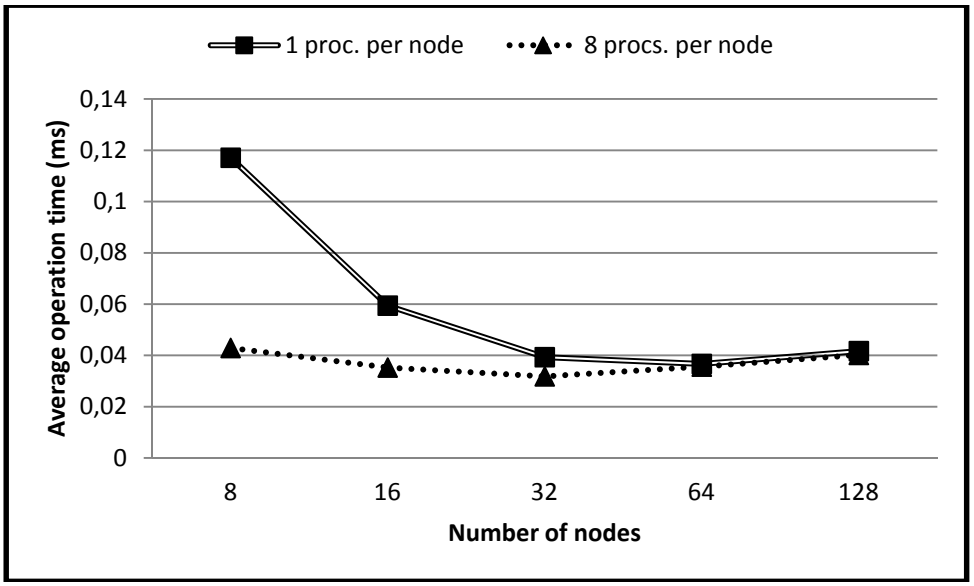


Figure 6 Lustre *utime* behaviour in larger systems (1024 files per node)

Figure 6 shows that beyond 64 nodes, there is an indication of an increase of the effective average operation time (resulting in a reduction of the performance). Possibly, this effect will become larger with further increases in the number of nodes (but this is still to be confirmed by experiments).

Regarding file creation, Lustre also shows differences between the creation of files in a shared directory and in a unique directory per process (see Figure 7), though they are not so significant as they were in GPFS.

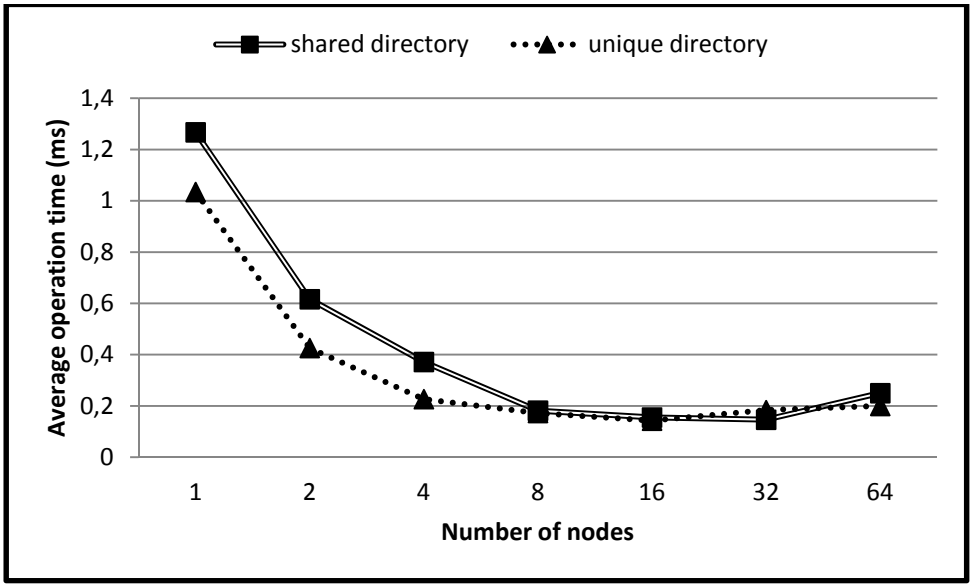


Figure 7 Average parallel creation times in Lustre (8 procs. per node, 1024 files per node)

Another important observation from Figure 7 is that the creation operation time is considerably longer than other metadata operations (such as *utime*). This may be an indication that there is room for improvement in file creation handling.

Finally the data also offers a hint of performance degradation beyond 32 nodes, suggesting the file system behaviour may not scale for a much larger number of nodes.

2.3. Conclusions

From the experiments, we have learned some facts about file system behaviour that may help us to optimize the Lustre prototype and improve its scalability.

The sequential behaviour of Lustre regarding metadata operations is not significantly affected by the number of files in the directory (differing from GPFS). The only exception is the *create* request: the optimal creation ratio is reached at 1024-2048 entries per directory, and stabilizes beyond that point.

Parallel creations on shared directories suffer from overhead with respect to parallel creations in separate directories per process. The overhead is much more significant in GPFS than in Lustre. One of the possible causes is that GPFS has a delegation mechanism that allows transferring the control of metadata operations from the server to the client temporarily, allowing near-local operation times when a node works on a directory which is uniquely accessed by a single node. On the contrary, Lustre does not have such a mechanism and operations to both unique and shared directories need to go to the remote metadata server (resulting in less significant improvements for non-shared operations).

Parallel operations in Lustre scale quite well up to an optimum value about 16-32 nodes (depending on the operation and conditions) but show hints of performance degradation beyond that point. The goal should be avoiding the performance degradation and even continue the performance improvement with more nodes.

Decoupling the name space from the low-level file system layout could be a way to mitigate the issues. A virtual layer would offer large shared directory views while internally splitting them to take advantage of non-shared operations. Moreover, this layer could be used to easily add delegation mechanisms and coalesce requests, reducing round-trips to the metadata server. The Composite File System (COFS), explained in section 3 was developed to validate this approach. Section 6.1 shows some of the results obtained up to now (for the GPFS file system).

3. Tools for Metadata Management: the COFS Framework

Given the observations on metadata behaviour described in previous sections, we have developed the Composite File System (COFS) as a prototype to validate the assumption that it is possible to boost the performance of a parallel file system by decoupling metadata, and name space handling, from the underlying directory layout. COFS acts as a metadata management layer on top of the file system, enabling the improvement of its behaviour under high pressure situations without harming the other aspects of file system performance.

In particular, COFS has been used to evaluate the performance benefits of separating the user view from the underlying file system structure on our systems under study. In this section we describe its main features, focusing on the implementation aspects that are relevant to explain the experimental results.

3.1. Principles

The main 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 the files. In particular, this allows us to present the user with a virtual view of the file system directories, while the actual layout can be optimized for the underlying file system.

Regarding functionality, POSIX compliance was a strong design requirement. Apart from the fact that this is still the dominant model for most applications, our generic goal was reducing the operational 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 (as is the case with Lustre), then COFS should also be able to deal with it. The current implementation fully supports POSIX except for some functionality that was not relevant for our present work (specifically, named pipes).

Another important point to mention is that COFS is implemented as a user-level FUSE (Filesystem in USErspace [6]) daemon, and it is independent from the underlying file system. The reasons for this are two-fold. First, even if one of the original motivations of this work was mitigating potential performance drops on a specific file system, we believe that equivalent issues affect other file systems; so, the solving mechanism should be generic enough to be applied to any file system. Second, we plan to deploy our framework in production-grade clusters, and having a user-land drop-in package without hard requirements on kernel modifications, configurations or complex software packages makes it much easier to have access to such environments, as the potential impact on the rest of the system is minimal.

The COFS framework does not directly deal with low-level data storage. There is no explicit 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 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).

Regarding security aspects, COFS relies on the underlying infrastructure: local file system operations are already protected by FUSE's in-kernel support; communications with the metadata service make use of the authentication mechanisms provided by the Erlang/OTP environment [7].

3.2. Architecture

Figure 8 shows how the COFS framework integrates with an existing file system environment. An original parallel file system is served by 3 file system servers, and *n* clients contact such servers through the network. COFS introduces an extra layer on each file system client, providing a virtual view of the file system layout. Metadata information is handled by an additional server 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.

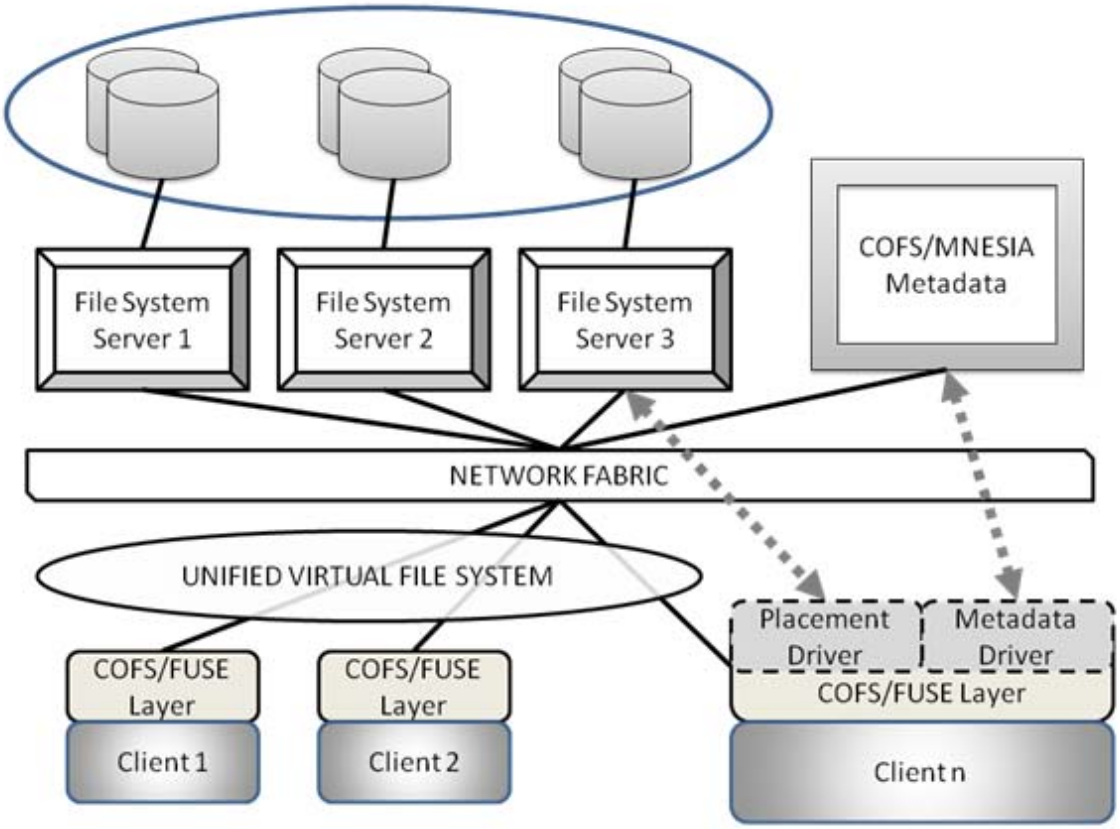


Figure 8 Parallel file system architecture augmented with COFS virtualization layer

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, 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 user-space 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 placement and metadata drivers) with well-defined interfaces. The placement driver is responsible for mapping 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 placement 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.

3.3. Metadata service details

How to handle metadata is an important factor that must be considered carefully when dealing with parallel and distributed file systems. This also applies to the COFS framework, as the

separation of metadata and name space handling from the actual data layout is a key feature to obtain performance benefits.

Currently adopted solutions for metadata range from having a single centralized metadata server (like Lustre) to fully distributing metadata (such as GPFS). While the first have simplicity on their side, centralized approaches are suspected to become a bottleneck and hinder scalability; on the other hand, the latter approach requires distributed locking/leasing techniques to keep the coherence and complex fault detection and recovery mechanisms. Other options lie in the middle, like explicitly dividing the system into smaller partitions (Panasas) or relaxing consistency or caching to avoid synchronization needs (PVFS2). The distributed directory service for Farsite also explores several techniques for partitioning file system metadata while mitigating synchronization hotspots.

In COFS, we have taken 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 file system partitions or separate volumes).

To this end, we chose the Mnesia database, which is part of the Erlang/OTP environment. 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, and the COFS metadata driver on each client simply forwards the requests to the server and handles metadata leases.

The server also keeps the current working set of metadata information as a cache of active objects to reduce the pressure on the backend database engine, using a concurrent caching mechanism similar to the one described by Jay Nelson. Although our performance tests show that a single node is enough to handle the metadata, the used algorithm would be compatible with a distributed multi-node Erlang system, if needed.

4. Data Management Issues

Metadata management represents just one of the aspects that have to be addressed in exascale systems. Data storage and handling constitutes the other important set of issues that must be carefully considered in order to avoid unacceptable performance losses.

Indeed, the ever-growing creation of massive amounts of data expected in exascale systems requires highly scalable solutions. The most flexible approach is to use a pool of storage devices that can be expanded and scaled down as needed by adding new storage devices or removing older ones. Nevertheless, such an approach brings out the following challenges:

1. These storage systems will inevitably be composed of a collection of heterogeneous hardware: as capacity requirements grow, new storage devices will be added to cope with demand, but it is unlikely that these new devices have the same capacity and/or performance that those currently in the storage system. Furthermore, when disks fail they are usually replaced by faster and larger ones, since it ends up being cheaper than finding a particular model of drive. In the long run, any mass storage system will have

to cope with a myriad of devices (SATA/SCSI drives, optical tapes, SSDs...) with very different performance characteristics and capacities.

2. Storage systems must be able to scale according to the needs of the users, and they must do so in an efficient manner. Whenever new devices are added to the system large amounts of data must be migrated in order to keep the system's load balanced across devices. For large-scale storage systems the amounts of data moved can be enormous and the required migration times impractical.

In summary, data must be carefully balanced to maintain acceptable access ratios in very large scale systems; nevertheless, the supporting hardware will need to be expanded and replaced continuously, which means that large amounts of existing data will have to be redistributed and moved from the old devices to the new ones in a proper way to maintain performance characteristics.

Research has taken some very different directions in order to solve the problem of scalability in heterogeneous storage. Table-based strategies can provide an optimal mapping between data blocks and storage systems, but obviously do not scale to large systems because tables grow linearly in the number of data blocks. Rule-based methods, on the other hand, run into fragmentation problems, so defragmentation must be performed periodically to preserve scalability.

Hashing-based strategies use a hashing function in order to map data blocks with unique identifiers into a set of devices, so that blocks are evenly distributed. Given a static set of devices, it is simple to construct a hash function so that every device gets a fair share of the data load. However, standard hashing techniques do not adapt well to a changing set of devices.

Pseudo-randomized hashing schemes that can adapt to a changing set of devices have been proposed and theoretically analyzed. The most popular is probably Consistent Hashing [8], which is able to evenly distribute single copies of each data block among a set of storage devices and to adapt to a changing number of disks. Nevertheless, these theoretically perfect approaches suffer from heavy memory usage which limits their applicability in real implementations.

5. Proposals for Data Management

5.1. Random Slicing

Random Slicing is a new data distribution strategy that aims to solve the problems mentioned in Section 4 by using a pseudo-randomized distribution of data blocks coupled to a scalable data structure that controls the mapping of sets of blocks to devices.

Random Slicing overcomes the drawbacks of randomized data distribution strategies by incorporating lessons learned from table-based, rule-based and pseudo-randomized hashing strategies. Random Slicing keeps a small table with information about previous storage system insertions and removals that helps to drastically reduce the required amount of randomness in the system and thus reduces the amount of necessary main memory by orders of magnitude.

Random Slicing has been designed to be fair and efficient both in homogeneous and heterogeneous environments and to adapt gracefully to changes in the number of devices. Suppose that we have a random function $h: \{1, \dots, M\} \rightarrow [0,1)$ that maps data blocks uniformly at random to real numbers in the interval $[0,1)$. Also, suppose that the relative capacities for the n given devices are $(c_0, \dots, c_{n-1}) \in [0,1)^n$ and that $\sum_{i=0}^{n-1} c_i = 1$.

The strategy begins by dividing the [0,1) range into intervals and assigning them to the devices currently in the system. Notice that intervals do not overlap and completely cover the [0,1) range. Also note that device i can be responsible for several non-contiguous intervals $P_i = (I_0, \ldots, I_k)$, where $k < n$, which constitute the *partition* of that device. To ensure an even distribution, Random Slicing will always enforce $\sum_{j=0}^{k-1} |I_j| = c_i$.

In an initial phase, i.e. when the first set of storage devices enters the system, each device i is given only one interval of length c_i, since this suffices to maintain a fair distribution of data. Whenever new devices enter the system, however, relative capacities for old devices change due to the increased overall capacity. To maintain this fairness, Random Slicing shrinks existing partitions by splitting the intervals that compose them until their new relative capacities are reached. The new intervals generated are used to create partitions for the newly added devices.

First, the algorithm computes how much partitions should be shrunk in order to keep the fairness of the distribution. Since the global capacity has increased, each partition P_i must be reduced by $r_i = c_i - c_i'$, where c_i' corresponds to the new relative capacity of device i.

Partitions become smaller by releasing or splitting some of their intervals, thus generating *gaps*, which can be used for new intervals. Notice, however, that the strategy's memory consumption directly depends on the number of intervals used and, therefore, the number of splits made in each addition phase can affect scalability. For this reason, the algorithm tries to collect as many complete intervals as possible and will only split an existing interval as a last resort. Furthermore, when splitting an interval is the only option, the algorithm tries to expand any adjacent gap instead of creating a new one.

The partition lengths for the old devices already represent the corresponding relative capacities. It is only necessary to use these gaps to create new partitions for the newly added bins. The strategy proceeds by greedily allocating the largest partitions to the largest gaps available in order to reduce the number of new intervals even more, which ends the process.

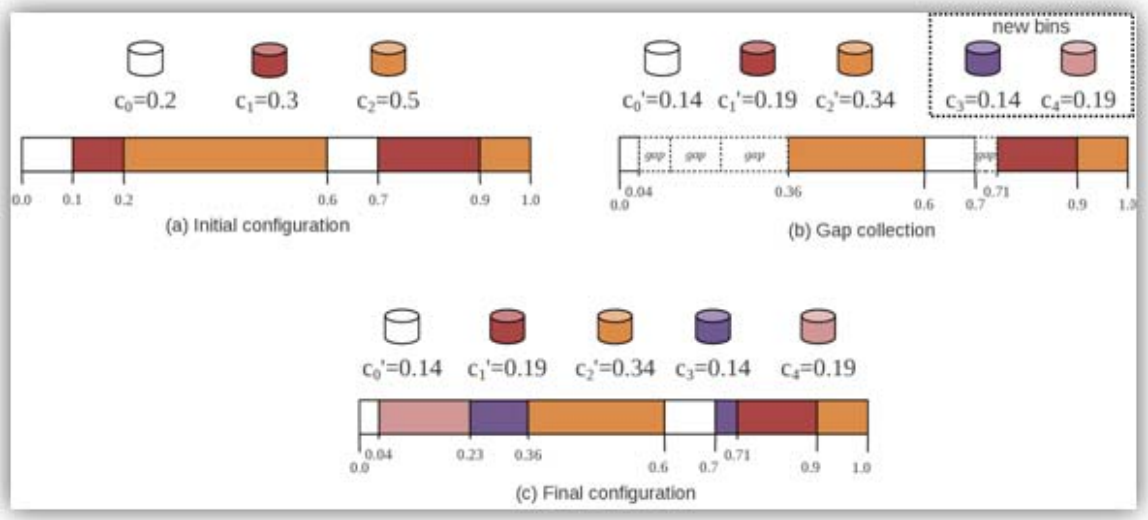


Figure 9 Random Slicing interval reorganization for the new devices

An example of this reorganization is shown in Figure 9,where two new bins B_3 and B_4 (which account for a 50% capacity increase) are added to the bins B_0, B_1, and B_2. Figure 9 (a) shows the initial configuration and the relative capacities for the initial bins. Figure 9 (b) shows that the partition of B_0 must be reduced by 0.06, the partition of B_1 by 0.11, and the one of B_2 by

0.16, whereas two new partitions with a size of 0.14 and 0.19 must be created for B_3 and B_4. The interval $[0.1, 0.2) \in B_1$can be completely utilized, whereas the intervals $[0.0, 0.1) \in B_0$, $[0.2, 0.6) \in B_2$ and $[0.7, 0.9) \in B_1$are split while trying to maximize the length of the created gaps. Figure 9 (c) shows the completed process where, the partition for B_3 is composed of intervals $[0.23, 0.36)$ and $[0.7, 0.71)$, while the partition for B_4 only contains interval $[0.04, 0.23)$.

When all partitions are created, the location of a data block b can be determined by calculating $x = h(b)$ and finding the interval I that contains x, which in turn determines the device. Notice that some blocks will change partition after the reorganization, but as partitions always match their ideal capacity, only a near minimal amount of blocks will need to be reallocated. Furthermore, if $h(b)$ is uniform enough and the number of expected blocks in the system is significantly larger than the number of intervals (both conditions easily feasible), the fairness of the strategy is guaranteed.

5.2. Multi-Zone Self-Caching Data Storage

As shown in [9], some workloads show relevant amounts of long-term locality, that is, data that tends to be accessed continuously for several days or weeks. We have designed and simulated a new allocation strategy that optimizes access times to hot data sets and maintains an appropriate balance of data load and operations across devices.

The basic idea of this strategy is to claim a portion of each device in the storage system and use it to create a storage zone where we can distribute active data. This effectively defines two zones in the storage system: an *archival zone* and a *caching zone*. The strategy also keeps track of changes to the currently active working set and updates the caching zone accordingly.

We consider that data is active when it begins receiving accesses. When this happens, the data block is copied to the caching zone and all subsequent requests are served from it. Once data stops being active it is copied back to the archival zone, where it will remain until it becomes active again. Each zone can be managed with a different data allocation policy that befits the requirements of the data that it contains, therefore combining the advantages of performance-oriented strategies with the cost-capacity benefits of archival-oriented ones.

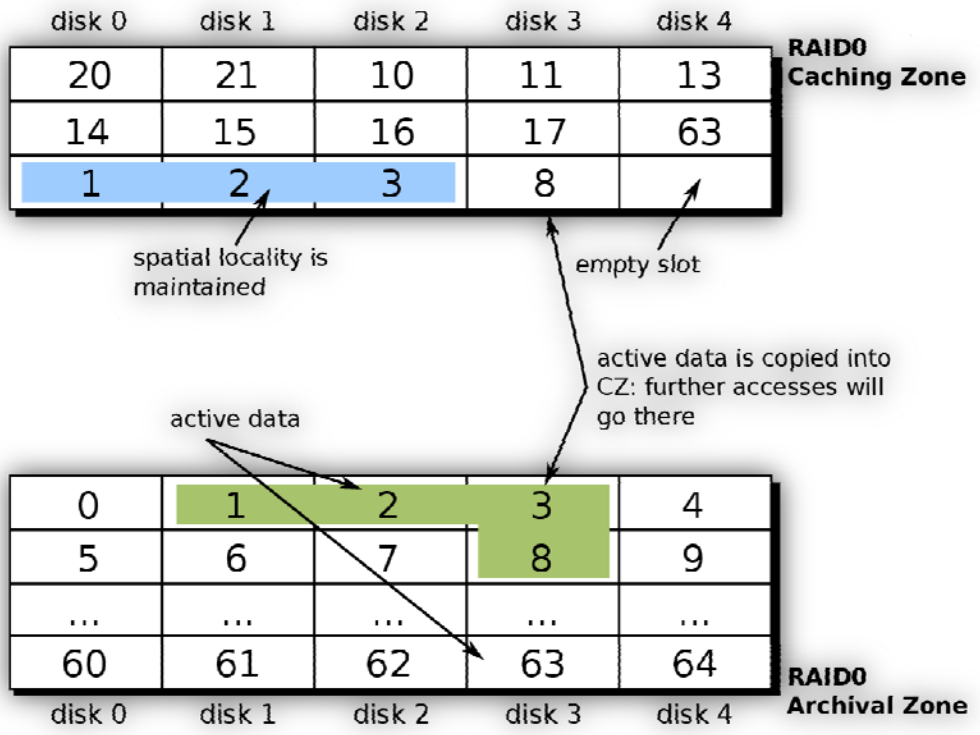


Figure 10 Overview of an hybrid architecture with a RAID0 caching zone

Figure 10 shows an example of an architecture with RAID0 for both the caching zone and the archival zone. The caching zone is assembled with segments from all five disks. Active data is copied to the caching zone, where it is accessed and updated. When a data block is no longer active, the original block is updated and the copy is removed from the caching zone.

6. Evaluations and Experimental Results

6.1. COFS

We deployed the COFS prototype to verify that it could mitigate some of the metadata performance issues of the parallel file systems and that the benefits obtained where significant enough to compensate the cost of adding an extra layer on top of the file system. Experiments were run using GPFS as the underlying system in the same hardware used for the initial measurements (described in section 2.1). The reason behind the selection of GPFS instead of Lustre was because the installation we planed to use for the Lustre experiments was delayed by six months and we only had time to measure the performance, but not to apply the techniques to the system. This evaluation results over Lustre will be presented in a later progress report.

6.1.1. Metadata virtualization results

The following measures of COFS over GPFS have been obtained in the same system described in Section 2.1. Additionally, COFS uses IP over Myrinet for communicating metadata service and clients.

Figure 11 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 10 ms in 16 nodes for GPFS to about 1 ms when using COFS over GPFS. The numbers for pure GPFS were limited to 64 nodes because beyond that point the system was suffering severe performance problems when running the benchmarks (that issue was not present when using COFS over GPFS).

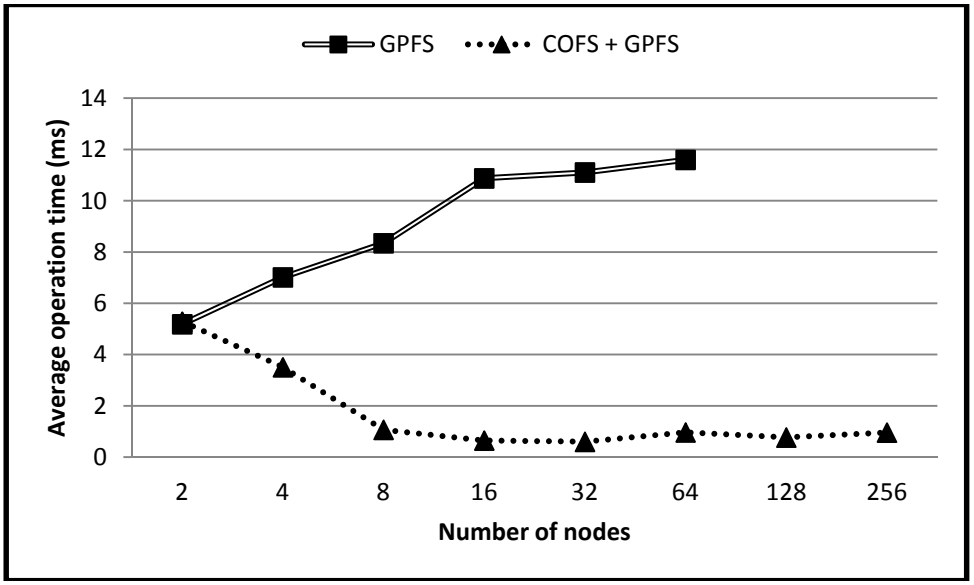


Figure 11 Parallel creation time improvements with COFS (1024 files per node)

Figure 12 shows the average time for *utime* requests. We can see that, in this case, the overhead introduced by COFS virtualization is noticeable for a small number of nodes, but it converges with pure GPFS results at 16 nodes. Beyond that point, we can observe that GPFS rapidly degrades its performance as we increase the number of participating nodes; on the contrary, COFS is able to compensate and eliminates the degradation, allowing the system to scale better to larger number of nodes.

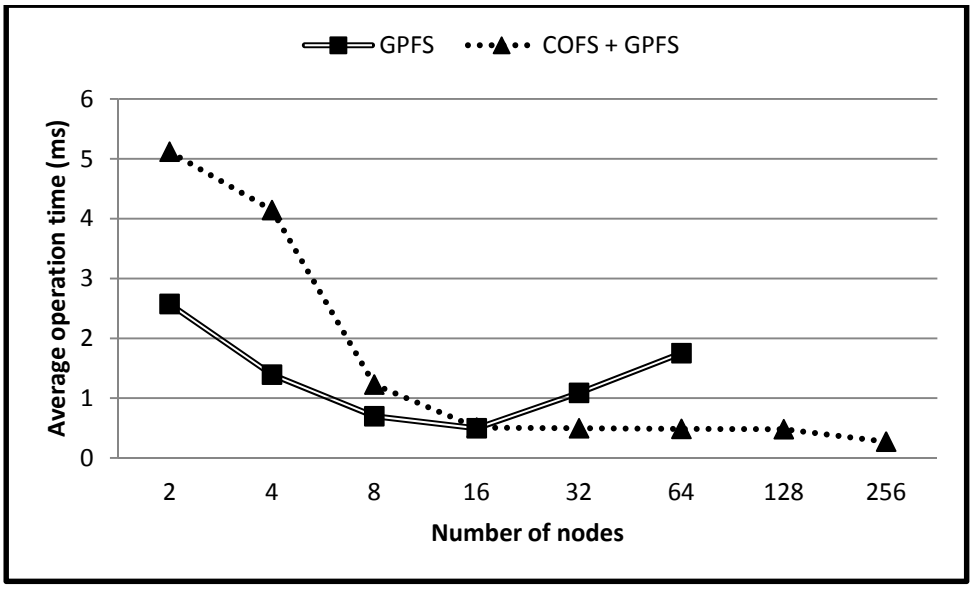


Figure 12 *Utime* request scalability (1024 files per node)

COFS takes advantage of the reduced operation time when GPFS delegation is active by redistributing the files in smaller directories and avoiding parallel accesses to such directories when possible, thus reducing the conflicts between different nodes and allowing GPFS to fully exploit the parallelism.

In summary, our measurements show that the virtualization of the name space provided by the COFS framework can drastically boost the base GPFS file system for file creations on shared parallel environments (with speed-up factors up to 10, as shown in Figure 11. For the rest of metadata operations, performance is also boosted for large numbers of nodes, and performance degradation due to conflicting parallel accesses is reduced.

6.1.2. 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.

We have 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.

Table 1 summarizes the results obtained with the *IOR* I/O benchmark in a small GPFS cluster. Overall, COFS over GPFS is usually able to obtain a data transfer performance similar to

native GPFS. The only remarkable exceptions occur when each node access independent small files.

Access 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 performance improvements of COFS over GPFS as the number of nodes is increased.	COFS performance drawback for single node and comparable performance for multiple nodes.
Random write	COFS performance comparable to GPFS except for small files (<32MB per node) where COFS suffers from slight slowdown.	COFS performance comparable to GPFS.

Table 1 Impact of COFS on data transfers

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 time 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 these 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 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 the data transfer rates. The isolated 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 incorporating the same aggressive caching and delegation techniques for strictly local accesses to the COFS framework.

6.2. Random Slicing

In order to evaluate Random Slicing we used a simulation. To establish a relevant baseline to compare against, we also simulate some of the best-known randomized data distribution strategies such as *Consistent Hashing*[8], *Redundant Share* [10], and *RUSH-R* [11].

We distinguish between homogeneous and heterogeneous settings and also between static and dynamic environments. We assume that each storage system in the homogeneous, static setting can hold up to $k \cdot 500{,}000$ data items, where k is the number of copies of each block. Assuming a hard disk capacity of 1TByte and putting 16 hard disks in each shelf means that each data item has a size of 2MByte. The number of placed data items is $k \cdot 250{,}000$ times the number of storage systems. In all cases, we compare the fairness, the memory consumption, as well as the performance of the different strategies for a different number of storage systems.

The heterogeneous setting assumes that in the beginning we have 128 storage systems and we add 128 new devices each step, which have 3/2 times the size of the previously added system. We are placing again half the number of items, which saturates all disks.

For each of the homogeneous and heterogeneous tests, we also count the number of data items, which have to be moved in case we are adding disks, so that the data distribution delivers the correct location for a data item after the redistribution phase. The number of moved items has to be as small as possible to support dynamic environments, as the systems typically tend to a slower performance during the reconfiguration process.

The dynamic behaviour can be different if the order of the k copies is important, e.g. in case of parity RAID, Reed-Solomon codes, or EvenOdd-Codes, or if this order can be neglected in case of pure replication strategies.

The following sections evaluate the impact of the different distribution strategies on the data distribution quality, the memory consumption of the different strategies, their adaptability and performance. All graphs presented in the section contain four bars for each number of storage systems, which represent the experimental results for one, two, four, and eight copies (please see Figure 13 for the colour codes in the legend). The white boxes in each bar represent the range of results, e.g., between the minimum and the maximum usage. Also, the white boxes include the standard deviation for the experiments. Small or non-existing white boxes indicate a very small deviation between the different experiments.

6.2.1. Fairness

The first simulations evaluate the fairness of the strategies for different sets of homogeneous disks, ranging from 8 storage systems up to 8192 storage systems (see Figure 13). A data distribution is considered to be *fair* when every single storage system gets a share of the data load which is proportional to its relative capacity with respect to the total aggregate capacity of participating storage systems. The following graphs are normalized so that a *usage* of 1 indicates that the distribution is fair.

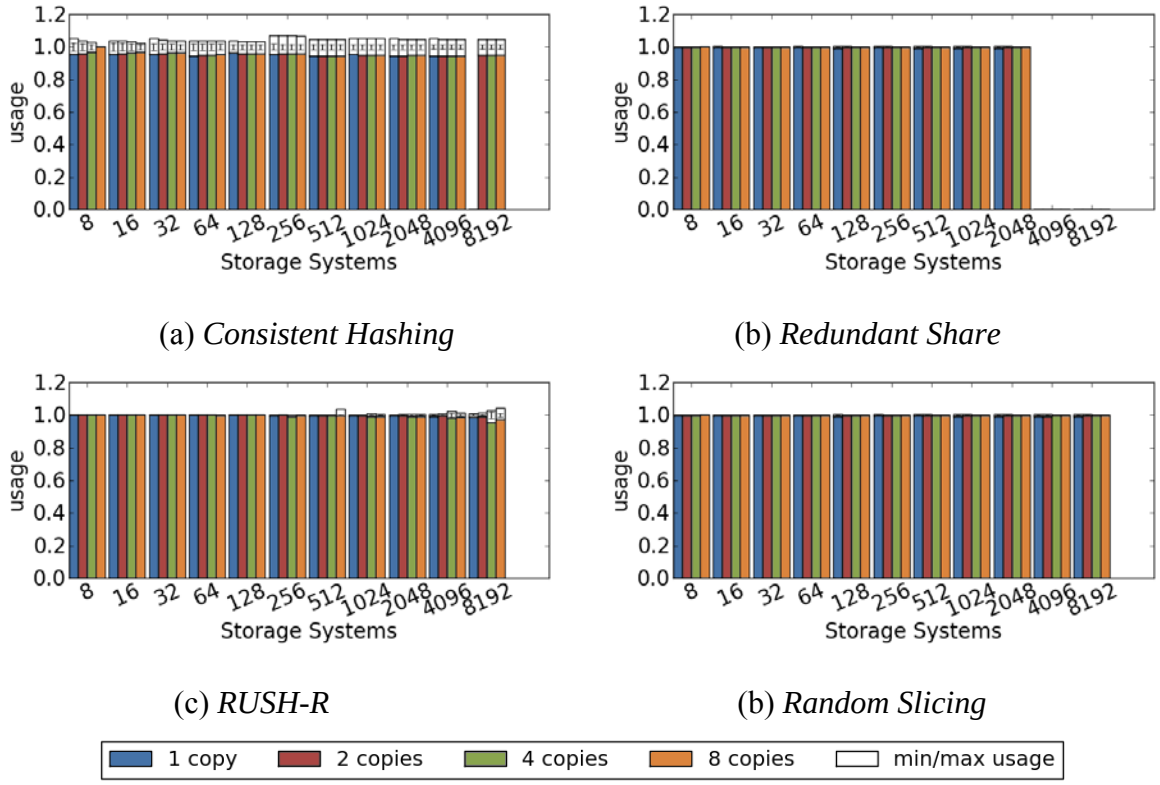


Figure 13 Fairness in an homogeneous setting

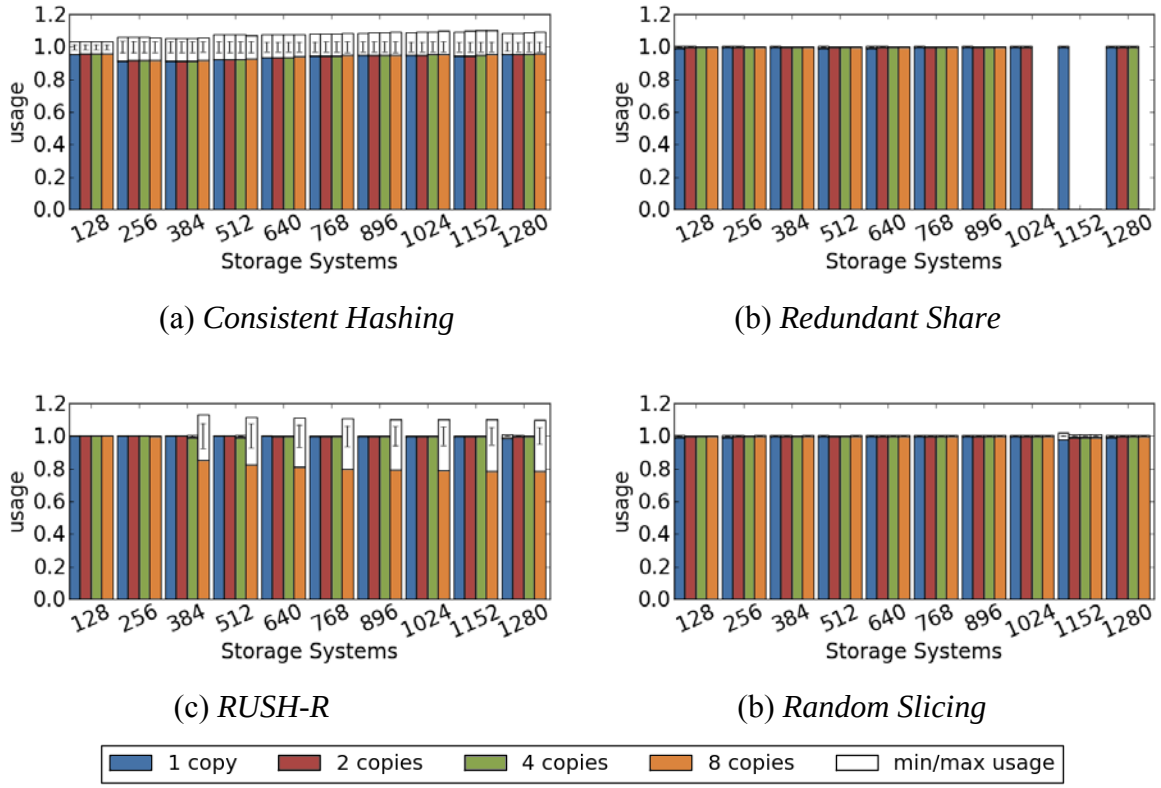
Consistent Hashing has been developed to evenly distribute one copy over a set of homogeneous disks of the same size. Figure 13 (a) shows that the strategy is able to fulfil these demands for the test case, in which all disks have the same size. The difference between the maximum and the average usage is always below 7% and the difference between the minimum and average usage is always below 6%. The deviation is nearly independent from the number of copies as well as from the number of disks in the system, so that the strategy can be reasonably well applied.

Redundant Share uses pre-computed intervals for each disk and therefore does not rely too much on randomization properties. The intervals exactly represent the share of each disk on the total disk capacity, leading to a very even distribution of the data items (see Figure 13 (b)). The drawback of this version of *Redundant Share* is that it has linear runtime, possibly leading to high delays in case of huge environments.

RUSH-R places objects almost ideally according to the appropriate weights, though it begins to degrade as the number of disks grows (see Figure 13 (c)). We believe this happens due to small variations in the probabilistic distribution, which build up for higher numbers of storage systems.

In *Random Slicing*, pre-computed partitions are used to represent a disk's share of the total system capacity, in a similar way to *Redundant Share*'s use of intervals. This property, in addition to the hash function used, enforces an almost optimal distribution of the data items, as shown in Figure 13 (d).

The fairness of the different strategies for a set of heterogeneous storage systems is depicted in Figure 14. As previously described, we start with 128 storage systems and add every time 128 additional devices with 3/2-times the capacity of the previously added.



(a) *Consistent Hashing* (b) *Redundant Share*

(c) *RUSH-R* (b) *Random Slicing*

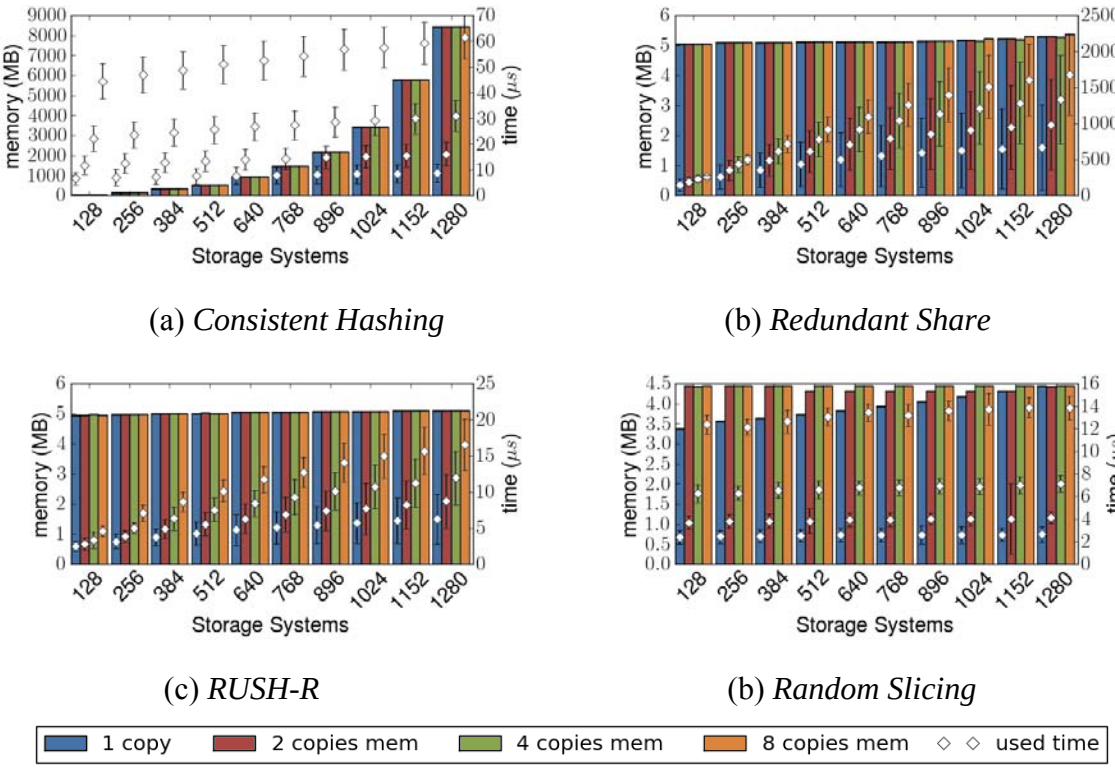
Figure 14 Fairness in an heterogeneous setting

The fairness of *Consistent Hashing* shows apparent deviations from the ideal load (see Figure 14 (a)). The difference between the maximum, respectively minimum and the average usage is around 10% and increases slightly with the number of copies.

Both *Redundant Share* and *Random Slicing* show again a nearly perfect distribution of data items over the storage systems, due to their precise modelling of disk capacities and the uniformity of the distribution functions (see Figure 14 (b) and Figure 14 (d), respectively). *RUSH-R*, on the other hand, does a good distribution job for 1, 2, and 4 copies but seems to degrade with 8 copies showing important deviations from the optimal distribution (Figure 14 (c)).

6.2.2. Memory Consumption and Computation Time

The memory consumption, as well as the performance of the different data distribution strategies, has a strong impact on the applicability of the different strategies. The bars in the graphs of Figure 15 represent the average allocated memory, the white bars on top the peak consumption of virtual memory over the different tests. The points in that figure represent the average time required for a single request. These latencies include confidence intervals.



(a) *Consistent Hashing* (b) *Redundant Share*

(c) *RUSH-R* (b) *Random Slicing*

Figure 15 Memory consumption and performance in an heterogeneous setting

The memory consumption of *Consistent Hashing* only depends on the number and kind of disks in the system, while the number of copies *k* has no influence on it (see Figure 15 (a)). The time to calculate the location of a data item only depends on the number of copies, as *Consistent Hashing* is implemented as a O(1)-strategy for a single copy.

Redundant Share (Figure 15 (b)) has very good properties concerning memory usage, but the computation time grows linearly in the number of storage systems. Even the calculation of a single item for 128 storage systems takes 145 µs. Using 8 copies increases the average access time for all copies to 258 µs, which is 50 µs for each copy, making it unsuitable for large-scale environments.

RUSH-R shows good results both in memory consumption and in computation time (see Figure 15 (c)). The reduced memory consumption is explained because the strategy does not need a great deal of in-memory structures in order to maintain the information about clusters and storage nodes. Lookup times depend only on the number of clusters in the system, which can be kept comparatively small for large systems.

Random Slicing shows very good behaviour concerning memory consumption and computation time, as both depend only on the number of intervals *I* currently managed by the algorithm (see Figure 15 (d)). In order to compute the position of a data item , the strategy only needs to locate the interval containing , which can be done O(log *I*) using and appropriate interval tree structure. Furthermore, the algorithm strives to reduce the number of intervals created in each step in order to minimize memory consumption as much as possible. In practice, this yields an average access time of 5µs for a single data item and 13µs for 8 copies, while keeping a memory footprint similar to that of *Redundant Share*.

6.2.3. Adaptability

Adaptability to changing environments is an important requirement for data distribution strategies and one of the main drawbacks of standard RAID approaches. Adding a single disk to a RAID system typically requires either the replacement of all data items in the system or splitting the RAID environment into multiple independent domains.

The theory behind randomized data distribution strategies claims that these strategies are able to compete with a best possible strategy in an adaptive setting. This means that the number of data movements to keep the properties of the strategy after a storage system has been inserted or deleted can be bounded against the best possible strategy. We assume in the following that a best possible algorithm just moves as much data from old disks to new disks, respectively from removed disks to remaining disks, as necessary to have the same usage on all storage systems. All bars in Figure 16 have been normalized to this definition of an optimal algorithm.

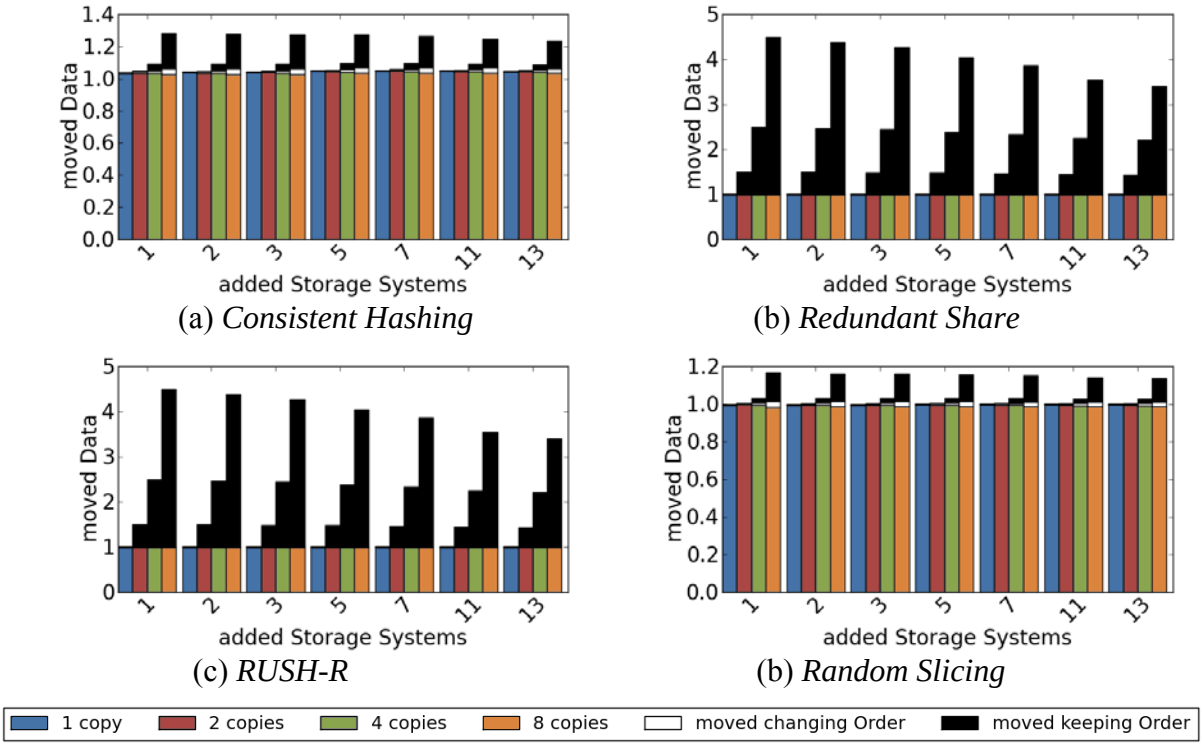


Figure 16 Adaptability in an heterogeneous setting

Furthermore, we distinguish between placements, where the ordering of the data items is relevant and where it is not. The first case occurs, e.g., for standard parity codes, where each data item has a different meaning: if a client accesses the third block of a parity set, then it is necessary to receive exactly that block. In contrast, the second case occurs for RAID 1 sets, where each copy has the same content and receiving any of these blocks is sufficient. The first situation is labelled “moved keeping order” in Figure 16, whereas the second is labelled “moved changing order”. We will see in the following that not having to keep the order strongly simplifies the rebalancing process.

We start our tests in all cases with 128 storage systems and increase the number of storage systems by 1, 2, 3, 5, 7, 11, or 13 storage systems. The new storage systems have 1.5-times the capacity of the original system.

Figure 16 (a) shows the adaptability of *Consistent Hashing* in case that the number of points is fixed for each individual storage system and only depends on its own capacity. We use

2,400 points for the smallest storage system and use a proportional higher number of points for bigger storage systems. In this case the insertion of new storage systems only leads to data movements from old systems to the new ones and not between old ones and therefore the adaptability is very good in all cases.

The adaptability of *Redundant Share* for adding new storage systems is nearly optimal, which is in line with the proofs presented in [12]. Nevertheless, *Redundant Share* is only able to achieve an optimal competitiveness if a new storage system is inserted that is at least as big as the previous ones. Otherwise it can happen that *Redundant Share* is only log n-competitive (see Figure 16 (b)).

Figure 16 (c) shows that *RUSH-R* performs nearly optimal when storage nodes are added. Note, however, that we did not evaluate the effect on replica ordering because the current implementation does not support replicas as distinct entities. Instead, *RUSH-R* distributes all replicas within one cluster.

Figure 16 (d) shows that the adaptability of *Random Slicing* is very good in all cases. This is explained because intervals for new storage systems are always created from fragments of old intervals, thus forcing data items to migrate only to new storage systems.

6.3. Multi-Zone Self-Caching Data Storage

The methodology we followed to evaluate Multi-Zone Self-Caching data storage strategy is analytical. We have created a trace-fed simulator that models large-scale storage architectures and we used it to evaluate our prototype strategy. Individual disks are simulated using the well-know *DiskSim* [13] simulator. In order to see if our strategy offers any significant improvement we also evaluated several well-known allocation strategies to establish a relevant baseline to compare against.

We have evaluated the new strategy against the following baseline policies:

- RAID0: RAID0 divides data in stripes that are distributed in a round-robin fashion across all devices. With a carefully chosen stripe size, this strategy provides extremely good results regarding response time (as requests are split and served in parallel by devices) and load balance.
- SEQUENTIAL: SEQUENTIAL places data sequentially in a device. When the device is full, sequential chooses the next device and proceeds to fill it sequentially, as well. This strategy offers very poor results regarding performance (a request can only be served by one or two devices at most) and also poor load balance (data fills up the disks one at a time, leaving the others unused).

By now, our mechanism supports the following combinations of policies for the caching zone and the archival zone:

- RAID0+SEQUENTIAL: This variant uses a sequential strategy for the archival zone and a RAID0 strategy for the caching zone. We want to evaluate how much of a benefit the caching zone can provide when the archival zone uses an unsuitable policy.
- RAID0+RAID0: This variant uses RAID0 both for the caching zone and the archival zone. We want to see if there is any improvement when comparing it with a traditional RAID0 approach.

For our simulations, we used the *CELLO99*, *DEASNA* [14], *HOME02* [14], *WEB-USERS* and *WEB-RESEARCH* [15]. For each experiment we configured the simulator with 50 disks with a capacity of 146GBytes and speed of 7,200 rpm. The allocation policy uses 0.02% of each disk to create the caching zone, because this value is not sufficient to contain the active working

set and will produce extra disk traffic due to data evictions. We decided to evaluate the behaviour of the cache with a simple LRU replacement policy. All RAID0 instances used a stripe size of 128KBytes and the SEQUENTIAL policy used the same value as block size.

6.3.1. Response time

Figure 17 shows the average response time for read requests for the traces simulated with the respective strategies. As expected, requests are significantly slower in sequential than in RAID0, with response times in general between one and two orders of magnitude slower. Notice that in most cases, the performance of RAID0+SEQUENTIAL is similar to that of RAID0 which validates our hypothesis that there's no need to optimize the data placement of all the data space. In addition, notice that the performance of RAID0+RAID0 shown in Figure 17(a), Figure 17(d) and Figure 17(e) is slightly better than that of RAID0. This can happen because currently active data is clustered in the caching zone which favours spatial locality. Interestingly, the performance gains for *WEB-RESEARCH* and *WEB-USERS* are lower than for others workloads, which might imply that RAID0 is not being exploited to its full potential.

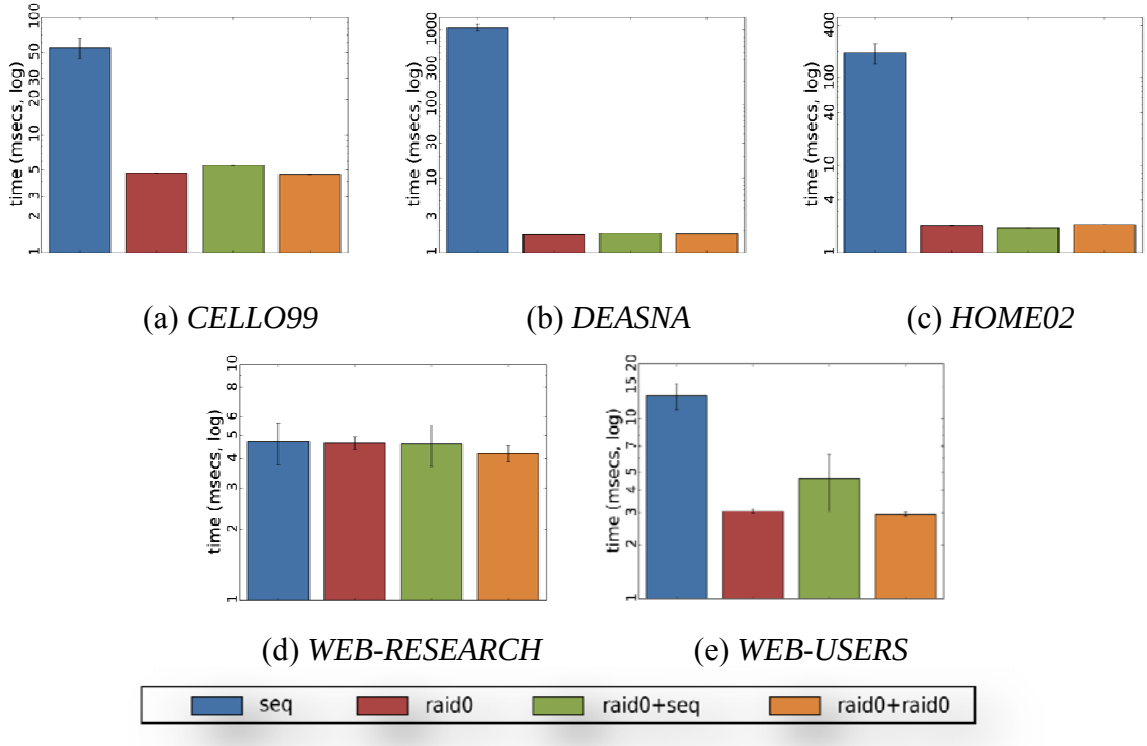


Figure 17 Average response time for read operations

Bar plots in Figure 18 show the average response time for writes requests, however. We can see the same behaviour than in the previous experiment: the response times of the hybrid strategies are similar to those of RAID0 even taking into account the simple LRU replacement policy and the extra overhead added by replacing data from the caching zone.

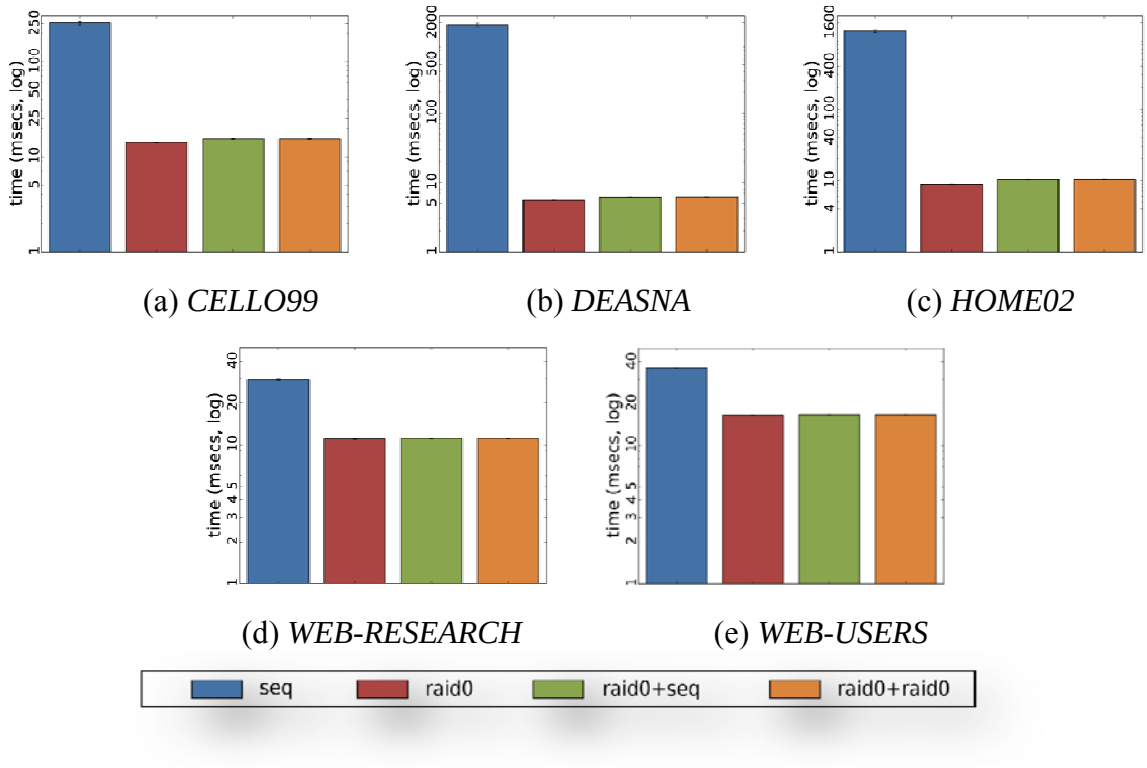


Figure 18 Average response time for write operations

6.3.2. Load Balance

Figure 19 and Figure 20 show the cumulative distribution function (CDF) over time of the standard deviation from an ideal load balance displayed by the devices. Every minute of simulation, we compute the I/O load of each device and how it deviates from an ideal load distribution. Therefore, an ideal load balance corresponds to a value of 1, whereas a value of 2 means that the distribution was twice as unbalanced as the ideal load. Since the experiments use 50 disks, an unbalance index of sqrt(50) ≈ 7.07 is the maximum load unbalance possible, and can only happen when all data access is directed towards only one device. The y axis represents the accumulated frequency (in measurement intervals) where a balancing index was observed, e.g.: Figure 19 (a) shows that, for RAID0, 90% of observations displayed a balancing index of 1.8 or less.

As expected, sequential is highly unbalanced both for read and write operations in all the simulations. RAID0 is significantly more balanced in all simulations except for the *WEB-RESEARCH* and *WEB-USERS* workloads (Figure 19 (d), Figure 19 (e), Figure 20 (d) and Figure 20 (e)). Since these workloads are from web servers, the distribution of requests probably is more random than those coming from human users, which might explain this variation. Furthermore, a stripe size of 128KBytes might be too large for this data workload, which would render the parallelism and load balance provided by RAID0 useless. This question deserves a careful examination and we plan to examine it in the future. Nevertheless, the hybrid strategy RAID0+SEQUENTIAL is successful in reducing the I/O unbalance of SEQUENTIAL, displaying results similar to those of RAID0, both for reads and for writes. This supports our assumption that the caching zone is able to absorb most of the requests and can distribute them effectively across all devices.

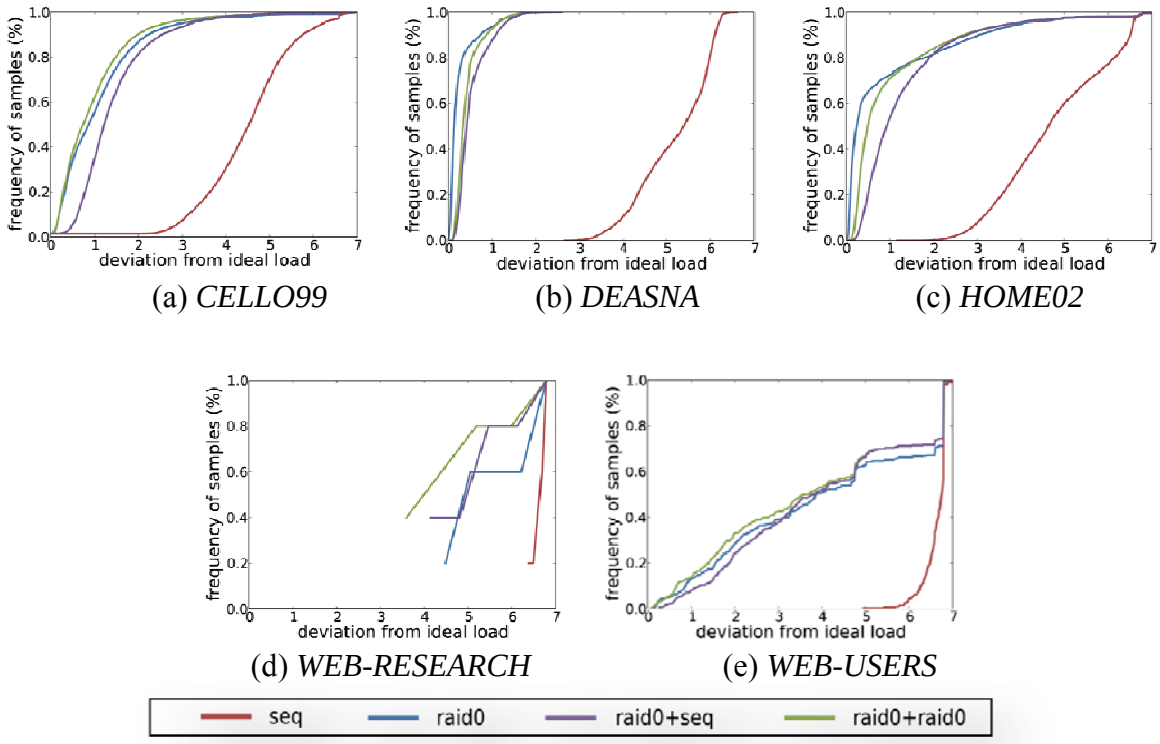


Figure 19 Deviation from ideal load for read operations

Most interestingly, RAID0+RAID0 provides a better load balance than RAID0 in some cases. The set of active data we're considering is significantly smaller than the entire data space (≈ 0.02%) and most requests are directed to it which probably makes it easier to effectively balance it.

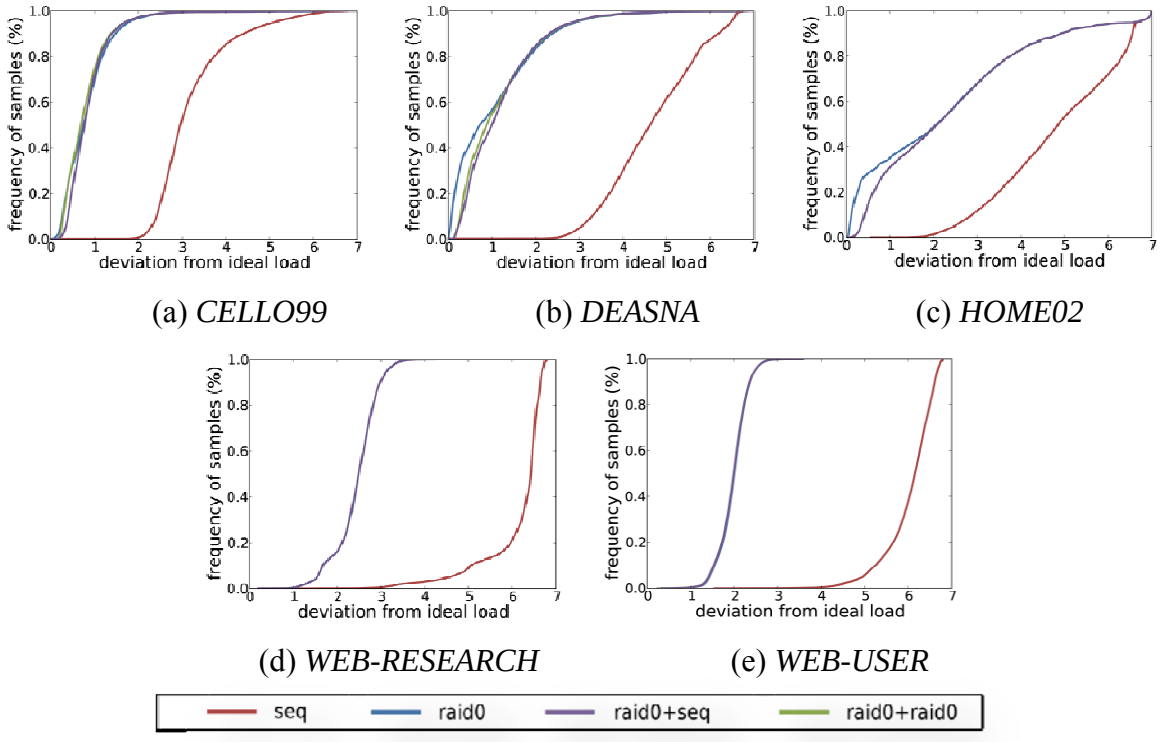


Figure 20 Deviation from ideal load for write operations

7. Conclusions

This document presents evaluations to detect problems in both metadata and data management in large-scale storage systems.

We have evaluated the effect the number of files and clients have on the performance of metadata operations in both Lustre and GPFS. In this evaluation we have detected that both the parameters have a significant negative effect in the obtained performance and we have proposed to use COFS a middleware to handle metadata. COFS is a middleware that decouples the view the user has from the one implemented by the file system and thus is able to transparently convert not optimized cases from the parallel file system point of view into optimized ones, thus getting all the benefits without asking the user to change their behaviours.

The information provided by the metadata performance measurements will help to tune the large scale PRACE systems (and high performance systems in general), as well as algorithms and specific applications, by showing which situations are likely to produce performance issues when accessing the storage systems and avoiding them. To this end, the current COFS implementation is available and can be used to mitigate certain metadata performance issues. The flexible architecture of COFS allows using it both as a transparent layer covering a whole file system, and as a tool to improve the behaviour of individual applications, without affecting the rest of the system.

We have also proposed some data distribution policies that are able to dynamically adapt to the increasing number of storage devices and still obtain the desired performance benefits guaranteeing minimal data movement among devices. These policies are based on a data distribution where blocks are placed by applying a function that is half randomized and half deterministic, and by increasing the caching hierarchy with one level built from a tiny portion of all available disks. The new data distribution policies will be used to enable the capacity upgrade of future large scale storage systems at a much lower cost than current algorithms.

We expect to take more measurements in the PRACE Lustre prototype and other large scale systems; any new developments and information acquired will be documented in a Progress Report at month 18.