Efficient Implementation of Data Objects in the OSD+-based Fusion Parallel File System

Juan Piernas, Pilar González-Férez

Computer Architecture and Parallel Systems
University of Murcia

5th Workshop on Runtime and Operating Systems
for the Many-core Era (ROME 2017)
Santiago de Compostela, August 28th, 2017

Outline

Introduction

- File systems for HPC environments provide clusters of data servers for:
    - High rates in read and write operations
    - Fault tolerance
    - Scalability, etc.
- Recently, they have also added support for clusters of metadata servers for:
    - Managing billions of files
    - Dealing with huge directories
    - Fault tolerance
    - Scalability, etc.
- Usually, separate clusters from a conceptual point of view
    - Although a data server and a metadata server can share a computer

Introduction: FPFS

- In FPFS, however, there exists a single cluster, made of OSD+ devices that handle both data and metadata operations
- OSD+s have proved a great performance in metadata operations. Thanks to them:
  - FPFS is able to create, stat and delete hundreds of thousands of files per second with a few servers
  - FPFS increases its throughput even further by means of distributed directories and batch operations
- In this work, we describe how we have implemented the support for data objects
- But, more importantly, we will show that the utilization of a unified data and metadata server (i.e., an OSD+ device) provides FPFS with a competitive advantage that allows it to speed up some file operations

Outline

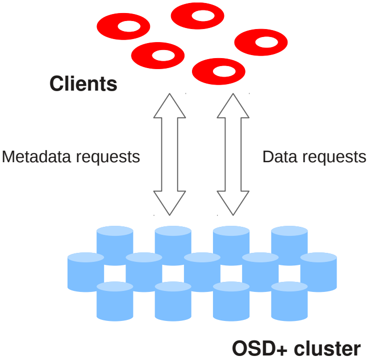
Fusion Parallel File System

FPFS

A parallel file system based on OSD+ devices. These devices:

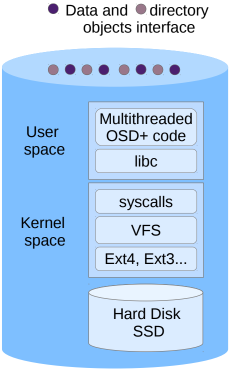
- Provide a single kind of servers
- Manage both data and metadata operations

- Advantages:
  - Simple architecture
  - Metadata cluster as large as the data cluster
  - Better use of resources
  - Increased scalability



OSD+ devices

- Enhanced OSD devices
- Support directory objects and related operations: `creat`, `mkdir`, `rmdir`, ...
  - Also support for data objects
- Implemented as multi-threaded user-space processes in Linux
- Regular file system used as storage backend
  - Should be POSIX-compliant
  - Must support extended attributes



Directory objects

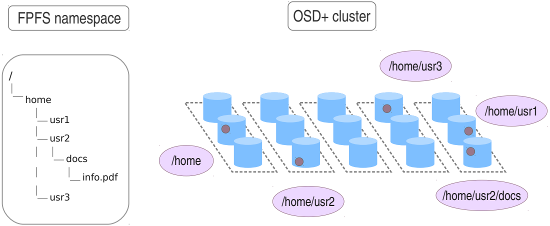
- Implemented as regular directories in the local file systems of the OSD+s:
  - Any directory-object operation is directly translated to a regular directory operation
  - Two important advantages:
    - Implementation simpler and overhead smaller
    - For metadata operations involving a single OSD+, POSIX semantics and atomicity guaranteed by the backend file system
  - For operations involving more than one OSD+ (e.g., `mkdir`), atomicity guaranteed by a three-phase commit protocol

Embedded i-nodes

- Directory objects use "embedded i-nodes":
  - Each directory entry stores file name and attributes, including information about data objects
  - Exceptions: size and modification time attributes of a file; stored at its data object(s)
- Internally implemented through i-nodes and extended attributes of empty files created in the directory

Namespace distribution

- FPFS distributes directory objects across its cluster for improved scalability and performance:



- Distribution function:

$$oid = F(hash(dirfullpath))$$

  - F: deterministic pseudo-random function (e.g., CRUSH)
  - oid: ID of the OSD+ storing the directory object with name *dirfullpath*
- Clients directly access directories without performing path resolutions
- CRUSH, lazy techniques, etc., to handle hashing drawbacks

Huge directories

- A directory is distributed among several directory objects in different OSD+ devices when it stores more than a given number of files
  - Threshold can be 0 → Distribution from the very beginning; useful for directories known to be huge
- Subset of OSD+s supporting a huge directory composed of:
  - Routing OSD+: provides clients with hugedir's distribution information
  - Storing OSD+s: store directory's content
- Storing objects work independently, thereby improving performance and scalability

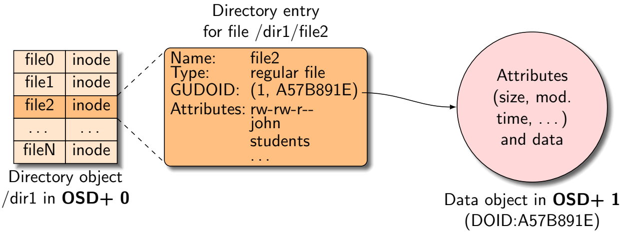
Outline

1. Introduction
2. Overview of FPFS
3. Data objects
4. Experimental Results
5. Conclusions

Data objects

- Storage elements
- Can also have attributes that users can set and get
- Main operations: read and write
- Each has a *data object ID* (DOID), unique inside an OSD+
- A data object globally identifiable by its DOID and the ID of its holding OSD+ device
  - This pair (device ID, data object ID) called a globally unique data object ID (GUDOID)

FPFS regular files and data objects



- An FPFS regular file poses three related elements:
  - A directory entry
  - An i-node
  - A data object
- Directory entry and i-node stored together in the corresponding directory object
- The data object stored separately → Different allocation policies:
  - *Same OSD+* (default) → Reduced network traffic during file creations
  - *Random OSD+* → Potentially more balanced workloads

Implementation of data objects

- Data objects internally implemented in an OSD+ as regular files
- A *data directory* stores those regular files, distributed into subdirectories to improve performance
- An `open()` call on an FPFS regular file returns:
  - A *file descriptor*, to directly operate on its data object(s)
  - A key (*secret*), used along with the file descriptor to guarantee that the client has been granted access to the data object(s)
- Currently supported operations on data objects: `read()`, `write()`, `fstat()`, `lseek64()`, and `fsync()`

Optimizing the implementation

- How can we profit the fact that an OSD+ devices manage both data and metadata?
- When an FPFS regular file is created:
  - An empty file is created in the directory supporting the directory object of the FPFS file
  - This empty file acts as dentry and embedded i-node
  - But, it can also act as data object if the default allocation policy for data objects is active
  - Result: creation is quite fast and also atomic
- Impossible for file systems with separate data and metadata servers → This adds overhead due to:
  - Independent operations in different servers
  - Network traffic generated to perform those operations and guarantee their atomicity

Optimizing the implementation

- The overlap between a dentry-inode and its data object disappears:
    - When a directory object is moved (rename of a directory or distribution of a huge directory)
        - However, new files for an already-distributed huge directory are created again with its three elements (directory entry, i-node and data object) internally backed up by a single file
    - When a file has several data objects
    - For hard links, handled by FPFS through *i-node objects*

Outline

1. Introduction
2. Overview of FPFS
3. Data objects
4. **Experimental Results**
5. Conclusions

System under test (hardware)

Hardware in every node of the cluster

Platform	Supermicro X7DWT-INF
Processor	2 x Intel Xeon E5420 quad-core at 2.50 Ghz
Main memory	4 GB
System disk	HDD Seagate ST3250310NS (250 GB)
Test disk	SSD Intel 520 Series (240GB)
OS	64-bit CentOS 7.2
Interconnect	Gigabit network
Switch	D-Link DGS-1248T

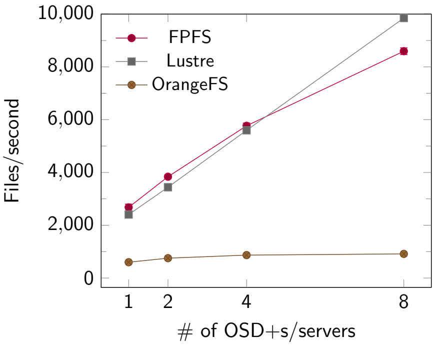
System under test (software)

- FPFS compared against OrangeFS 2.9.6 and Lustre 2.9.0
- Backend file system for FPFS and OrangeFS: Ext4
  - I/O scheduler for SSDs: noop
  - Formatting and mount options of Ext4 properly set to try to obtain maximum throughput when FPFS and OrangeFS are deployed
- We do not change Lustre's default configurations
- Directories shared out among all the available servers

HPCS-IO: benchmark details

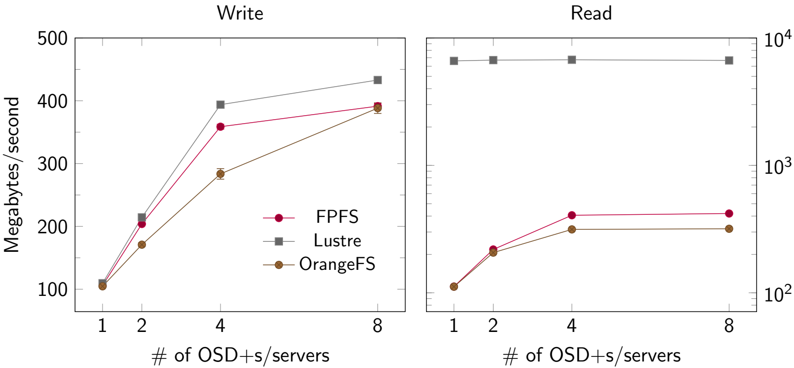
- Version: 1.2.0-rc1
- Scenarios:
  - **Scenario 4**: 64 processes with 10 directories each. Processes create as many files (with sizes between 1 kB and 64 kB) as possible in 50 secs.
  - **Scenario 8**: 128 processes; each one creates a 32 MB file
  - **Scenario 9**: a single process issues `stat()` operations on empty files in a sequential order. There are 256 directories created with 10 000 empty files in each (2 560 000 files altogether)
  - **Scenario 10**: like scenario 9, but `stat()` operations issued by 10 processes
  - **Scenario 11**: like scenario 9, but the process issues `stat()` operations in a random order
  - **Scenario 12**: like scenario 11, but `stat()` operations are issued by 128 processes
- Client processes shared out among four compute nodes
- OrangeFS starts crashing when more that 64/128 processes are used

HPCS-IO: results for scenario 4



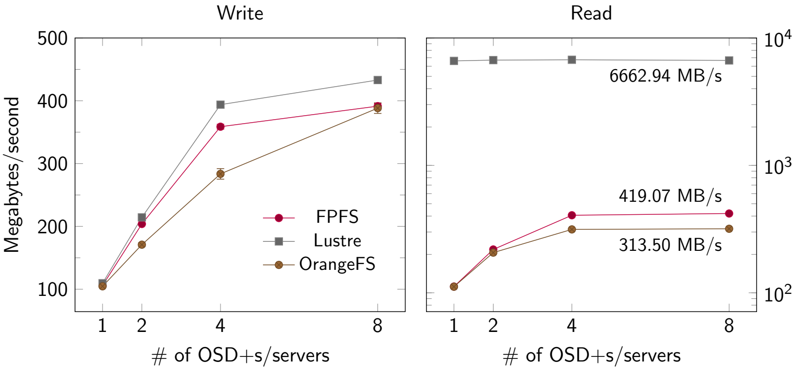
- FPFS competes with Lustre
- FPFS/Lustre around one order of magnitude better than OrangeFS
  - Creation of many small files in this scenario
  - FPFS and Lustre deal with data and metadata operations much better than OrangeFS
- OrangeFS hardly improves its performance by adding servers

HPCS-IO: results for scenario 8



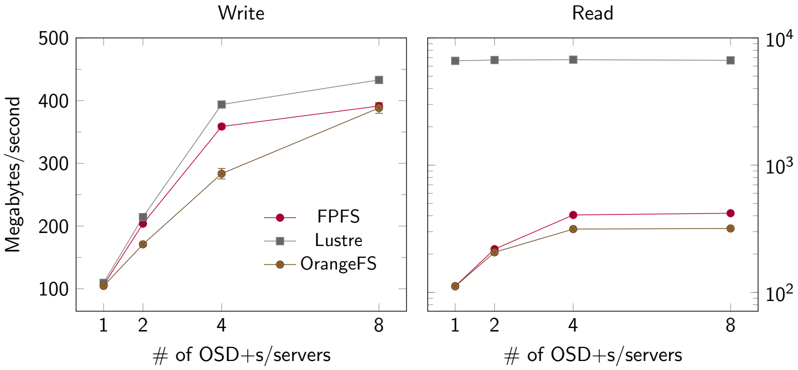
- NICs in the clients saturated with 8 servers → Rates hardly increases beyond 4 servers

HPCS-IO: results for scenario 8



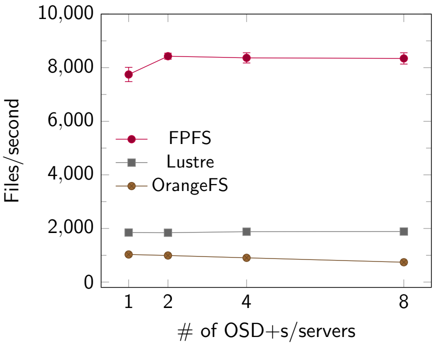
- Results of each file system depend on its implementation and features:
  - Lustre implements a client-side cache → High aggregated read rates
  - Lustre implemented in kernel space and optimized use of the interconnect → Smaller overhead → Higher aggregated write rates

HPCS-IO: results for scenario 8



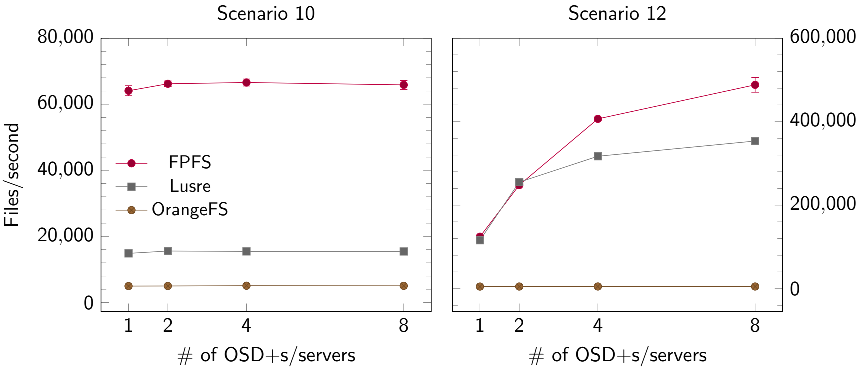
- FPFS and OrangeFS implemented in user space without client-side cache
- However, FPFS provides 23.5% more aggregated bandwidth for writes, and 34% for reads than OrangeFS

HPCS-IO: results for scenario 9



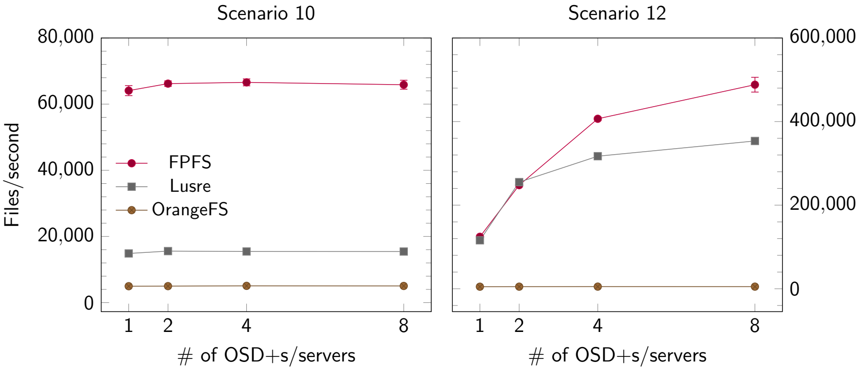
- One order of magnitude more operations/s in FPFS than in OrangeFS
- 4× more operations/s in FPFS than in Lustre
- Steady performance in FPFS/Lustre regardless the number of servers
- OrangeFS's performance slightly decreases with the number of servers

HPCS-IO: results for scenarios 10 and 12



- FPFS's performance is more than 12× better than OrangeFS's in Scenario 10, and more than 95× better in Scenario 12
  - The number of clients is important in FPFS: 10 clients issuing `stat()` operations in Scenario 10, and 128 in Scenario 12
  - OrangeFS does not scale in any case. Its behavior does not change between scenarios either

HPCS-IO: results for scenarios 10 and 12

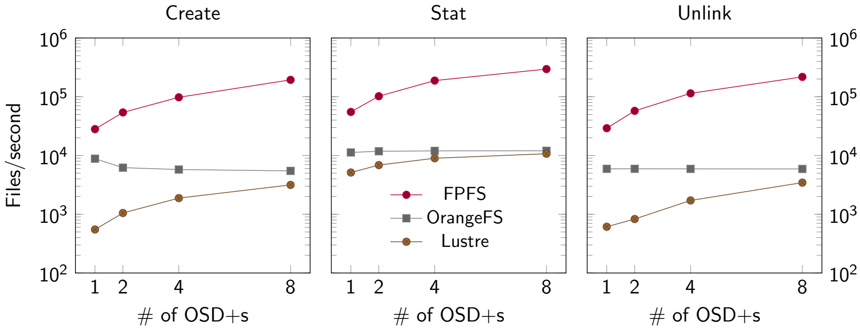


- FPFS's performance is more than 4× better than Lustre's in Scenario 10, and up to 38% in Scenario 12
- Lustre's performance does not improve beyond two servers in Scenario 12
  - An analysis of network traffic reveals that Lustre's "packaging" of requests adds delays that downgrade performance

Single shared directory: benchmark details

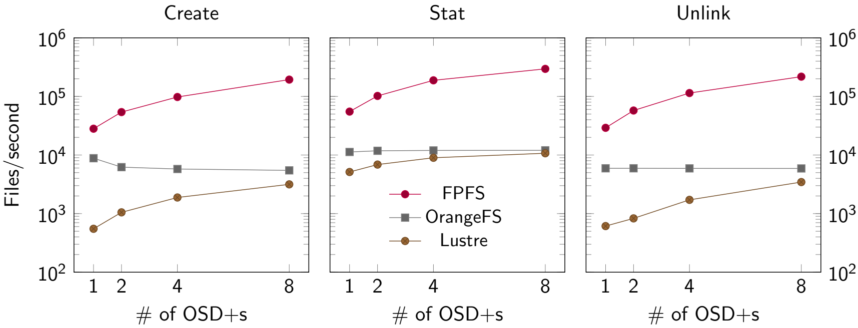
- Synchronization points among client processes placed by HPCS-IO scenarios limit performance
- HPCS-IO scenarios do not operate on a single directory → Benefits of distributing hugedirs are not clear either
- Proposed benchmarks:
  - **Create**: each process creates a subset of empty files in a shared directory (write-only metadata workload)
  - **Stat**: each process gets the status of a subset of files in a shared directory (read-only metadata workload)
  - **Unlink**: each process deletes a subset of files in a shared directory (read-write metadata workload)
- 256 client processes spread across four compute nodes:
  - No synchronization points among clients
- Directory size: $400\,000 \times N$ files
  - $N =$number of servers → We test weak scaling

Single shared directory: results



- Huge throughput of FPFS with respect to Lustre and OrangeFS
  - Note the log scale in the Y-axis!
  - FPFS gets, at least, one order of magnitude more ops/s
  - But FPFS usually much better: up to 70× more ops/s than OrangeFS and 37× more than Lustre in some cases
  - With just 8 OSD+s and a Gigabit interconnect, FPFS able to create, stat, and delete more than 205 000, 298 000 and 221 000 files/s, resp.

Single shared directory: results



- Performance differences between FPFS and the rest due to:
    - Network traffic per file: much higher in Lustre and OrangeFS, and even increases with the number of servers
    - Imbalances: Lustre and OrangeFS have a metadata server that sends/receives much more packets that the other metadata servers
    - Possibly, some serialization problems in Lustre and OrangeFS
    - Consequently, serious scalability problems in Lustre and OrangeFS

Outline

Conclusions

- Presented the design and implementation of data objects in the OSD+ devices
- Optimization of the implementation due to OSD+s' unique features:
  - Some common file operations sped up
  - Optimization impossible in file systems with conceptually-independent data and metadata severs (Lustre, OrangeFS, . . .)
- FPFS's performance much better than Lustre's and OrangeFS's:
  - At least, 10× better for metadata-intensive workloads, but up to 95× better than OrangeFS's and 37× better than Lustre's
  - Up to 34% more aggregated bandwidth than OrangeFS for workloads with large data transfers
  - Competes with Lustre for data writes
- Experimental results show serious scalability problems in Lustre and OrangeFS

Efficient Implementation of Data Objects in the OSD+-based Fusion Parallel File System

Juan Piernas, Pilar González-Férez

Questions?

5th Workshop on Runtime and Operating Systems
for the Many-core Era (ROME 2017)
Santiago de Compostela, August 28th, 2017