SCALABLE COMPUTING SOFTWARE LABORATORY

ILLINOIS INSTITUTE OF TECHNOLOGY

Datacloud'17

Sunday, Nov 12th

Enosis: Bridging the Semantic Gap between File-based and Object-based Data Models

Anthony Kougkas - akougkas@hawk.iit.edu, Hariharan Devarajan, Xian-He Sun

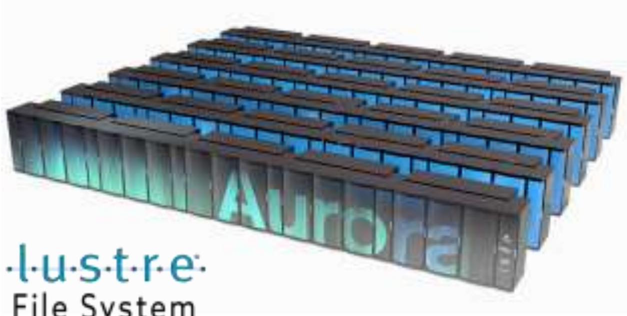
Outline

- Introduction
- Background
- Approach
- Evaluation
- Conclusions
- Q&A

What is this talk about?

- Highlights of this work:
  - Key characteristics of file-based and object-based storage systems.
  - Design and implementation of a unified storage access system that bridges the semantic gap between file-based and object-based storage systems.
  - Evaluation results show that, in addition to providing **programming convenience and efficiency**, our library, Enosis, can grant **higher performance** avoiding costly data movements between file-based and object-based storage systems.

Different communities - different systems



The tools and cultures of HPC and BigData analytics have diverged, to the detriment of both;
unification is essential to address a spectrum of major research domains.

- D. Reed & J. Dongarra

Enosis: Unified Storage Access System

Challenges of storage unification

- Wide range of issues:
  1. There is a gap between
     a) **traditional storage solutions** with semantics-rich data formats and high-level specifications, and
     b) **modern scalable data frameworks** with simple abstractions such as key-value stores and MapReduce.
  2. There is a big difference in architecture of programming models and tools.
  3. Lack of management of
     1. heterogeneous data resources
     2. diverse global namespaces stemming from different data pools.

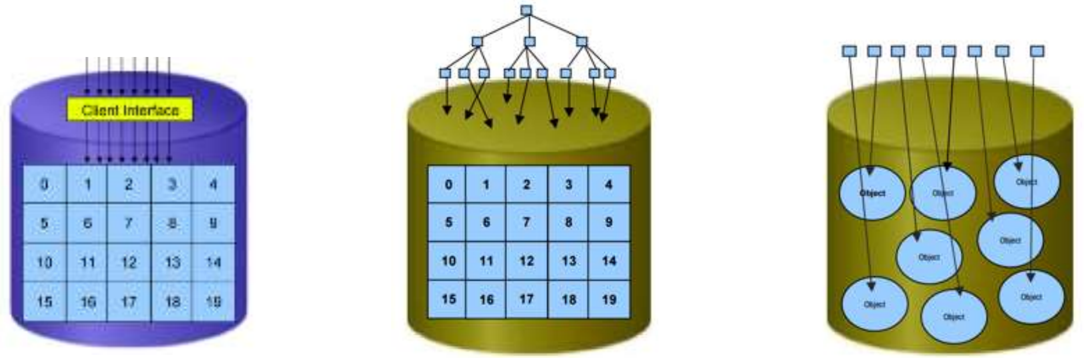
Our thesis

- A radical departure from the existing software stack for both communities is not realistic.
- Future software design and architectures will have to raise the abstraction level and therefore,
  - **bridge the semantic and architectural gaps.**
- We envision
  - a data path agnostic to the underlying data model
  - leverage each storage system's strengths while complementing each other for known limitations.

Anthony Kougkas, akougkas@hawk.iit.edu

Data formats and storage systems

- Data are typically represented as files, blocks, or objects.
- Two major camps of storage solutions:
  - File-based storage systems
    - POSIX-I/O with fwrite(), fread(), MPI-I/O with MPI_File_read(), MPI_File_Write()
    - High-level I/O libraries e.g., HDF5, pNetCDF, MOAB etc
  - Object-based storage systems
    - REST APIs, Amazon S3, OpenStack Swift with get(), put(), delete()

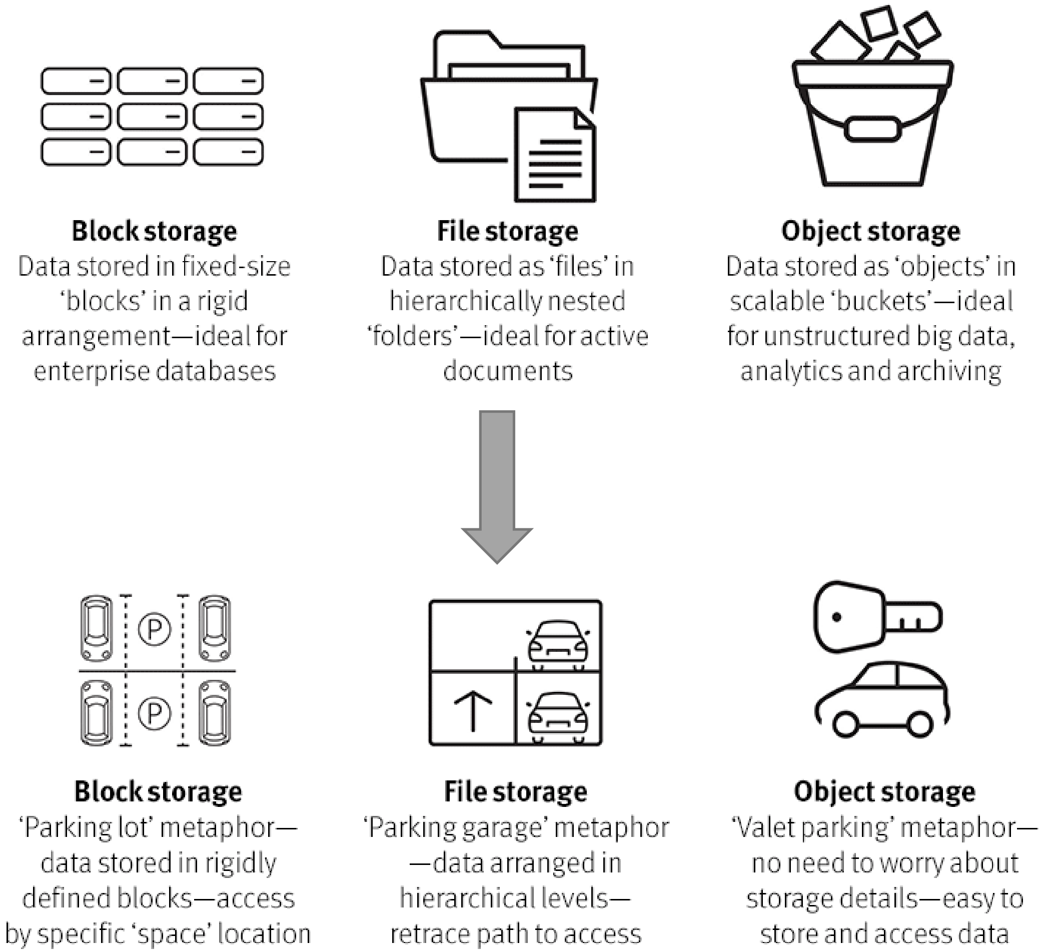


Blocks | Files | Objects

Interface and API

- Storage expectations:
  - MPI and scientific computing
  - Hadoop ecosystem and BigData computing
  - POSIX compliant or not?
  - Structured, semi-structured, and unstructured data
  - Consistency models: strong vs eventual?

Data models differences



Source: Dell EMC

Category	Object Storage	File Storage
Data unit	Objects	Files
Update	Create new object	In-place updates
Protocols	REST and SOAP	NSF with POSIX
Metadata	Custom	Fixed attributes
Strengths	Scalability	Simplified access
Limitations	Frequent updates	Heavy metadata
Performance	High throughput	Streaming of data

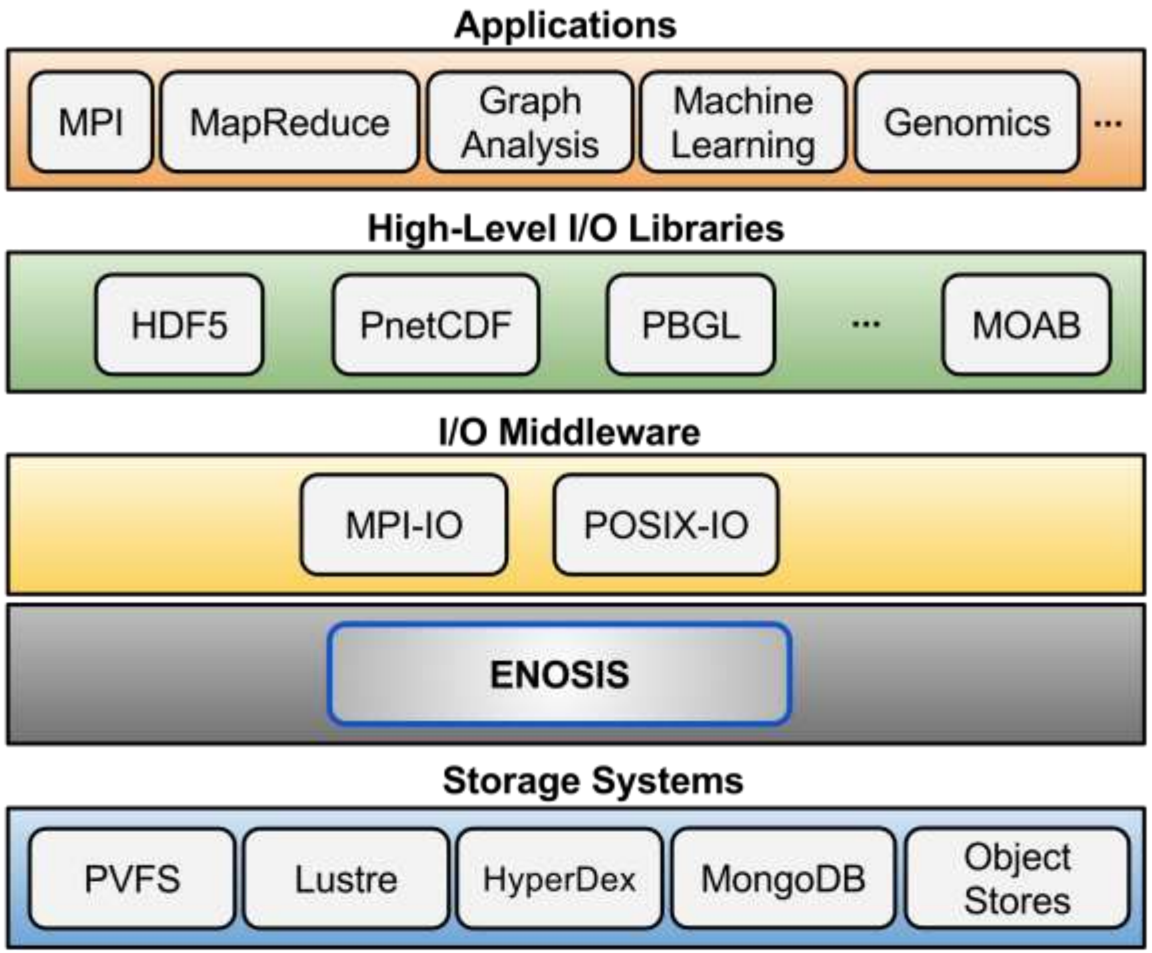
- There is no "one-storage-for-all" solution.
- Each system is great for certain workloads
- Unification is essential

Related work

- From the File system side:
  - CephFS
  - PanasasFS
  - OBFS: A File System for Object-based Storage Devices OSD
- From the Object store side:
  - AWS Storage Gateway
  - Azure Files and Azure Disks
  - Google Cloud Storage FUSE

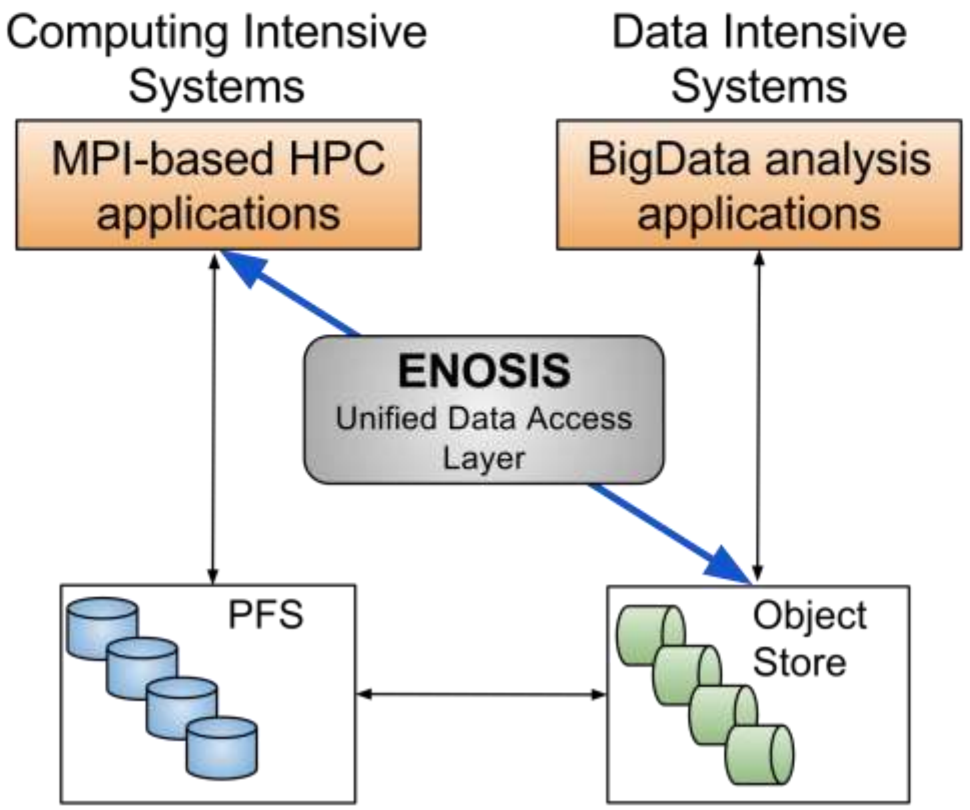
Enosis is a general solution that can bridge any File System with any Object Store and does NOT require change in user code and underlying system deployments.

Design



- Middle-ware library
- Link with applications (i.e., re-compile or LD_PRELOAD)
- Wrap-around I/O calls
- Written in C++, modular design
- Existing datasets loaded upon bootstrapping via crawlers
- Directory operations not supported
- Deletions via invalidation
- Enosis is Not yet-another file system on top of Object Store but a semantics bridge that maintains strong data consistency

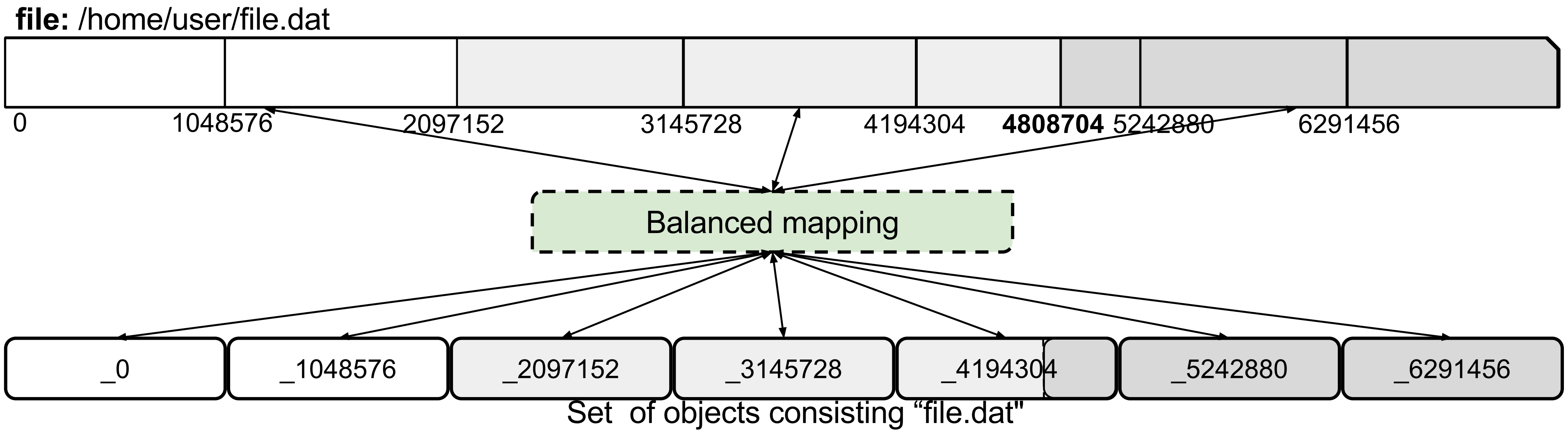
Design



- Three mapping strategies for POSIX files
  1. Balanced
  2. Read-optimized
  3. Write-optimized
- One new HDF5 mapping strategy
- A naïve strategy is when one file is mapped to one object
  - It is used as our baseline reference
  - It is what most connectors do

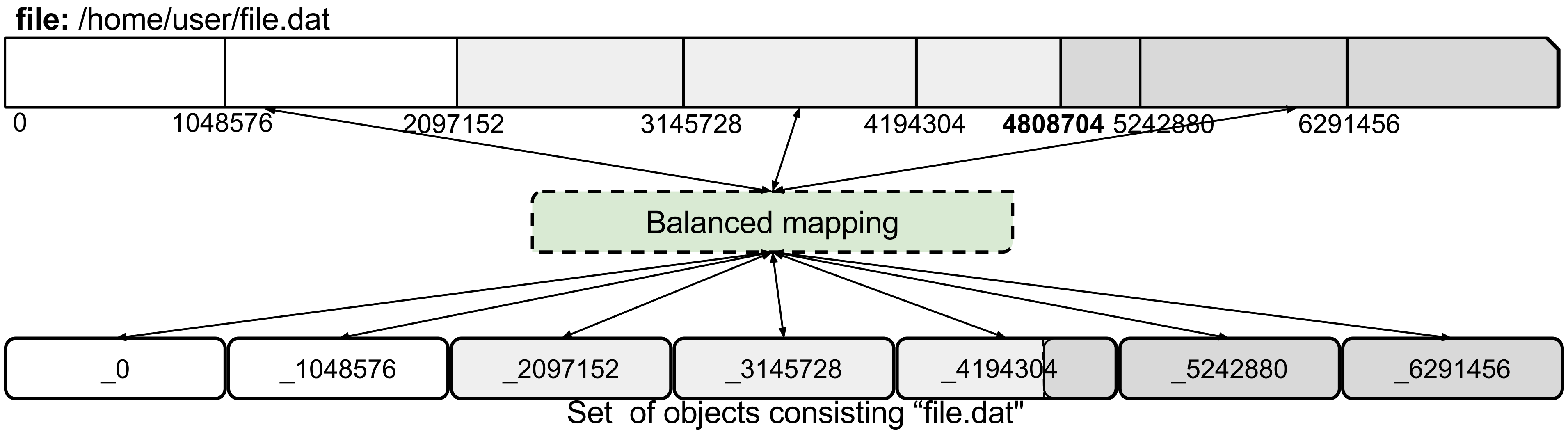
Design - Balanced Mapping

- Ideal for mixed workloads (both `fread()` and `fwrite()`).
- File is divided into predefined (but configurable), fixed-size, smaller units of data, called *buckets*.
- Natural mapping of **buckets-to-objects.**



Design - Balanced Mapping

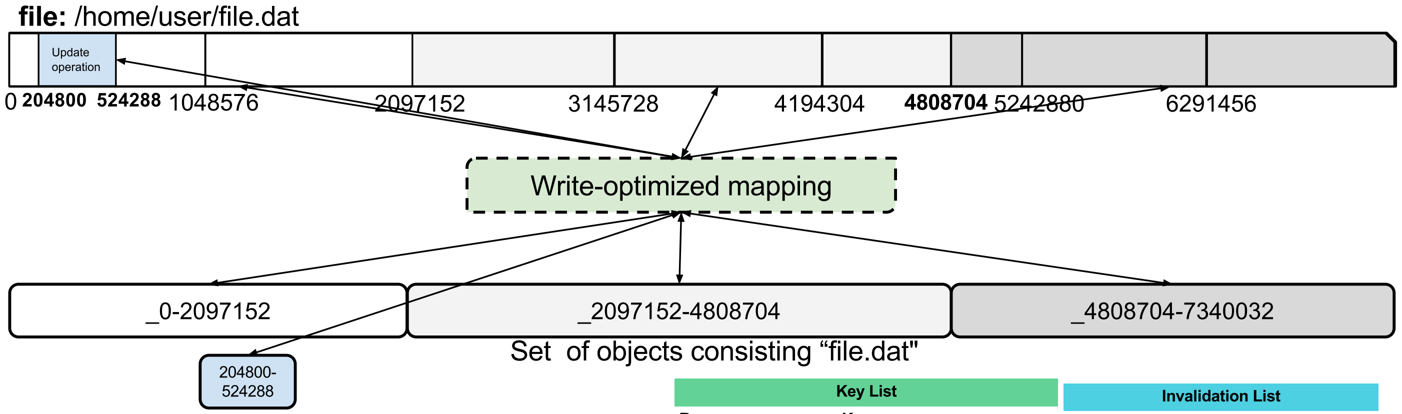
- Bucket size is a tunable parameter and plays a big role in performance.
- After extensive testing, we found that a bucket size equal to the median of all request sizes is the best and more balanced choice.
- Corner buckets and updates might create more reading.



Anthony Kougkas, akougkas@hawk.iit.edu

Design – Write-optimized Mapping

- Ideal for write-only or write-heavy (e.g., >80% write) workloads.
- Each request creates a new object.
- A mapping of offset ranges to available keys is kept in a B+ tree for fast searching.

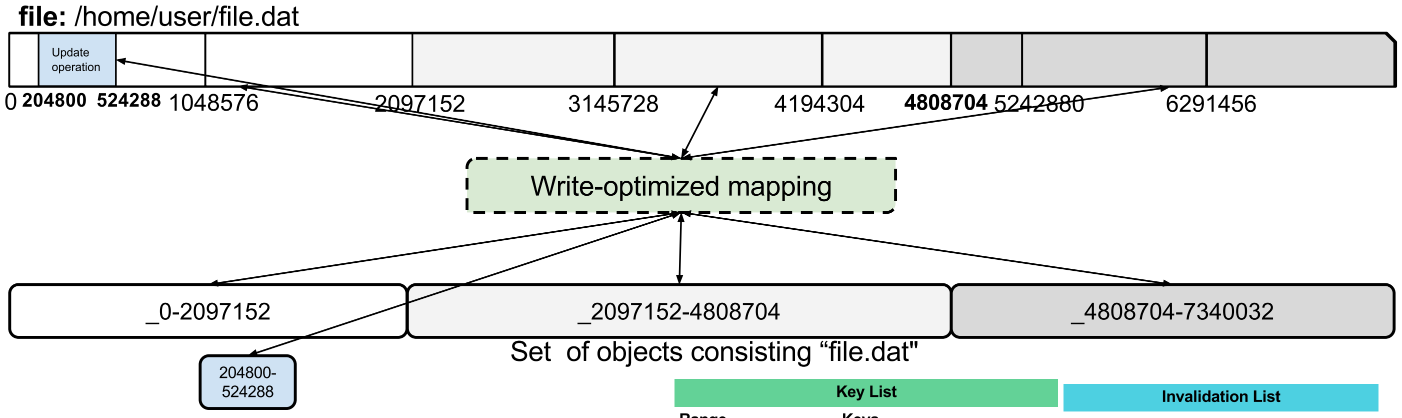


Key List	
Range	Keys
0 - 2097152	file.data_0-2097152
2097152 - 4808704	file.data_2097152-4808704
4808704 - 7340032	file.data_4808704-7340032
0 - 2097152	file.data_204800-524288

Invalidation List	
Key	Range
File.dat_0-2097152	204800 - 524288

Design – Write-optimized Mapping

- Update operations create a new object and invalidate the newly written part of the old object ensuring consistency.
- Results in fast writes in the expense of read operations.
- Any `fread()` first needs to retrieve all keys in the range, fetch the objects, concatenate the data, and finally return to the user.

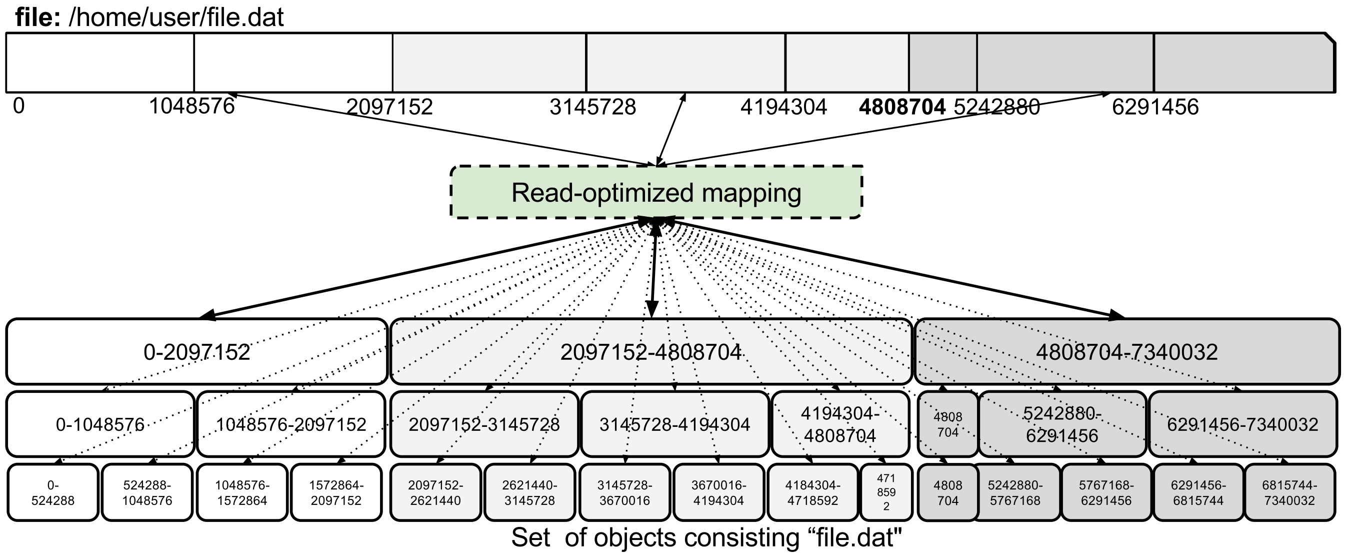


Key List	
Range	Keys
0 - 2097152	file.data_0-2097152
2097152 - 4808704	file.data_2097152-4808704
4808704 - 7340032	file.data_4808704-7340032
0 - 2097152	file.data_204800-524288

Invalidation List	
Key	Range
File.dat_0-2097152	204800 - 524288

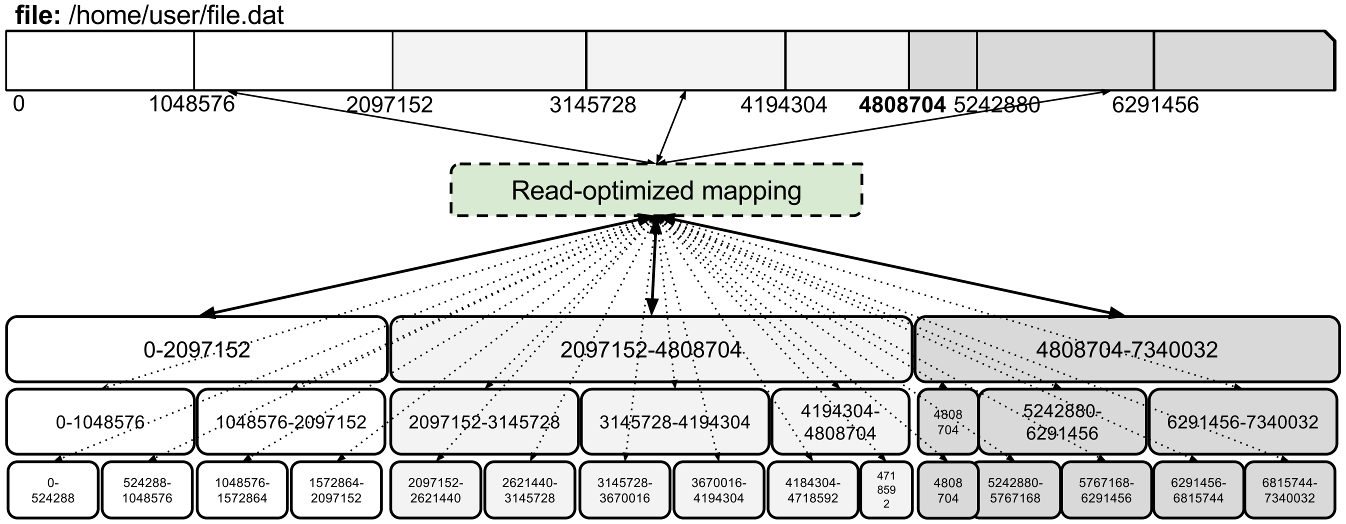
Design – Write-optimized Mapping

- Ideal for read-only or read-heavy (e.g., >90% read) workloads.
- Each write creates a plethora of new various-sized objects.
- Equivalent to concept of replication: sacrifice disk space to increase availability for reads.



Design – Write-optimized Mapping

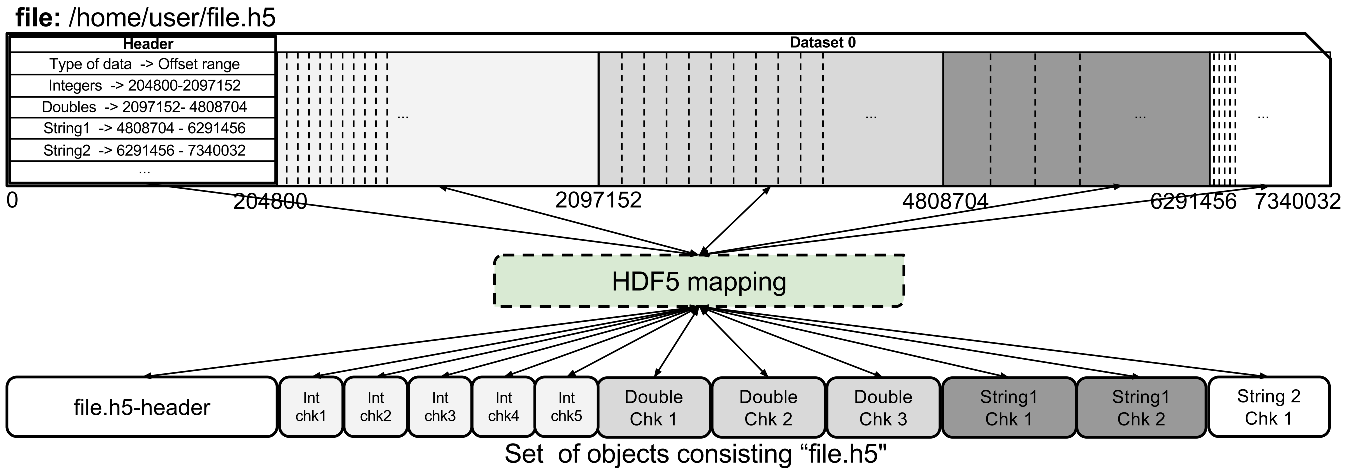
- Tunable granularity of creating objects. Suggested: 512KB
- All available keys in a range of offset are kept in a B+ tree.
- Results in fast reads in the expense of write operations and the extra disk space required.



Set of objects consisting "file.dat"

Design – HDF5 Mapping

- Exploit rich metadata info HDF5 offers (i.e., self-descriptive nature of the format) to create better mappings.
- Each HDF5 file creates 2 types of objects: header object and data object.
- Header object contains metadata information and is kept in memory for fast query. It is persisted upon file close.
- Variable-sized data objects are created based on each dataset's dimensions and datatype. E.g., every 20 integers -> 1 object

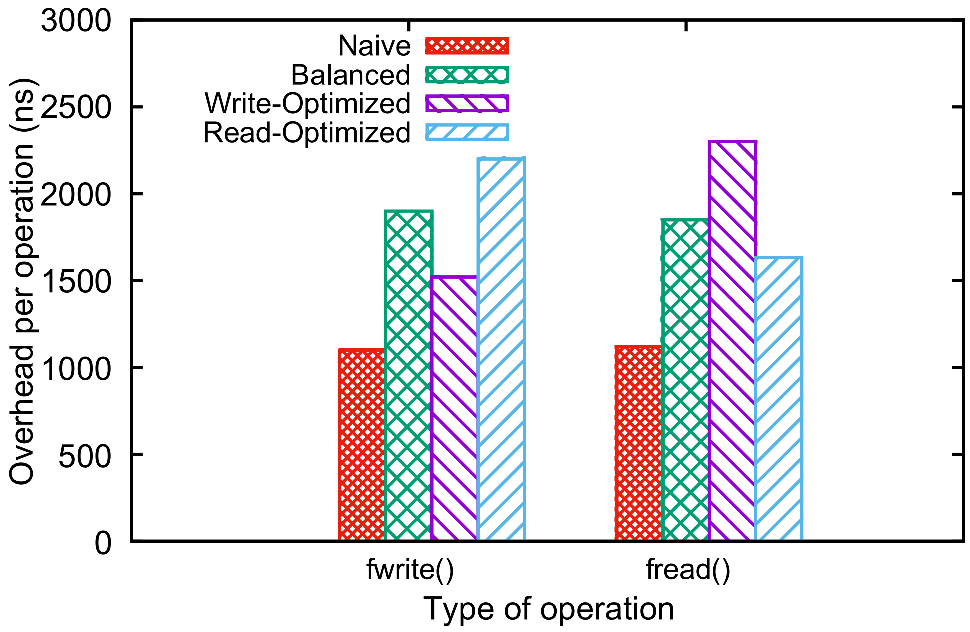


Evaluation Methodology

- Testbed: Chameleon System
- Appliance: Bare Metal
- OS: Centos 7.1
- Storage:
  - OrangeFS 2.9.6
  - MongoDB 3.4.3
- MPI: Mpich 3.2
- Programs:
  - Synthetic benchmark
  - Montage

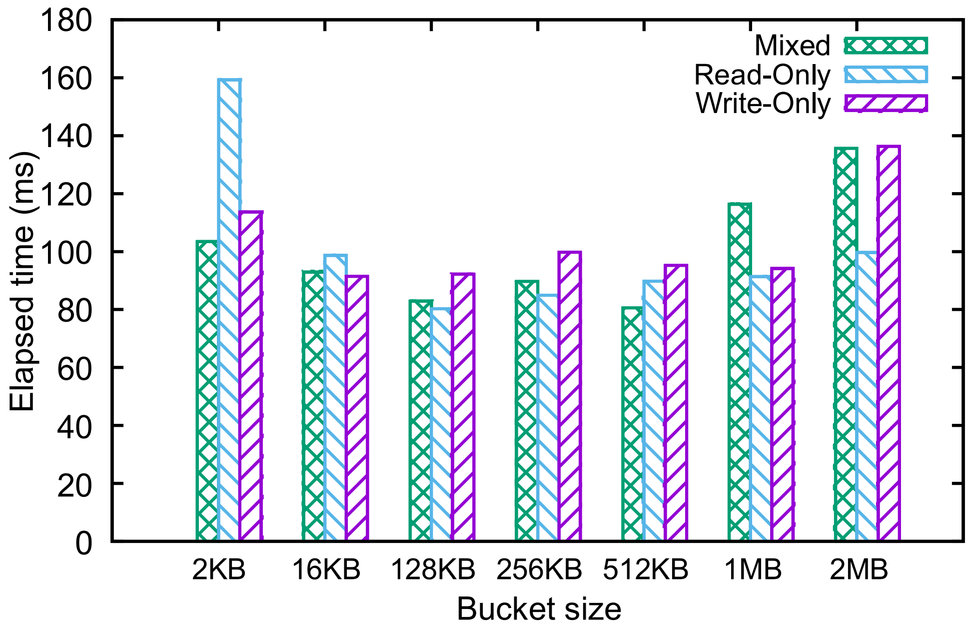
Evaluation Results – Library Overhead

- Input: 65536 POSIX calls
- Output: Average time spend in mapping in ns (per operation)
- Naïve: simple 1-file-to-1-object
- Overheads kept minimum
    - 0.0050 - 0.0080% of the overall execution time (Mapping time over I/O time)



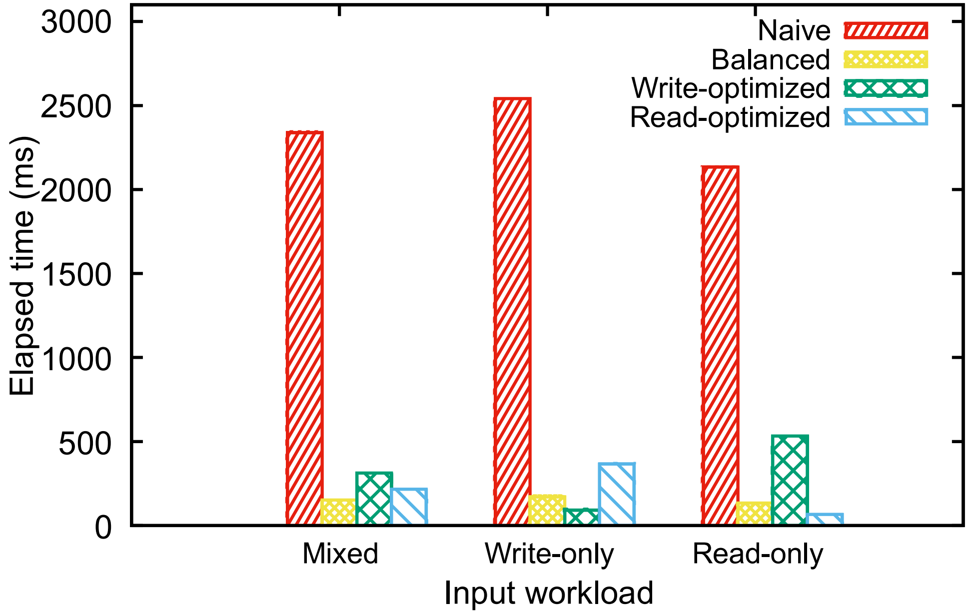
Evaluation Results – Bucket Size

- Balanced mapping uses buckets
- Input: 3 workloads
  - Mixed
  - Read-only
  - Write-only
- Request size: 128KB
- Output: Execution time in ms
- Best bucket size: close to median size of requests
  - Too small (i.e., 2KB) results in many objects
  - Too big (i.e., 2MB) results in excessive reading



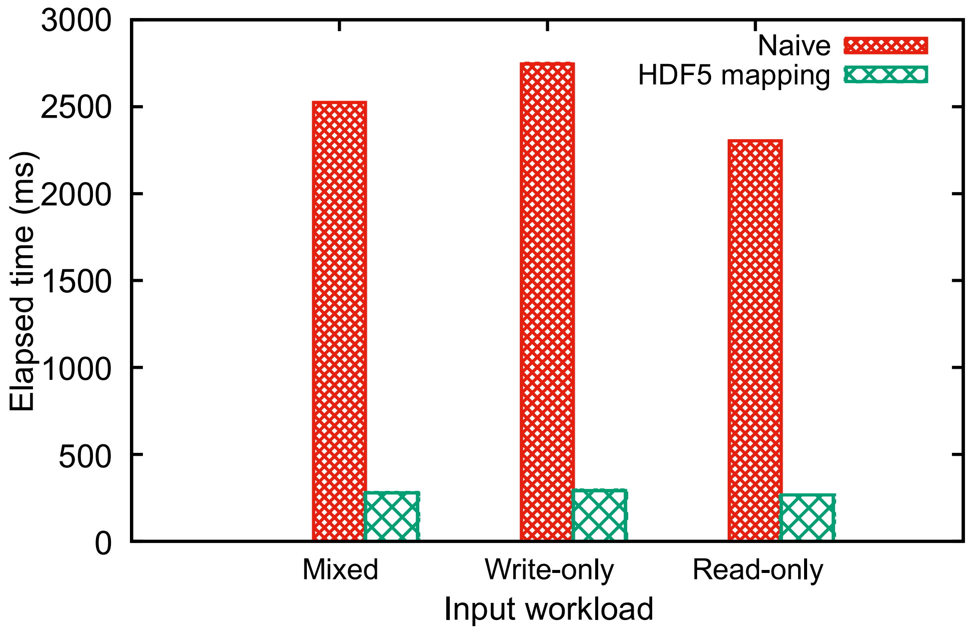
Evaluation Results – Synthetic Benchmark

- POSIX files
- Input: 3 workloads
  - Mixed
  - Read-only
  - Write-only
- Request size: 1MB
- Total I/O: 32MB
- Output: Execution time in ms
- Naïve: simple 1-file-to-1-object
- 15x speedup for Balanced and mixed input
- 32x speedup for Read-opt and read-only input
- 27x speedup for Write-opt and write-only input



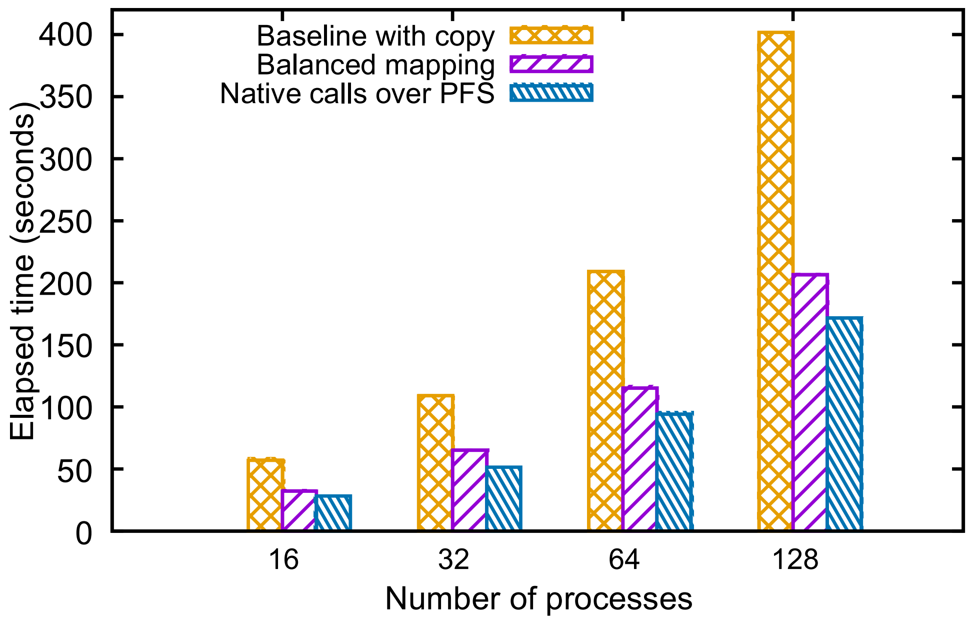
Evaluation Results – Synthetic Benchmark

- HDF5 file:
  - One dataset, integer datatype
- Input: 3 workloads
  - Mixed
  - Read-only
  - Write-only
- Request size: 1MB
- Total I/O: 32MB
- Output: Execution time in ms
- Naïve: simple 1-file-to-1-object
- 9x speedup for writes
- 8x speedup for reads



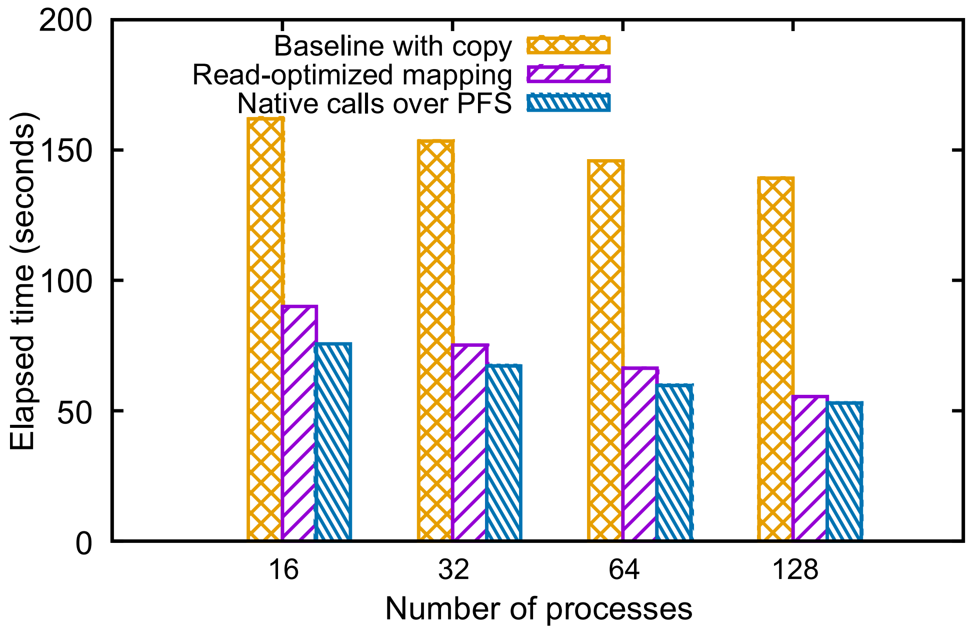
Evaluation Results – IOR

- Setup:
  - 4 client nodes, 4 servers
  - Num processes: 16, 32, 64, 128
  - Balanced Mapping mode
  - Strong scaling
- IOR:
  - MPI-IO
  - Block size = 2MB
  - Transfer size = 512KB
  - Total I/O = 512MB per process
  - File-per-process
  - DirectIO (-B, -Y options)
- OS buffering disabled
- Output: Execution time in seconds
- Baseline: first copy the input files from MongoDB to OrangeFS and then run IOR
- More than **2x speedup**



Evaluation Results – Montage

- Setup:
  - 4 client nodes, 4 servers
  - Num processes: 16, 32, 64, 128
  - Read-Optimized Mapping mode
  - Weak scaling
- Montage:
  - POSIX-IO
  - Total I/O = 24GB
- OS cache disabled and flushed before
- Output: Execution time in seconds
- Baseline: first copy the input images from MongoDB to OrangeFS and then run Montage
- More than **2x speedup**



Conclusions & Future Steps

- File-based vs Object-based Storage solutions
- Four new algorithms to map a file to one or more objects
- Evaluation shows:
  - 2x-30x speedup compared to 1-to-1 naïve mappings
  - More than 2x in real application scenarios
- Future work:
  - A new I/O management framework that integrates different storage subsystems and thus, get us closer to the convergence of HPC and BigData.
  - A system that offers universal data access regardless of the storage interface. Our mappings is a great first step ☺

Questions


thank you
danke
рахмат
謝謝
ngiyabonga
teşekkür ederim
gracias
merci
obrigado
bedankt
dziękuje
спасибо
Баярлалаа
go raibh maith agat
mochchakkeram
sukriya
kop khun krap
terima kasih
dank je
tapadh leat
dakujem
arigatô
grazie
мерси
감사합니다
ありがとう
ευχαριστώ
hvala
mauruuru
sagolun
takk
taiku
merci
enkosi
nanni
nandri
kiitos
dankie
dhanyavad
faafetai lava
vinaka
спасибі
blagodaram
mersi
barka
welalin
tack
spas
kia ora
misaotra
matondo
paldies
grazzi
mahalo
хвала
asante
manana
obrigada
murakoze
tenki
djiere dieuf
tau
дякую
mamnun
chokrane
chnorakaloutioun
gratias ago
gràcies
sulpáy
akun
dankon
aciu
köszönöm
gràcie
bayarlalaa
sobodi
děkuji
mèsi
didi madloba
najis tuke
kam sah hamnida
rahmat
tanemirt
rahmet
xièxie
diolch
dhanyavadagalu
shukriya
mercé
trugarez
তোমাকে ধন্যবাদ