EXPLORING HIGH PERFORMANCE STORAGE WITH DAOS

Kenneth Cain
Mohamad Chaarawi
Johann Lombardi

Hewlett Packard Enterprise

Adrian Jackson
(a.jackson@epcc.ed.ac.uk)
Nicolau Manubens

epcc

Aims

- Understand object storage hardware and software
- Learn about DAOS and filesystems on DAOS
- Learn about DAOS lower-level APIs and using them for your applications
- Get hands on with DAOS and high performance storage hardware
- Learn how to program, and design, for object storage systems

Hewlett Packard Enterprise

epcc

Aims cont.

- Understand/think about your application data requirements
  - Both storage and discovery
- Thinking about different ways you undertake I/O or storing data
- Move beyond bulk, block-based, I/O paradigms

Format

- Lectures and practicals
- Slides and exercise material available online:
  - https://github.com/adrianjhpc/ObjectStoreTutorial
  - Exercises will be done on remote system (NEXTGenIO)
  - We will provide access for these

Hewlett Packard Enterprise

epcc

Timetable

- 08.30 Introduction
- 08.40 Object storage and storage approaches
- 09.20 Practical: Benchmarking different storage approaches
- 09.30 DAOS programming APIs
- 10.00 Break
- 10.30 DAOS programming APIs, filesystems, and examples
- 11.30 Practical: Using DAOS for applications
- 11.50 Performance, design, and summary
- 12.00 Finish

Object Storage

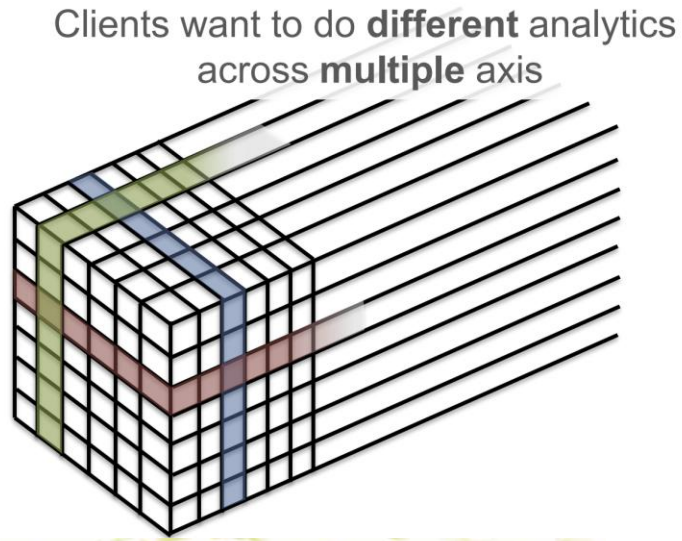
- Design and performance considerations are the challenge
  - Programming against the interfaces is (relatively) easy(ish)
  - Direct use often straight forward (i.e. filesystem interfaces)
  - More intelligent functionality takes more work/more specialised
- Design for functionality
  - When to store, when transactions should happen, what granularity I/O operations should be, what failures can you tolerate, etc..
- Design for performance
  - Memory size, I/O, data access costs, etc...

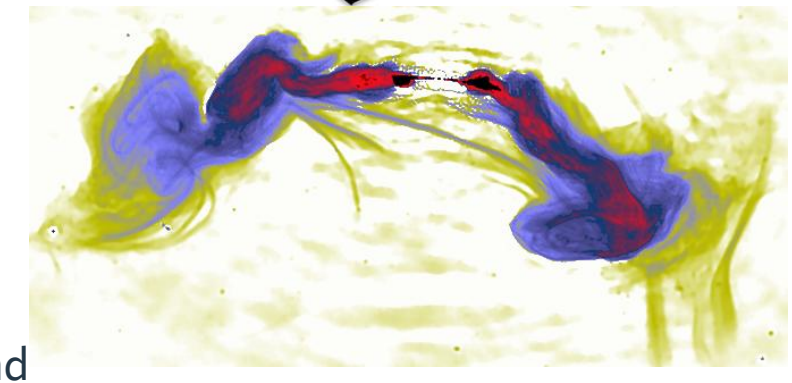
Object storage

- Data stored in unstructured objects
  - Data has identifier
  - Size and shape can vary
  - Metadata can also vary
- Originally designed for unstructured data sets
  - Bunch of data with no specific hierarchy required
- Can also enable efficient/fast access to data in different structures
  - Supports different creation, querying, analysis, and use patterns

Object storage

- Can enable efficient/fast access to data in different structures
  - Supports different creation, querying, analysis, and use patterns
- Granular storage with rich metadata
  - Data retrieval leverages metadata
  - **Build structure on the fly**
- Weather/climate
  - Pursuing optimal I/O for applications
    - Weather forecasting workflows
  - End-to-end workflow performance important
  - Simulation (data generation) only one part
    - Consumption workloads different layout/pattern from production
- Radio astronomy
  - Data collected and stored by antenna (frequency and location) and capture time
  - Reconstruction of images done in time order
  - Evaluation of transients or other phenomenon undertaken across frequency and location





Summary

- Please don't hesitate to ask questions!
- We have a practical sessions
  - Login account will remain active for you to try out using DAOS after the tutorial (not really enough time to finish them during the tutorial today)
  - Email a.jackson@epcc.ed.ac.uk to get an account on the system we will use for practical/try out sessions (if you use the subject "Object store account" that will help us respond to the emails)

Hewlett Packard Enterprise

epcc

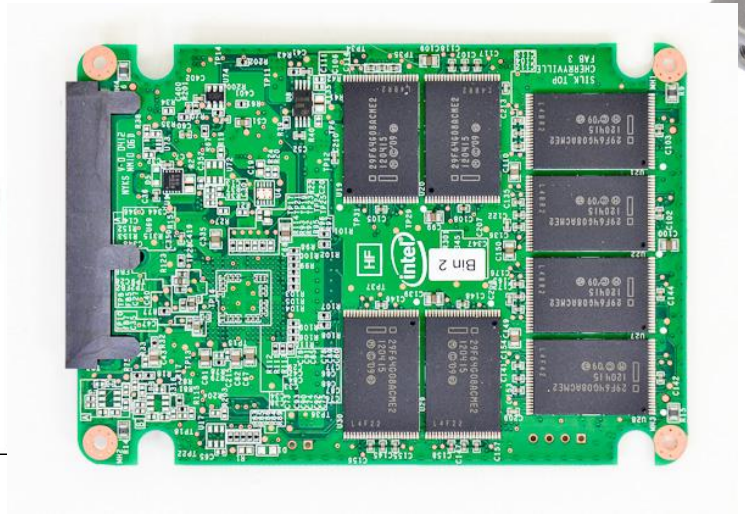
OBJECT STORES AND I/O APPROACHES

Hardware and Software interfaces

Hewlett Packard Enterprise

epcc

Storage



Hewlett Packard Enterprise

epcc

Filesystems

- Lots of ways to store data on storage devices
- Filesystems have two components:
  - Data storage
  - Indexing
- Data stored in blocks
  - Chunks of data physically stored on hardware somewhere
- Indexing is used to associate names with blocks

  - File names are the index
  - Files may consist of many blocks
- Variable sized nature of files makes this a hard problem to solve

Filesystems

- Different ways of defining how inodes → blocks, and how directories, filenames, etc… are structured
  - As well as alternative approaches (i.e. log-structured filesystem)
  - Extended functionality (replication, distribution, backup, erasure coding, etc…)
- These are what differentiate filesystems, i.e.:
  - ext*: ext3, ext4
  - xfs
  - zfs
  - btrfs
  - etc…
- Maybe be important for performance or required functionality but the default can be used by most

Parallel filesystems

- Build on local filesystem but provide
  - Aggregated distributed local filesystem
  - Custom approach to define how inodes → blocks, and how directories, filenames, etc... are structured
  - Relaxed consistency (potentially) for concurrent writing
- i.e. Lustre:
  - Open-source parallel file system
  - Three main parts
    - Object Storage Servers (OSS)
      - Store data on one or more Object Storage Targets (OST)
      - The OST handles interaction between client data request and underlying physical storage
      - OSS typically serves 2-8 targets, each target a local disk system
      - Capacity of the file system is the sum of the capacities provided by the targets (roughly)
      - The OSS operate in parallel, independent of one another
    - Metadata Target (MDT)
      - One(ish) per filesystem
      - Storing all metadata: filenames, directories, permissions, file layout
      - Stored on Metadata Server (MDS)
  - Clients
    - Supports standard POSIX access

POSIX I/O

- Standard interface to files
  - Linux approach
  - Based on systems with single filesystem
  - open, close, write, read, etc...
- Does not support parallel or HPC I/O well
  - Designed for one active writer
  - Consistency requirements hamper performance
  - Has a bunch of functions that can impact performance, i.e. locking (flock, etc...)
- Some filesystems/approaches relax POSIX semantics to improve performance
  - Moving beyond filesystems allows other semantics to be targeted

Object storage

- Filesystems use Files
  - container for blocks of data
  - lowest level of metadata granularity (not quite true)
- Object stores use Objects
  - container for data elements
  - lowest level of metadata granularity
- Allows individual pieces of data to be:
  - Stored
  - Indexed
  - Accessed separately
- Allows independent read/write access to “blocks” of data

Object storage

- Generally restricted interface
  - Put: Create a new object
  - Get: Retrieve the object
- Removes the requirements for lots of functionality r.e. POSIX style I/O
- Traditionally objects are immutable
  - Once created cannot be changed
  - This removes the locking requirement seen for file writes
  - Makes updates similar to log-append filesystems, i.e. copy and update
  - Can cause capacity issues (although objects can be deleted)
- Object ID generated when created
  - Used for access
  - Can be used for location purposes in some systems

Object stores

- Often helper services and interfaces
  - Manage metadata
  - Permissions
  - Querying
  - Etc...
- Distribution and redundancy etc... part of the complexity
  - Often eventual consistency
- Lots of complexity in implementations
- Commonly use web interfaces as part of the Put/Get interface

S3 – Simple Storage Service

- AWS storage service/interface
  - Defacto storage interface for a range of object stores
- Uses a container model
  - Buckets contain objects
  - Buckets are the location point for data
  - Defined access control, accounting, logging, etc...
  - Bucket names have to be globally unique
- Buckets can be unlimited in size
  - Maximum object size is 5TB
  - Maximum single upload is 5GB
- A bucket has no object structure/hierarchy
  - User needs to define the logic of storage layout themselves (if there is any)
- Fundamental operations corresponding to HTTP actions:
  - http://bucket.s3.amazonaws.com/object
  - POST a new object or update an existing object.
  - GET an existing object from a bucket.
  - DELETE an object from the bucket
  - LIST keys present in a bucket, with a filter.

S3

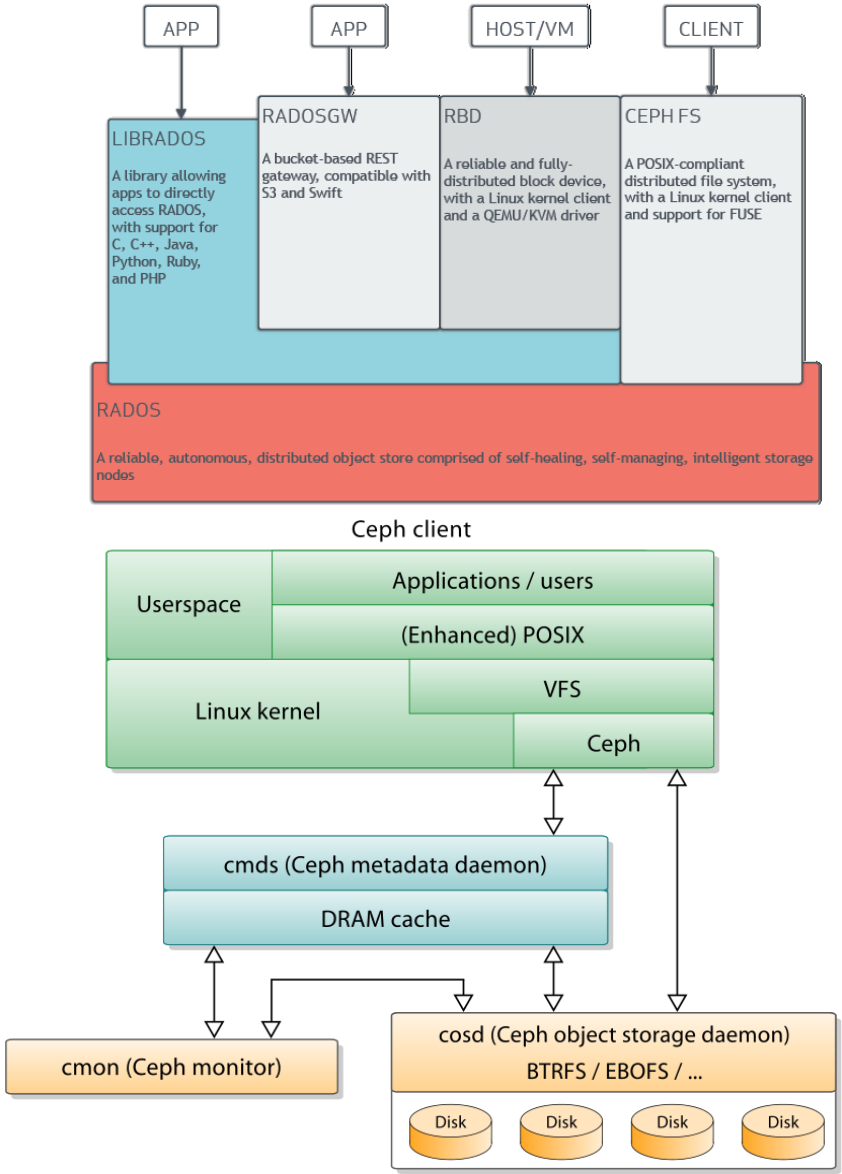
- Objects are combination of data and metadata
- Metadata is name – value pair (key) identifying the object
  - Default has some other information as well:
    - Date last modified
    - HTTP Content-Type
    - Version (if enabled)
    - Access Control List (if configured)
  - Can add custom metadata
- Data
  - An object value can be any sequence of bytes (up to 5TB)
  - Multi-part upload to create/update objects larger than 5GB (recommended over 100MB)

S3 Consistency Model

- Strong RAW (read after write) consistency
  - PUT (new and overwrite) and DELETE operations
  - READ on metadata also strong consistency
  - Across all AWS regions
- Single object updates are atomic
  - GET will either get fully old data or fully new data after update
  - Can't link (at the S3 level) key updates to make them atomic
- Concurrent writers are *racy*
  - No automatic locking
- Bucket operations are eventually consistent
  - Deleted buckets may still appear after the delete has occurred
  - Versioned buckets may take some time to setup up initially (15 minutes)

Ceph

- Widely used object store from academic storage project
- Designed to support multiple targets
  - Traditional object store: RadosGW → S3 or Swift
  - Block interface: RBD
  - Filesystem: Ceph FS
  - Lower-level object store: LibRados
- Distributed/replicated functionality
  - Scale out by adding more Ceph servers
  - Automatic replication/consistency
    - replication, erasure coding, snapshots and clones
- Supports striping
  - Has to be done manually if using librados
- Supports tiering
- Lacking production RDMA support



Hewlett Packard Enterprise

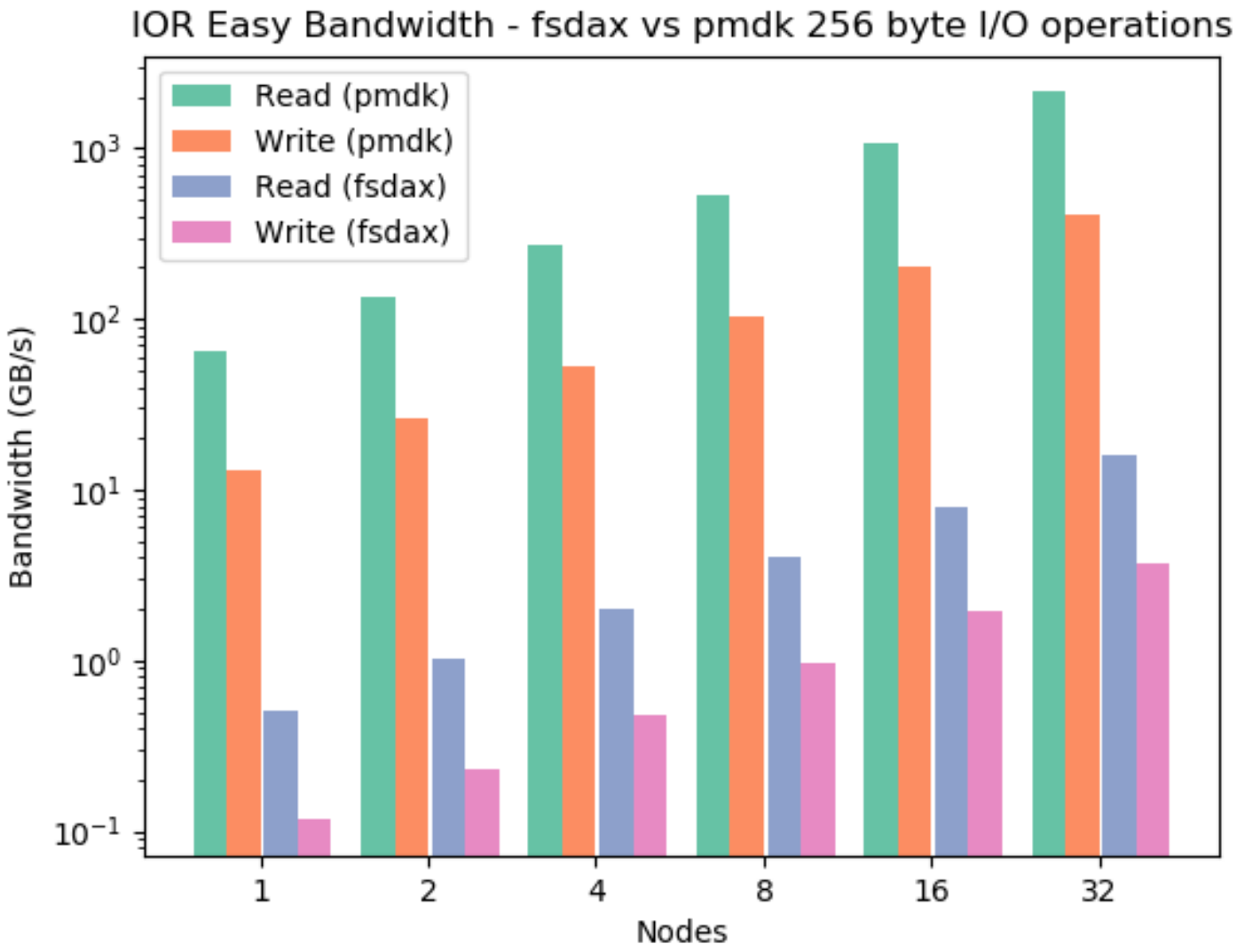
epcc

NVRAM

Hewlett Packard Enterprise

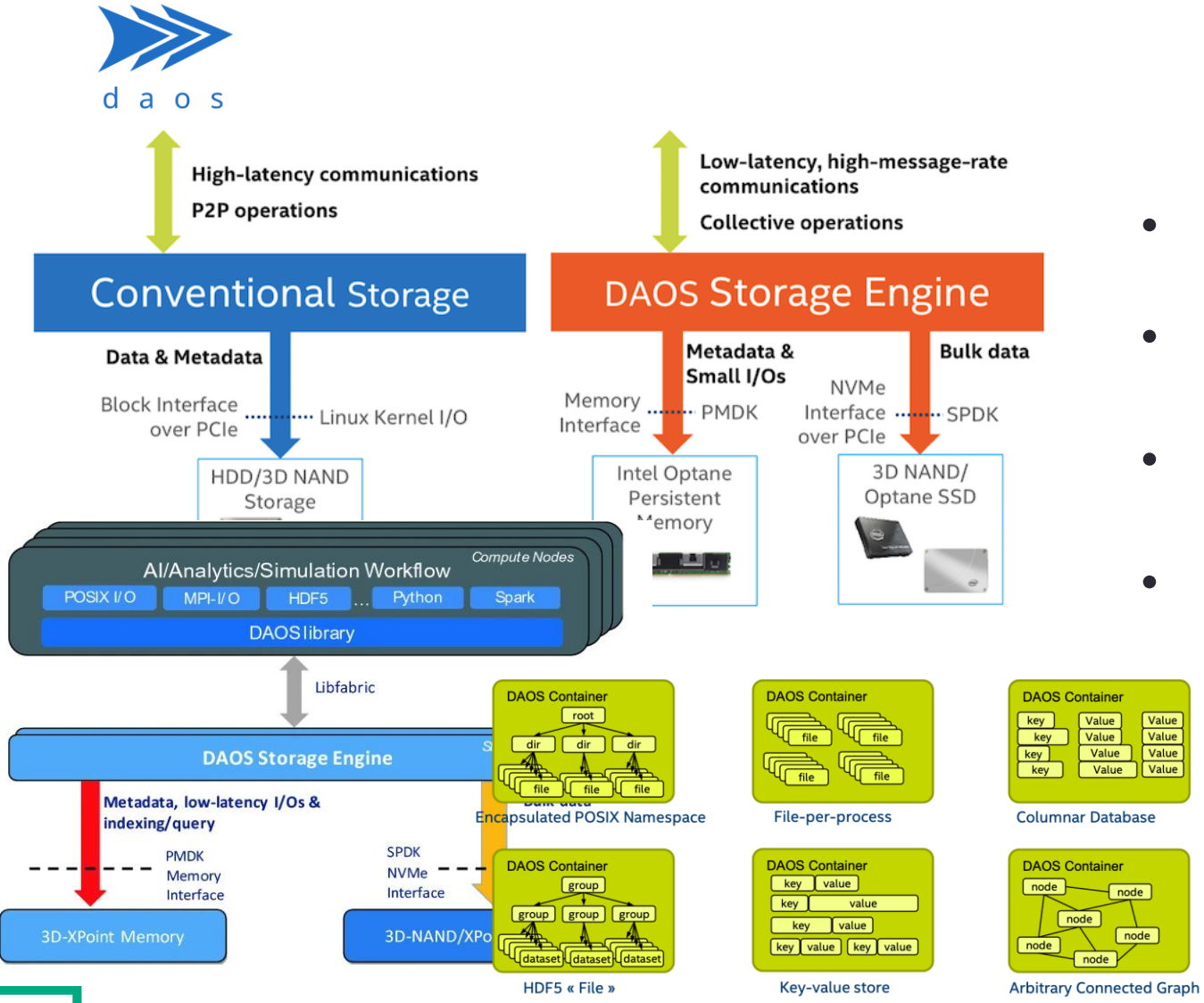
epcc

Optane



Hewlett Packard Enterprise

epcc



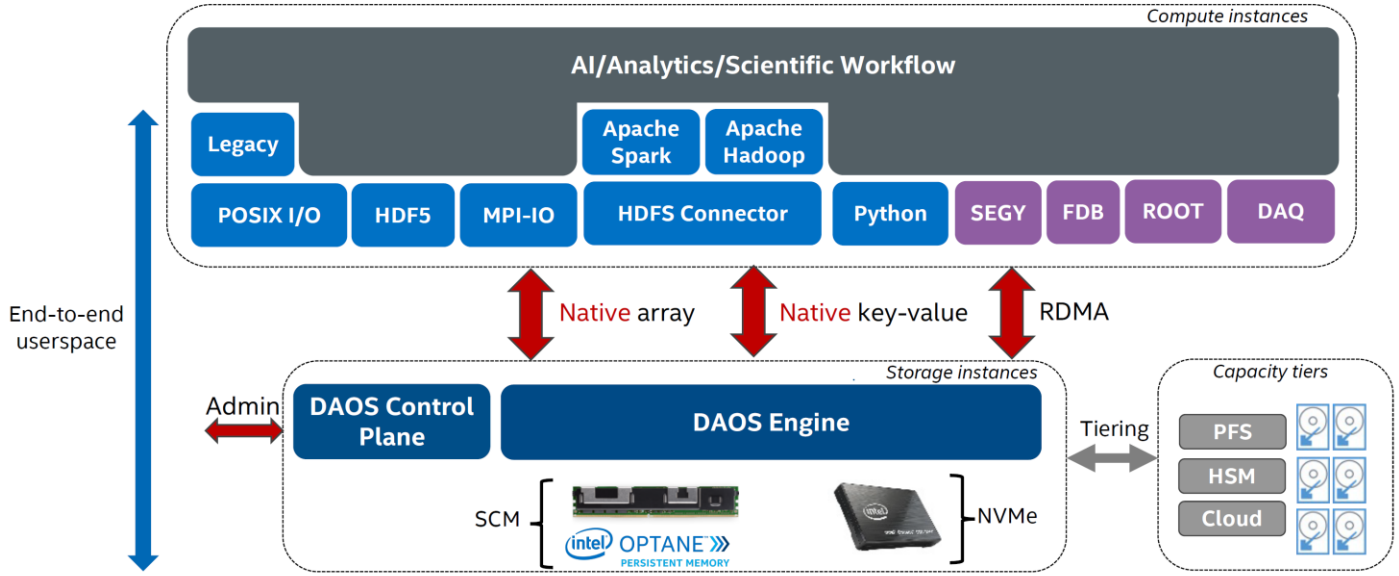
- Native object store on non-volatile memory and NVMe devices and designed for HPC
- Pools
  - Define hardware range of data
- Containers
  - User space and data configuration definitions
- Objects
  - **Multi-level key-array** API is the native object interface with locality
  - **Key-value** API provides a simple key and variable-length value interface. It supports the traditional put, get, remove and list operations.
  - **Array API** implements a one-dimensional array of fixed-size elements addressed by a 64-bit offset. A DAOS array supports arbitrary extent read, write and punch operations.

Hewlett Packard Enterprise

epcc

daos

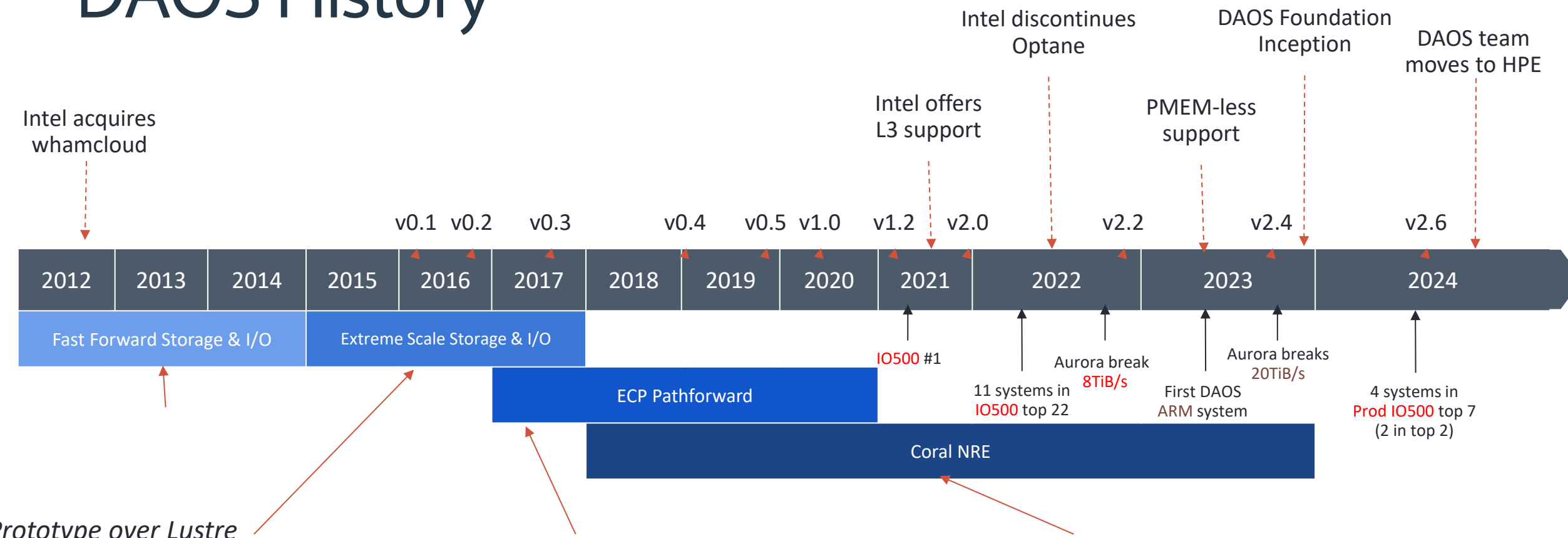
- Range of storage interfaces
  - Native object store (libdaos)
  - Filesystem (various approaches)
  - Raw block device
  - MPI-I/O (ROMIO)
  - HDF5
  - PyDAOS
  - Spark/Hadoop
  - TensorFlow I/O
- DAOS systems built from DAOS servers
  - One per socket, has own NVMe and NVRAM
  - Scale system by adding more servers (in node or across nodes)
  - Metadata and data entirely distributed/replicated (no metadata centralisation)
  - RAFT-approach used for consensus across servers



Hewlett Packard Enterprise

epcc

DAOS History



Prototype over Lustre

- Build over ZFS OSD
- DAOS API over Lustre

Standalone prototype

- OS-bypass
- Persistent memory via PMDK
- Replication & self healing

DAOS embedded on FPGA

- Disaggregated I/O
- Monitoring
- NVMe SSD support via SPDK

DAOS Productization for Aurora

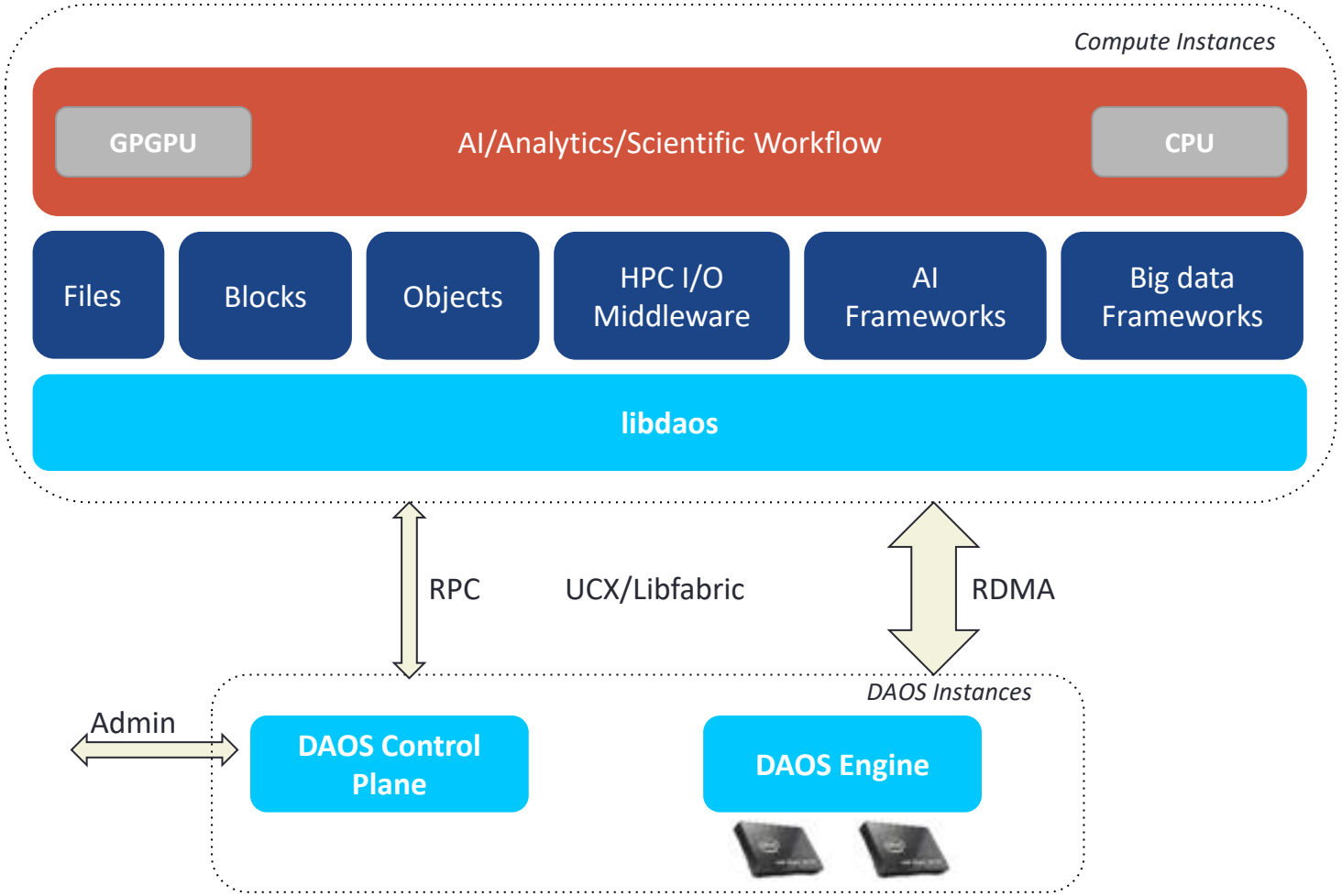
- Hardening
- 10+ new features
- Support for extra AI/Big data frameworks

Hewlett Packard Enterprise

epcc

DAOS: Nextgen Open Storage Platform

- Platform for innovation
- Files, blocks, objects and more
- Full end-to-end userspace
- Flexible built-in data protection
  - EC/replication with self-healing
- Flexible network layer
- Efficient single server
  - O(100)GB/s and O(1M) IOPS per server
- Highly scalable
  - TB/s and billions IOPS of aggregated performance
  - O(1M) client processes
- Time to first byte in O(10) µs



Hewlett Packard Enterprise

epcc

DAOS Design Fundamentals

- No read-modify-write on I/O path (use versioning)
- No locking/DLM
- No client tracking or client recovery
- No centralized (meta)data server
- No global object table
- Non-blocking I/O processing (futures & promises)
- Serializable distributed transactions
- Built-in multi-tenancy
- User snapshot

Scalability & Performance

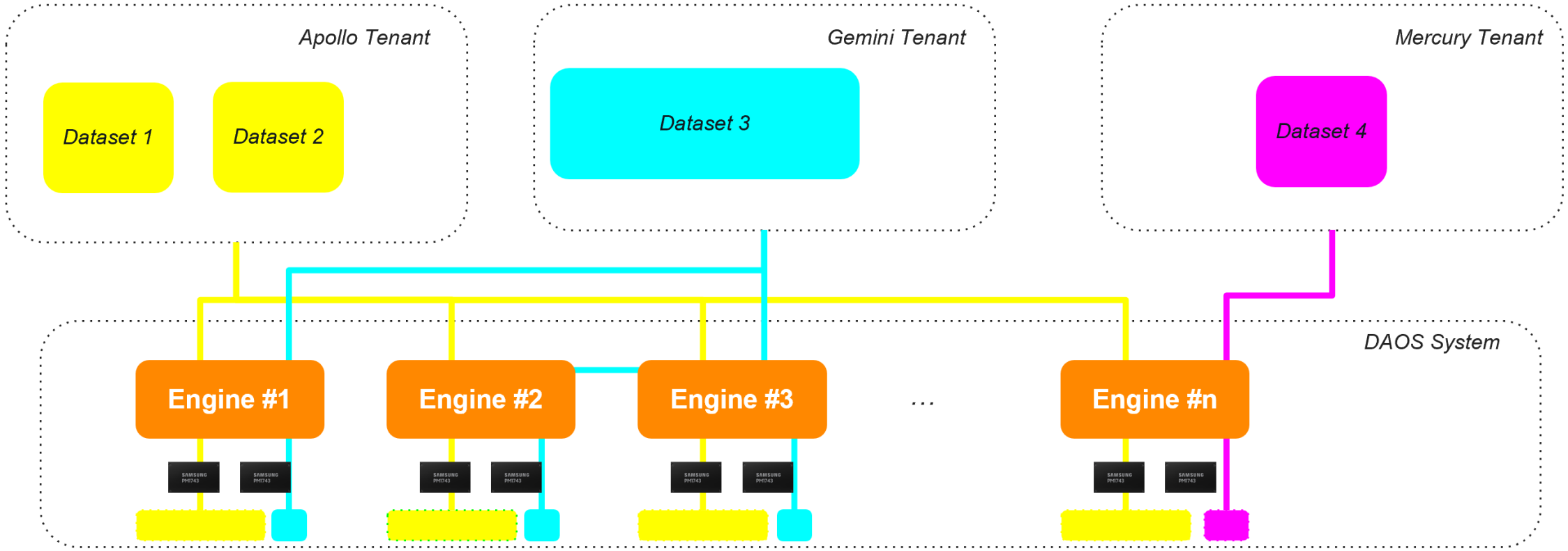
High IOPS

Unique Capabilities

Hewlett Packard Enterprise

epcc

Flexibility in data space creation and operation



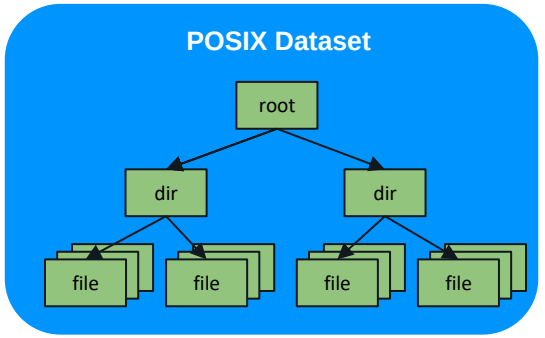
Pool 1	Apollo Tenant	100PB	20TB/s	200M IOPS
Pool 2	Gemini Tenant	10PB	2TB/s	20M IOPS
Pool 3	Mercury Tenant	30TB	80GB/s	2M IOPS

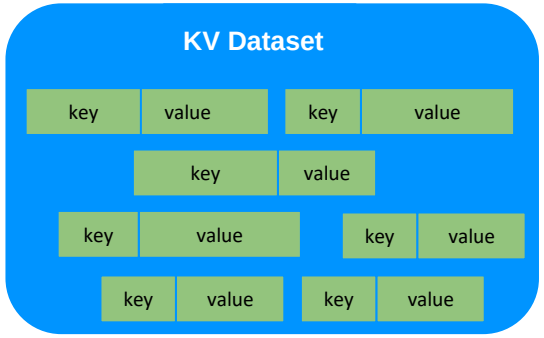
Hewlett Packard Enterprise

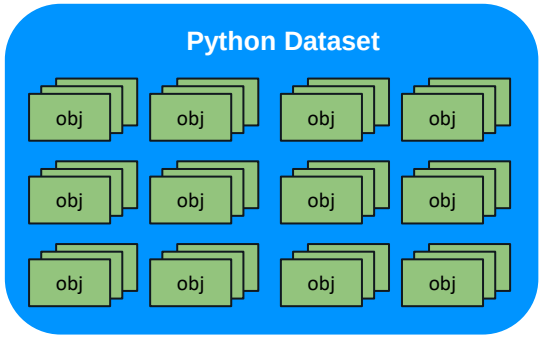
epcc

Beyond files

- New data model not based on files and file based approaches
- Introduce notion of dataset
- Basic unit of storage
- Datasets have a type
- POSIX datasets can include trillions of files/directories
- Advanced dataset query capabilities
- Unit of snapshots
- ACLs/IAM per dataset





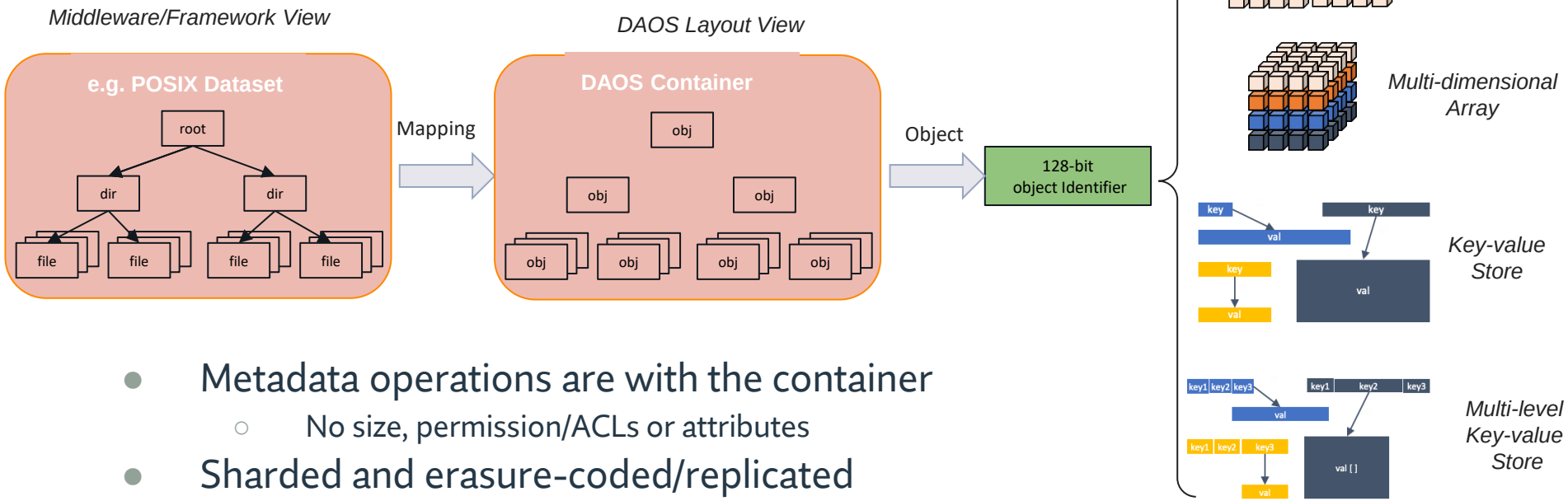


Hewlett Packard Enterprise

epcc

Objects

- Objects are the final level of storage for DAOS



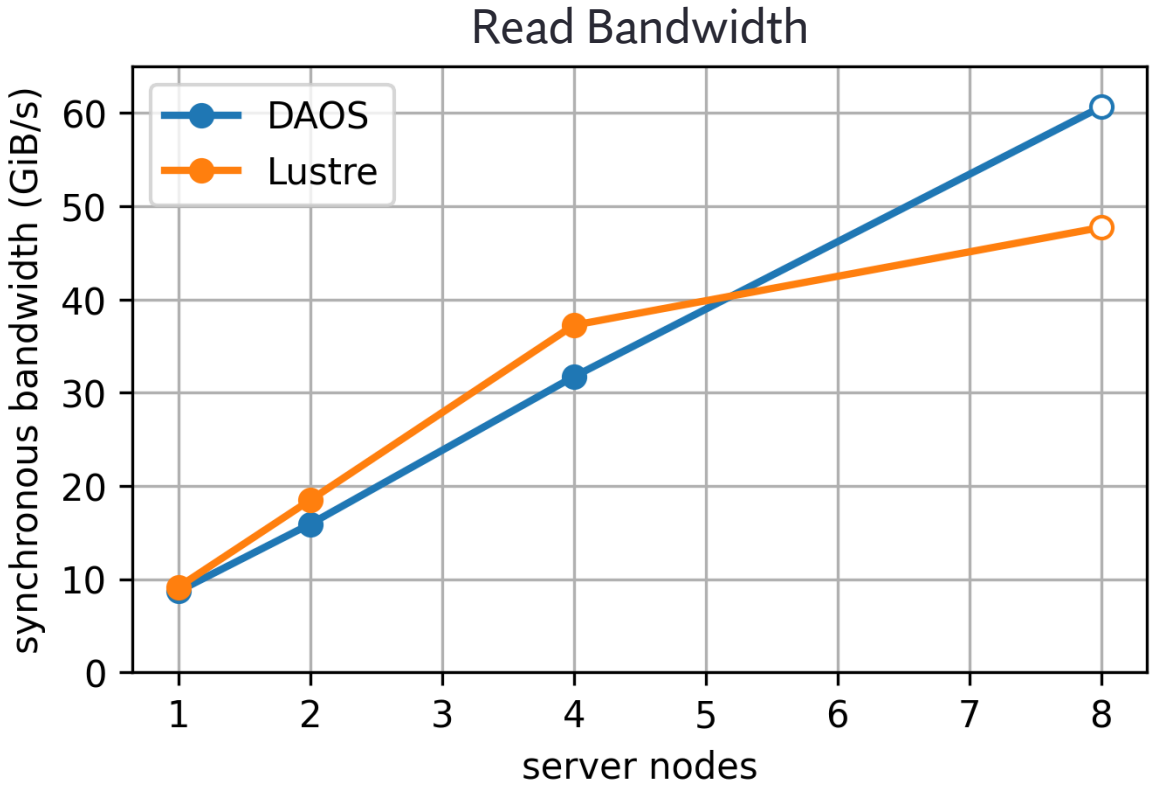
  - Metadata operations are with the container
    - No size, permission/ACLs or attributes
  - Sharded and erasure-coded/replicated
  - Algorithmic object placement
  - Very short Time To First Byte (TTFB)

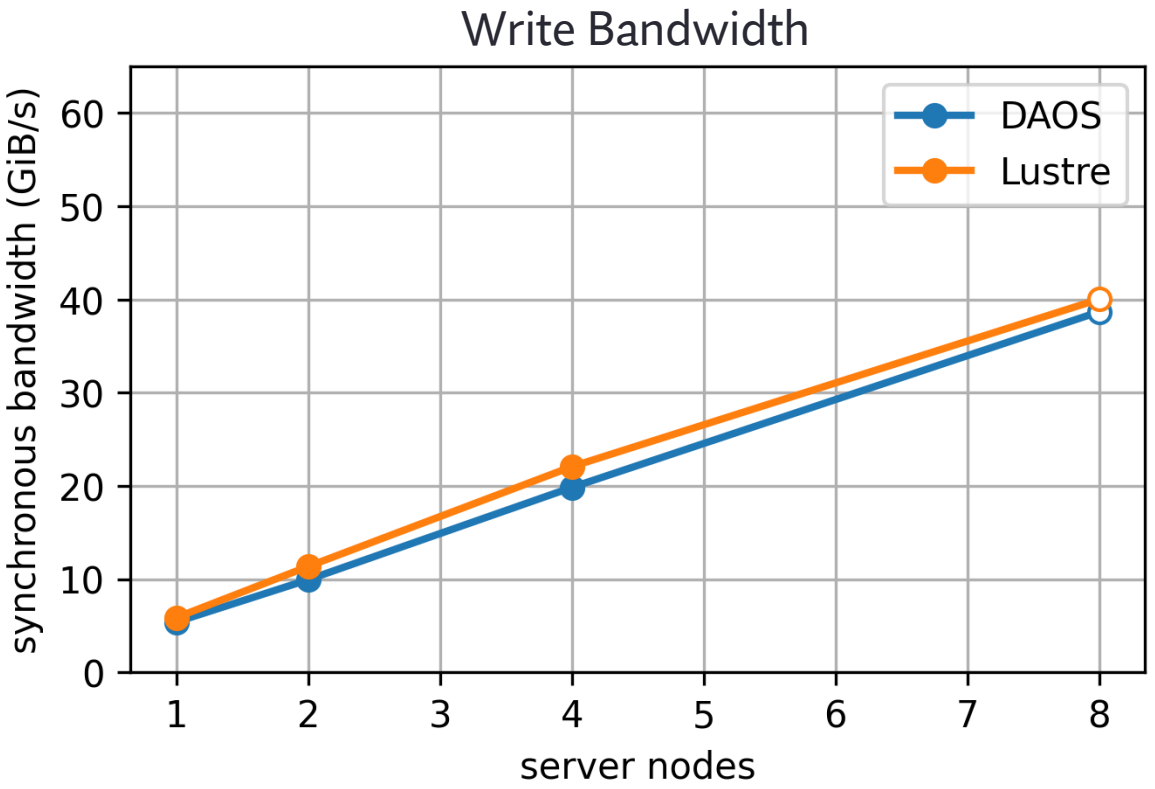
Hewlett Packard Enterprise

epcc

DAOS Performance

- Comparing Lustre and DAOS on the same hardware
  - IOR bulk synchronous I/O



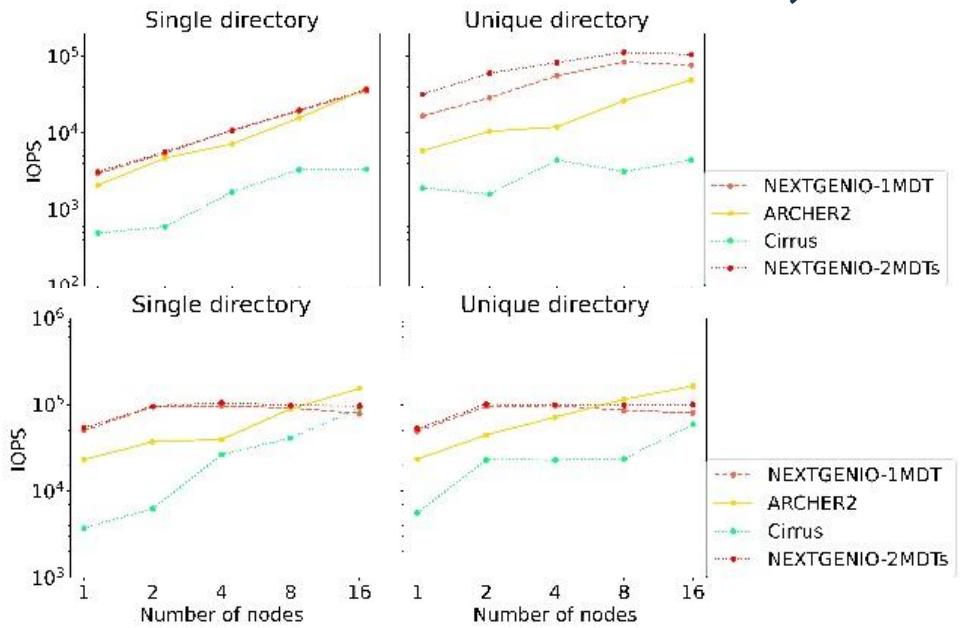


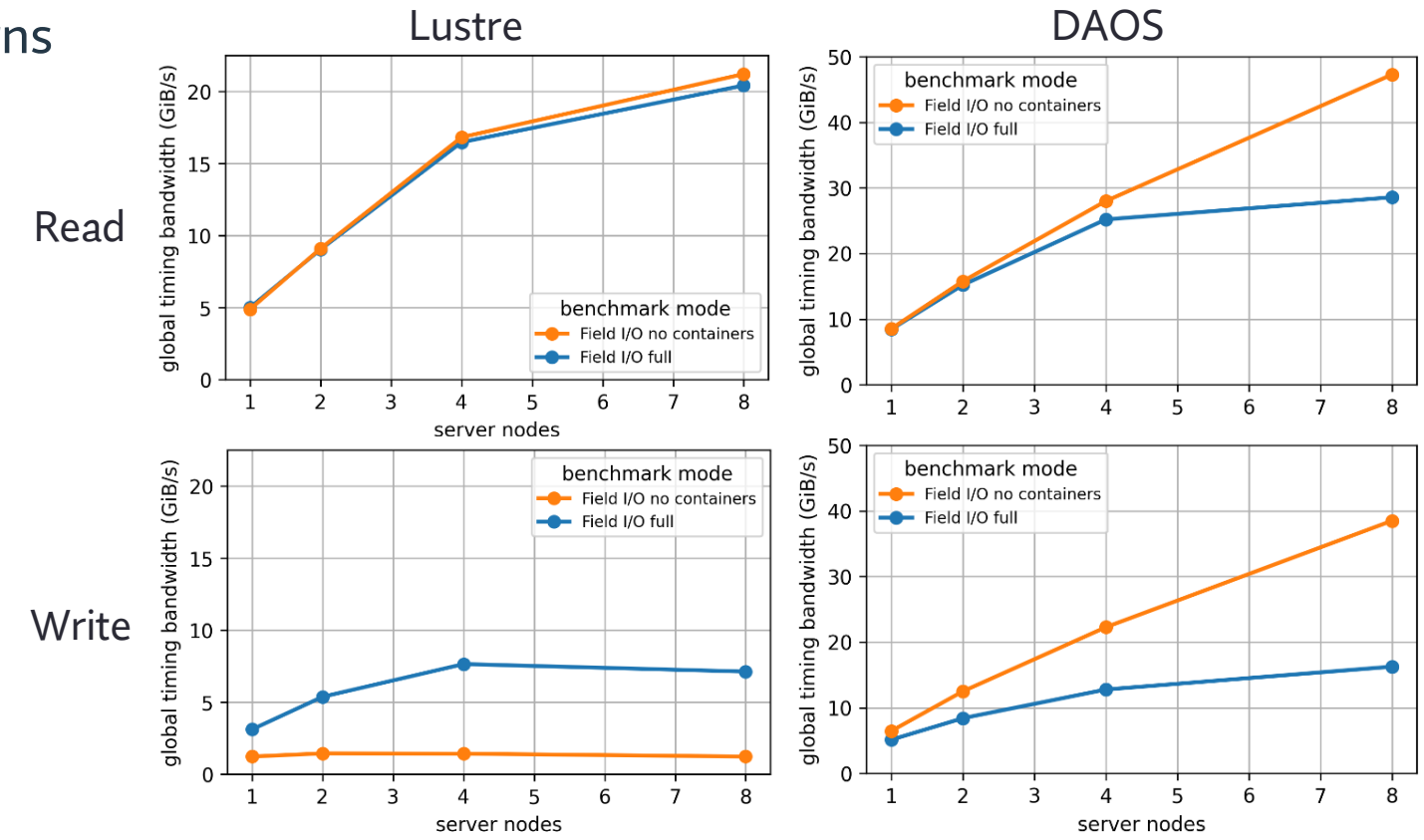
Hewlett Packard Enterprise

epcc

DAOS performance

- Separate read and write steps
  - More “object like” access patterns
  - Weather field -> Object or file



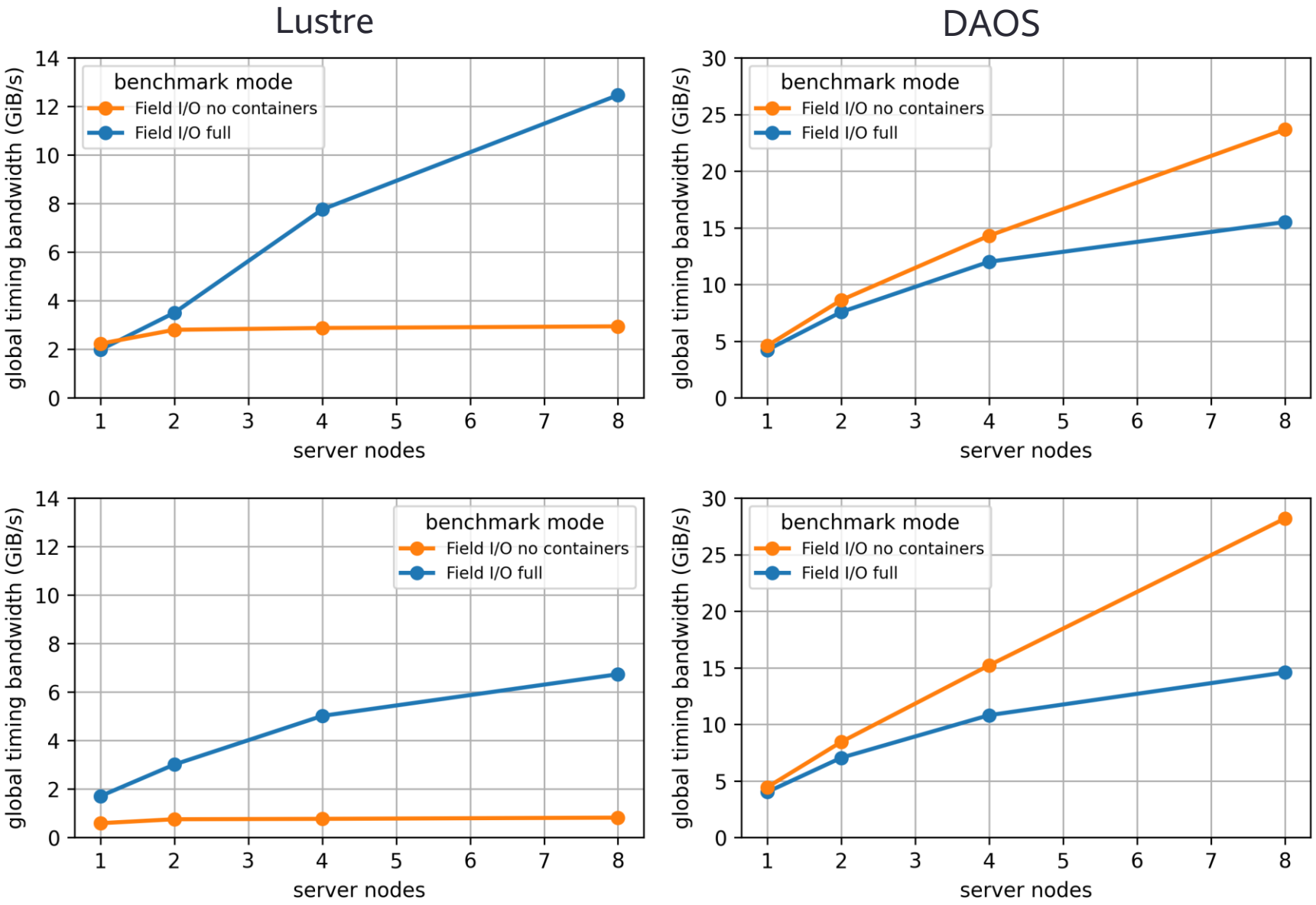


Hewlett Packard Enterprise

epcc

DAOS performance

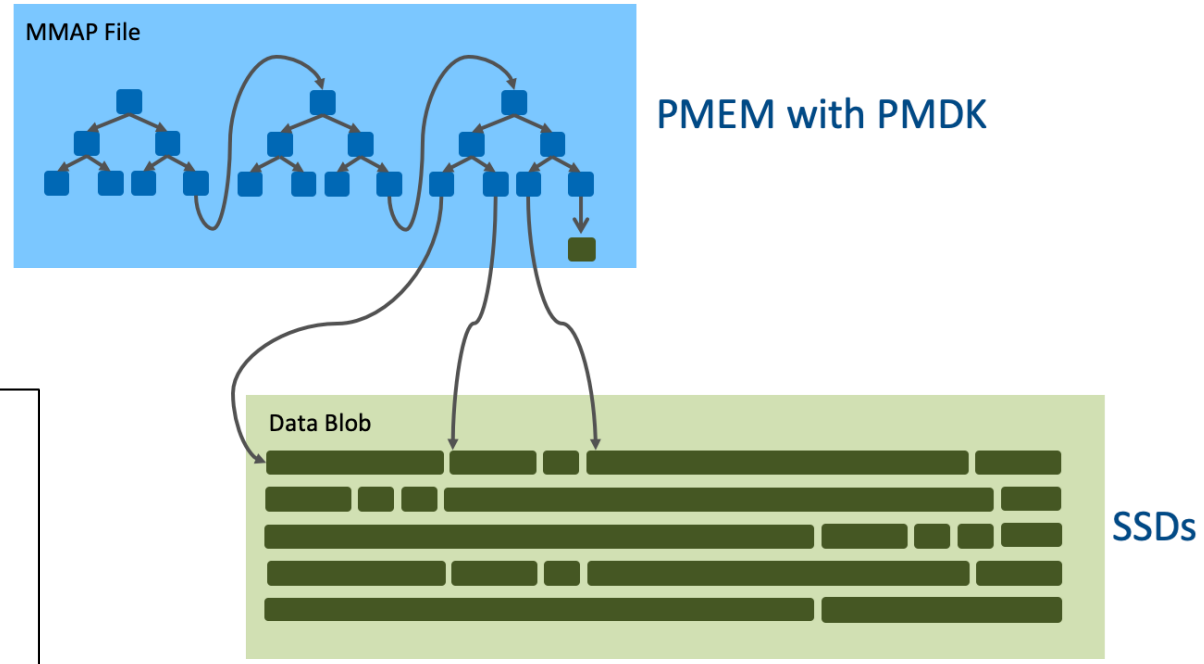
- Contending read and write workers
  - Containers represent filesystem or object store structure



Hewlett Packard Enterprise

epcc

DAOS beyond NVRAM/Optane/PMem

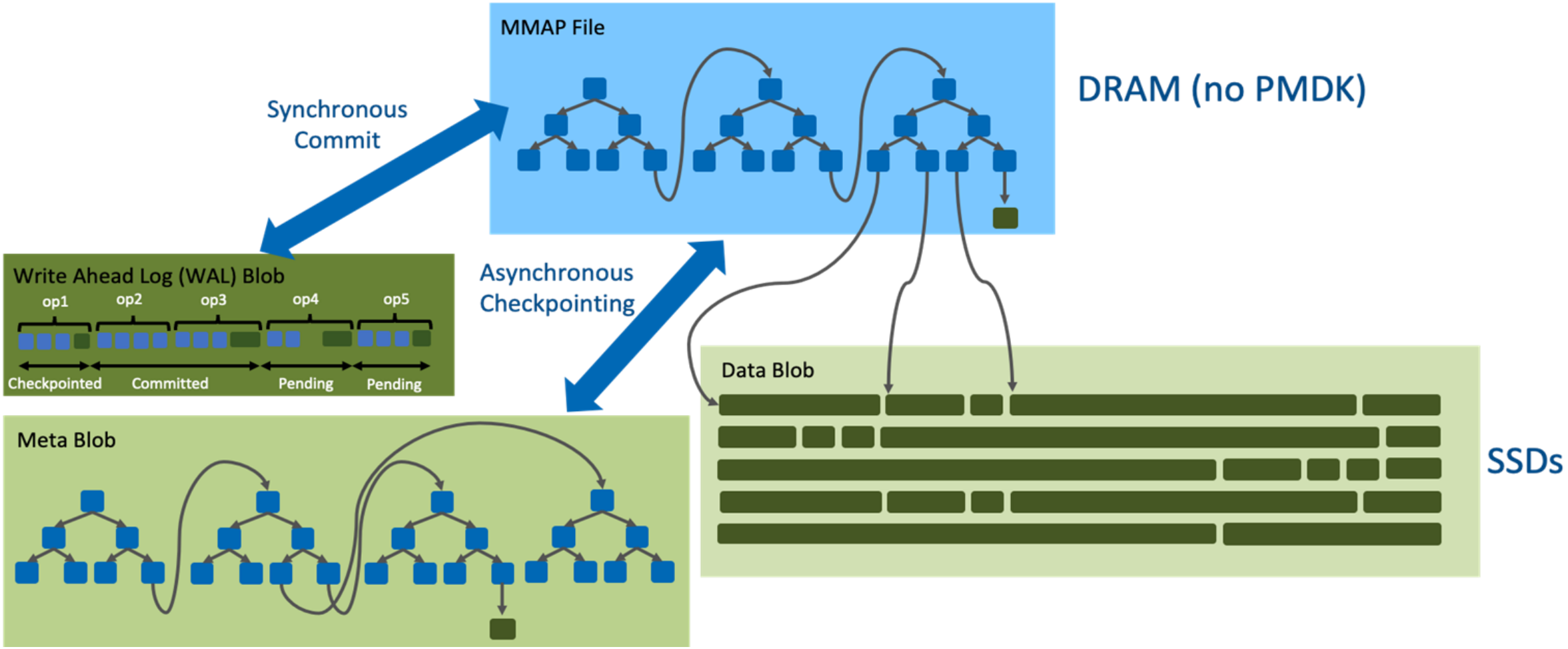


- Persistent metadata
- Require Intel Optane PMEM (or NVDIMM-N)
- App Direct mode
- Mode used on Aurora

Hewlett Packard Enterprise

epcc

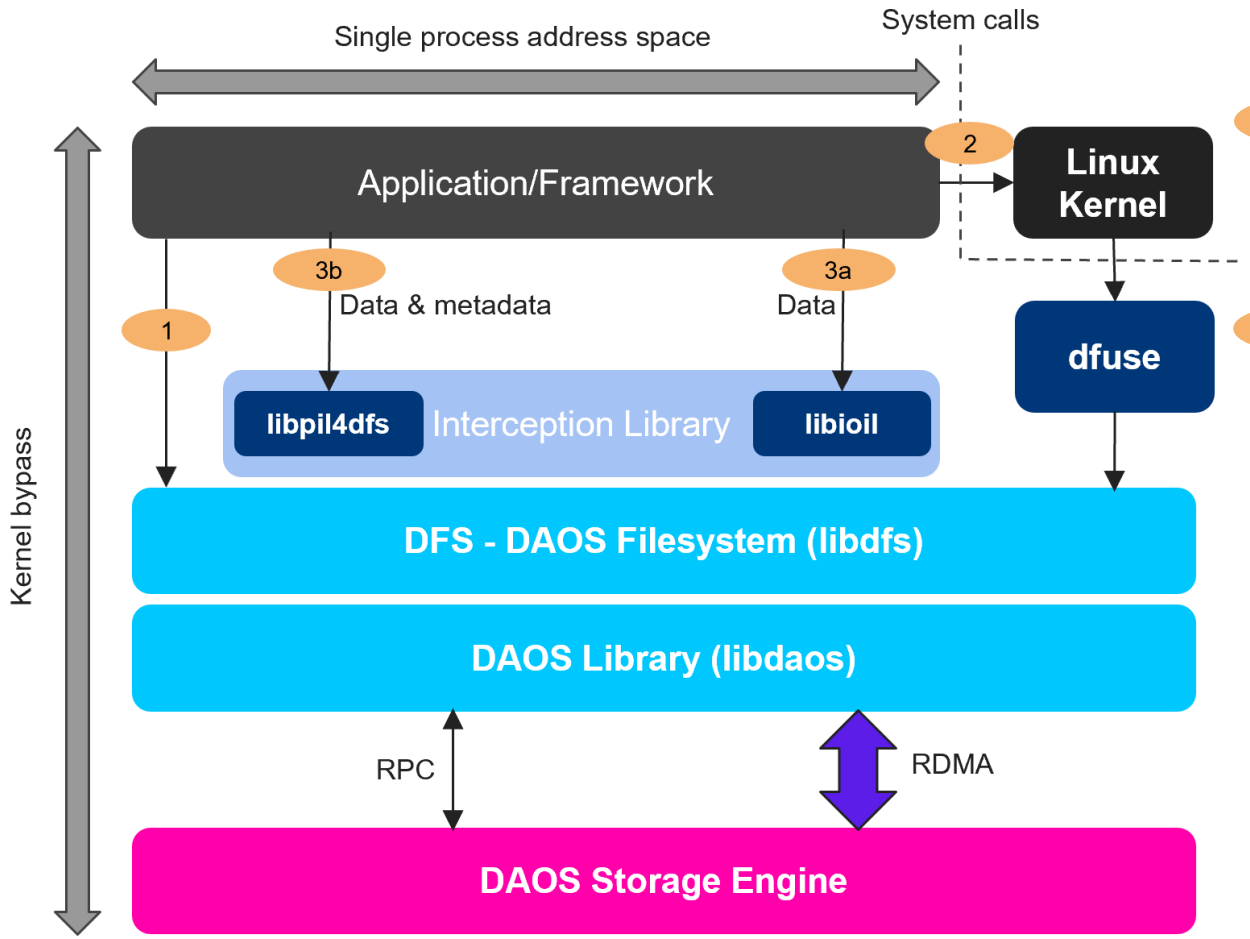
DAOS beyond NVRAM/Optane/PMem



Hewlett Packard Enterprise

epcc

DAOS access approaches



1. Userspace DFS library with API like POSIX
   - **Require** application changes
   - Low latency & high concurrency
   - No caching
2. DFUSE daemon to support POSIX API
   - **No** application changes
   - VFS mount point & high latency
   - Caching by Linux kernel
3. DFUSE + Interception library
   - **No** application changes
   - 2 flavours using LD_PRELOAD
   - 3a libioil
     - (f)read/write interception
     - Metadata via dfuse
   - 3b libpil4dfs
     - Data & metadata interception
     - Aim at delivering same performance as #1 w/o any application change
     - mmap & binary execution via fuse

Hewlett Packard Enterprise

epcc

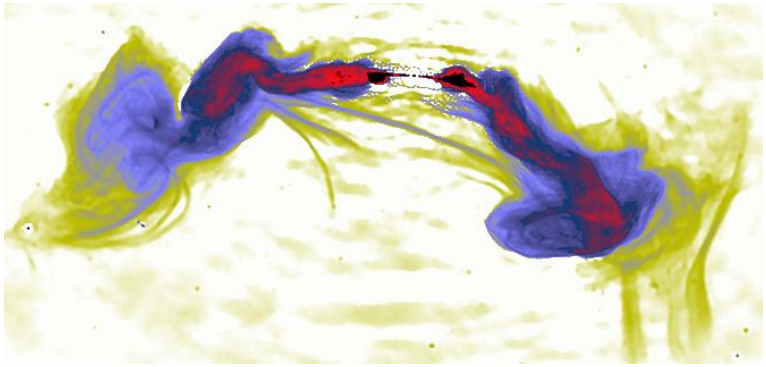
Object stores

- High performance object stores offer:
  - Server-side consistency by default
    - reducing round trip messaging for some operations
  - Distributed metadata functionality
    - no single performance bottleneck
  - Small object size performance
    - non-kernel space I/O operations so don't have interrupt/context switch performance issues
    - designed for faster hardware and for large scale operation
  - Decoupled metadata from data
  - In-built redundancy control/configuration
  - Multi-versioning and transactions to reduce contention/provide consistency tools
  - Scaling across storage resources
  - Searching/discovery across varying data dimensions

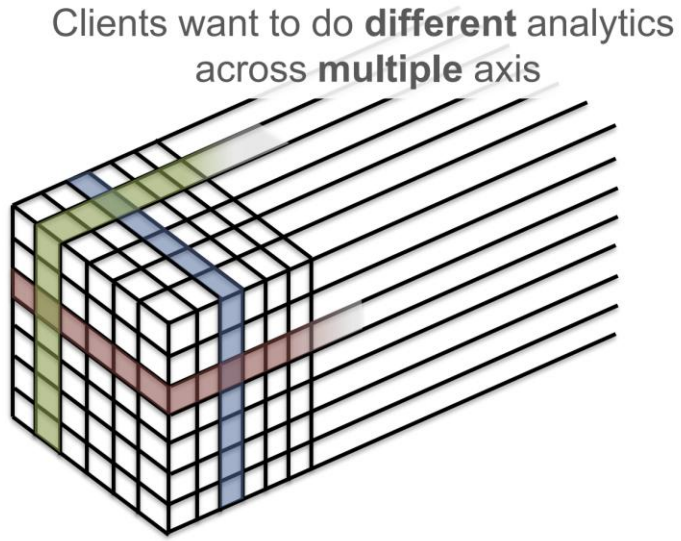
Object stores

- Object can have as much or as little complexity as you want
  - Single array
  - Single key-value
  - Nested object containing table of entries
  - etc..
- High performance object stores can't....
  - Beat filesystems for bulk I/O with low metadata overheads
  - Support high performance alternative functionality without porting effort
  - Eliminate server side contention
  - Fix poor storage design
  - Create your data layout and indexing for you
  - Fix configuration/resource issues

- Object Stores can unlock previously expensive I/O patterns



- Enable discovery as well as storage



Hewlett Packard Enterprise

epcc

Practical Setup

- https://github.com/adrianjhpc/ObjectStoreTutorial/Exercises/exercisesheet.pdf
- Take IOR source code
- Run on the EPCC system
- SSH details will be sent
- You will get a username
  - ngguestXX
  - Password and key

Hewlett Packard Enterprise

epcc

Hewlett Packard
Enterprise

Tutorial on the DAOS API

Kenneth Cain, Mohamad Chaarawi, Jerome Soumagne HPE
Adrian Jackson (EPCC, The University of Edinburgh)

May 5, 2025

Presentation Outline

- Pools:
  - Connect, disconnect
- Containers:
  - Create, destroy, open, close
- Objects: access APIs based on type
  - Flat KVS
  - Global array
  - Multi-level KVS
- POSIX Support:
  - DFS API (for modified applications to use daos)
  - dfuse mount, interception libraries (for unmodified applications to use daos)
  - Best practices

DAOS API Usage, Program Flow

- Initialize DAOS stack
- Connect to a Pool
- Create / open a container
- Access an object in the container through the unique OID
  - Open object
  - update/fetch/list
  - Close object
- Close / disconnect from container & pool, finalize DAOS stack

Program Flow – Initialize DAOS, Connect to a Pool

- First (typically): initialize DAOS, connect to your pool:

```
int daos_init(void);
int daos_pool_connect(const char *pool, const char *sys,
      unsigned int flags, daos_handle_t *poh,
      daos_pool_info_t *info, daos_event_t *ev);
```

Note: pool already exists

Administrator used daos management utility "dmg" to create a pool – e.g.,

```
dmg pool create
--size=10TB mypool
```

- MPI program: connect from 1 rank, serialize handle, then share with MPI

```
int daos_pool_local2global(daos_handle_t poh, d_iov_t *glob)
int daos_pool_global2local(daos_handle_t poh, d_iov_t *glob)
```

- Last, disconnect from your pool, finalize DAOS:

```
int daos_pool_disconnect(daos_handle_t poh, daos_event_t *ev);
int daos_fini(void);
```

Program Flow – Initialize DAOS, Connect to a Pool

- First (typically): initialize DAOS, connect to your pool:

```
int daos_init(void);
int daos_pool_connect(const char *pool, const char *sys,
      unsigned int flags, daos_handle_t *poh,
      daos_pool_info_t *info, daos_event_t *ev);
```

- MPI program: connect from 1 rank, serialize handle, then share with MPI

```
int daos_pool_local2global(daos_handle_t poh, d_iov_t *glob)
int daos_pool_global2local(daos_handle_t poh, d_iov_t *glob)
```

- Last, disconnect from your pool, finalize DAOS:

```
int daos_pool_disconnect(daos_handle_t poh, daos_event_t *ev);
int daos_fini(void);
```

const char *pool: label string

daos_handle_t: opaque handle type
- "poh" - "pool open handle"
- "coh" – "container open handle"
- "oh" – "object open handle"

daos_pool_info_t:
- capacity, free space, (im)balance
- Health, rebuild state
- Also output by `daos_pool_query()`

d_iov_t:
- Refers to a contiguous app buffer

daos_event_t: (not covered here)
- Asynchronous API invoke/test

Program Flow – Create a Container

- Using the daos tool:

```
daos cont create mypool mycont
```

```
    Container UUID : 5d33d6e0-6c8b-4bf5-bb49-c8723bf30c91
    Container Label: mycont
    Container Type : unknown

  Successfully created container 5d33d6e0-6c8b-4bf5-bb49-c8723bf30c91
```

User admin tool 'daos'

API:
- input poh from pool_connect
- output coh

Container also has a label id string

daos_prop_t: properties
- Label
- Type (POSIX, HDF5, untyped)
- Redundancy Factor (RF)

- Using the API:

```
int daos_cont_create_with_label(daos_handle_t poh, const char *label,
      daos_prop_t *cont_prop, uuid_t *uuid, daos_event_t *ev);

int daos_cont_destroy(daos_handle_t poh, const char *cont, int force, /* ev */);
```

Program Flow – Access a Container

- Need to open a container to access objects in it:

```
int daos_cont_open(daos_handle_t poh, const char *cont,
       unsigned int flags, daos_handle_t *coh, daos_cont_info_t *info, /* ev */);
```

- MPI program: connect from 1 rank, serialize handle, then share with MPI:

```
int daos_cont_local2global(daos_handle_t poh, d_iov_t *glob)
int daos_cont_global2local(daos_handle_t poh, d_iov_t *glob)
```

- Close container when done:

```
int daos_cont_close(daos_handle_t coh, daos_event_t *ev);
```

daos_cont_info_t:

- Pool UUID
- Container UUID
- # container open handles
- Metadata open/close/modify times
- RF
- …
- Also output by `daos_cont_query()`

Program Flow – Recap

```c
#include <daos.h>
int main(int argc, char **argv)
{
        daos_handle_t   poh, coh;

        daos_init();
        daos_pool_connect("mypool", NULL, DAOS_PC_RW, &poh, NULL, NULL);
        daos_cont_create_with_label(poh, "mycont", NULL, NULL, NULL);
        daos_cont_open(poh, "mycont", DAOS_COO_RW, &coh, NULL, NULL);

        /** perform object I/O - presented next */

        daos_cont_close(coh, NULL);
        daos_pool_disconnect(poh, NULL);
        daos_fini();
        return 0;
}
```

DAOS Object IDs – Types and Classes

- DAOS Object Types:
  - DAOS Flat KV – Each item having 1 string key, 1 opaque value
    - Operations: put, get, list, remove
    - Entire value collocated on 1 target, and atomic update
  - DAOS ARRAY – 1D array of fixed-size value
    - Operations: read, write, get/set size
  - DAOS Multi-Level KV – lower-level API
    - Operations: update, fetch, list
    - Multi-level keys (distribution / attribute)
    - Different value types (single value, array w/ fine-grain update)
- Object ID 128-bit space:
  - Lower 96 bits set by user
    - Unique OID allocator available in API for convenience
  - Upper 32 bits set by daos. OID Embeds:
    - Object type
    - Object class (redundancy level and type – Replication, EC, None)

DAOS Object IDs – Types and Classes

- DAOS Object Types:
  - DAOS Flat KV – Each item having 1 string key, 1 opaque value
    - Operations: put, get, list, remove
    - Entire value collocated on 1 target, and atomic update
  - DAOS ARRAY – 1D array of fixed-size value
    - Operations: read, write, get/set size
  - DAOS Multi-Level KV – lower-level API
    - Operations: update, fetch, list
    - Multi-level keys (distribution / attribute)
    - Different value types (single value, array w/ fine-grain update)
- Object ID 128-bit space:
  - Lower 96 bits set by user
    - Unique OID allocator available in API for convenience
  - Upper 32 bits set by daos. OID Embeds:
    - Object type
    - Object class (redundancy level and type – Replication, EC, None)

Sample Object Types (`enum daos_otype_t`)

```
/** flat KV (no akey) with hashed dkey */
    DAOS_OT_KV_HASHED,
/** Array, attributes provided by user */
    DAOS_OT_ARRAY_ATTR,
/** multi-level KV with uint64 [ad]keys */
    DAOS_OT_MULTI_UINT64,
```

DAOS Object IDs – Types and Classes

- DAOS Object Types:
  - DAOS Flat KV – Each item having 1 string key, 1 opaque value
    - Operations: put, get, list, remove
    - Entire value collocated on 1 target, and atomic update
  - DAOS ARRAY – 1D array of fixed-size value
    - Operations: read, write, get/set size
  - DAOS Multi-Level KV – lower-level API
    - Operations: update, fetch, list
    - Multi-level keys (distribution / attribute)
    - Different value types (single value, array w/ fine-grain update)
- Object ID 128-bit space:
  - Lower 96 bits set by user
    - Unique OID allocator available in API for convenience
  - Upper 32 bits set by daos. OID Embeds:
    - Object type
    - Object class (redundancy level and type – Replication, EC, None)

Sample Object Types (`enum daos_otype_t`)

```
/** flat KV (no akey) with hashed dkey */
    DAOS_OT_KV_HASHED,
/** Array, attributes provided by user */
    DAOS_OT_ARRAY_ATTR,
/** multi-level KV with uint64 [ad]keys */
    DAOS_OT_MULTI_UINT64,
```

Sample Object Classes (`daos_oclass_id_t`)

```
/* Explicit layout, no data protection
 * Examples: OC_S1, OC_S2, …, OC_S32, OC_SX
 * S1 : shards=1, S2 shards=2, SX shards=all tgts
 */

/* Replicated object (OC_RP_), explicit layout:
 * <number of replicas> G<redundancy groups>
 * Ex OC_RP_2G1, 2G2..32 2GX, 3G1..32 3GX, …
 * 2G1 : 2 replicas group=1
 * 3G2 : 3 replicas groups=2, ...
 * 6GX : 6 replicas, spread across all targets
 */

/* Erasure coded (OC_EC_), explicit layout:
 * <data_cells>P<parity_cells>G<redun_groups>
 * Ex: EC_8P2G1, EC_8P2G<2..32>, EC_8P2GX,
 *     EC_16P2G1, EC_16P2G<2..32>, EC_16P2GX,
 * - 8P2G2:   8+2 EC object, groups=2
 * - 16P2GX: 16+2 EC object, all targets in pool
 * - 2P1G1:   2+1 EC object, group=1
 * - 4P2G8:   4+2 EC object, groups=2
 */
```

DAOS Object IDs – Types and Classes

- Object ID 128-bit space (Lower 96 user; upper 32 daos):
  - Object type (e.g., KV, Array, Multi-Level KV)
  - Object class (Replication, EC, None)

```
int daos_obj_generate_oid(
        daos_handle_t coh,
        daos_obj_id_t *oid,
        enum daos_otype_t type,
        daos_oclass_id_t cid,
        daos_oclass_hints_t hints, uint32_t args);
```

```
daos_obj_id_t oid;
oid.hi = 0;
oid.lo = 1;
daos_obj_generate_oid(coh, &oid,
DAOS_OF_KV_HASHED, OC_RP_2GX, 0, 0);
```

Sample Object Types (`enum daos_otype_t`)

```
/** flat KV (no akey) with hashed dkey */
    DAOS_OT_KV_HASHED,
/** Array, attributes provided by user */
    DAOS_OT_ARRAY_ATTR,
/** multi-level KV with uint64 [ad]keys */
    DAOS_OT_MULTI_UINT64,
```

Sample Object Classes (`daos_oclass_id_t`)

```
/* Explicit layout, no data protection
 * Examples: OC_S1, OC_S2, …, OC_S32, OC_SX
 * S1 : shards=1, S2 shards=2, SX shards=all tgts
 */

/* Replicated object (OC_RP_), explicit layout:
 * <number of replicas> G<redundancy groups>
 * Ex OC_RP_2G1, 2G2..32 2GX, 3G1..32 3GX, …
 * 2G1 : 2 replicas group=1
 * 3G2 : 3 replicas groups=2, ...
 * 6GX : 6 replicas, spread across all targets
 */

/* Erasure coded (OC_EC_), explicit layout:
 * <data_cells>P<parity_cells>G<redun_groups>
 * Ex: EC_8P2G1, EC_8P2G<2..32>, EC_8P2GX,
 *     EC_16P2G1, EC_16P2G<2..32>, EC_16P2GX,
 * - 8P2G2:   8+2 EC object, groups=2
 * - 16P2GX: 16+2 EC object, all targets in pool
 * - 2P1G1:   2+1 EC object, group=1
 * - 4P2G8:   4+2 EC object, groups=2
 */
```

DAOS KV Object – Management Operations

- Recall: KV store interface providing access operations: Put, Get, Remove, List

- Management API:

```
int daos_kv_open(daos_handle_t coh, daos_obj_id_t oid,
       unsigned int mode, daos_handle_t *oh, daos_event_t *ev);

int daos_kv_close(daos_handle_t oh, daos_event_t *ev);

int daos_kv_destroy(daos_handle_t oh, daos_handle_t th, /* ev */);
```

KV: string key → opaque/atomic value

API:

- input coh from daos_cont_open()
- Input oid from daos_obj_generate_oid()
- output object handle (oh)

DAOS KV Object – Access Operations

- Access API:

```
int daos_kv_put(daos_handle_t oh, daos_handle_t th,
      uint64_t flags, const char *key,
      daos_size_t size, const void *buf, daos_event_t *ev);

int daos_kv_get(daos_handle_t oh, daos_handle_t th,
      uint64_t flags, const char *key,
      daos_size_t *size, void *buf, daos_event_t *ev);

int daos_kv_remove(daos_handle_t oh, daos_handle_t th,
      uint64_t flags, const char *key, daos_event_t *ev);

int daos_kv_list(daos_handle_t oh, daos_handle_t th, uint32_t *nr,
      daos_key_desc_t *kds, d_sg_list_t *sgl, daos_anchor_t *anchor, /* ev */);
```

API: input oh from kv_open()

Put/get/remove values (given string key)

List keys

- Key sizes in daos_key_desc_t *kds
- Key strings in sgl

DAOS KV Object – KV Conditional Operations

- By default, KV put/get operations do not check “existence” of key before operations:
  - Put(key): overwrites the value
  - Get(key): does not fail if key does not exist, just returns 0 size.
  - Remove(key): does not fail if key does not exist.

- One can use conditional flags for different behavior:
- `DAOS_COND_KEY_INSERT`: Insert a key if it doesn't exist (fail if it already exists)
- `DAOS_COND_KEY_UPDATE`: Update a key if it exists, (fail if it does not exist)
- `DAOS_COND_KEY_GET`: Get key value if it exists, (fail if it does not exist).
- `DAOS_COND_KEY_REMOVE`: Remove a key if it exists (fail if it does not exist).

DAOS KV Object – Put/Get Example

```
/** daos_init, daos_pool_connect, daos_cont_open */

oid.hi = 0;
oid.lo = 1;
daos_obj_generate_oid(coh, &oid, DAOS_OF_KV_HASHED, OC_RP_2GX, 0, 0);
daos_kv_open(coh, oid, DAOS_OO_RW, &kv, NULL);

/** set val buffer and size */
daos_kv_put(kv, DAOS_TX_NONE, 0, "key1", val_len1, val_buf1, NULL);
daos_kv_put(kv, DAOS_TX_NONE, 0, "key2", val_len2, val_buf2, NULL);

/** to fetch, can query the size first if not known */
daos_kv_get(kv, DAOS_TX_NONE, 0, "key1", &size, NULL, NULL);
get_buf = malloc (size);
daos_kv_get(kv, DAOS_TX_NONE, 0, "key1", &size, get_buf, NULL);
daos_kv_close(kv, NULL);

/** free buffer, daos_cont_close, daos_pool_disconnect, daos_fini */
```

DAOS KV Object – List Keys Example

```
/** enumerate keys in the KV */
daos_anchor_t   anchor = {0};
d_sg_list_t     sgl;
d_iov_t         sg_iov;

/** size of buffer to hold as many keys in memory */
buf = malloc(ENUM_DESC_BUF_BYTES);
d_iov_set(&sg_iov, buf, ENUM_DESC_BUF_BYTES);
sgl.sg_nr               = 1;
sgl.sg_nr_out           = 0;
sgl.sg_iovs             = &sg_iov;
```

```
daos_key_desc_t kds[ENUM_DESC_NR];

while (!daos_anchor_is_eof(&anchor)) {
  /** how many keys to attempt to fetch in one call */
  uint32_t        nr = ENUM_DESC_NR;


  memset(buf, 0, ENUM_DESC_BUF_BYTES);
  daos_kv_list(kv, DAOS_TX_NONE, &nr, kds, &sgl,
               &anchor, NULL);

  if (nr == 0)
    continue;
  /** buf now contains nr keys */
  /** kds[] has nr key descriptors (length keys) */
}
```

DAOS Array Object – Management Operations

- 1-Dimensional Array object to manage records
  - cell_size: single array value size (bytes)
  - chunk_size: number of cells placed together in a storage target –controls striping of array regions across storage cluster

- Management API:

```
int daos_array_create(daos_handle_t coh, daos_obj_id_t oid, daos_handle_t th,
      daos_size_t cell_size, daos_size_t chunk_size, daos_handle_t *oh, /* ev */);
int daos_array_open(daos_handle_t coh, daos_obj_id_t oid, daos_handle_t th,
      unsigned int mode, daos_size_t *cell_size,
      daos_size_t *chunk_size,daos_handle_t *oh, daos_event_t *ev);

int daos_array_close(daos_handle_t oh, daos_event_t *ev);

int daos_array_destroy(daos_handle_t oh, daos_handle_t th, daos_event_t *ev);
```

DAOS Array Object – Access Operations

- Reading & writing record to an Array:

```
int daos_array_read(daos_handle_t oh, daos_handle_t th, daos_array_iod_t *iod,
      d_sg_list_t *sgl, daos_event_t *ev);
int daos_array_write(daos_handle_t oh, daos_handle_t th, daos_array_iod_t *iod,
      d_sg_list_t *sgl, daos_event_t *ev);
```

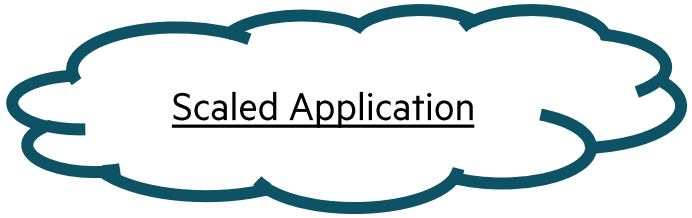
- Misc

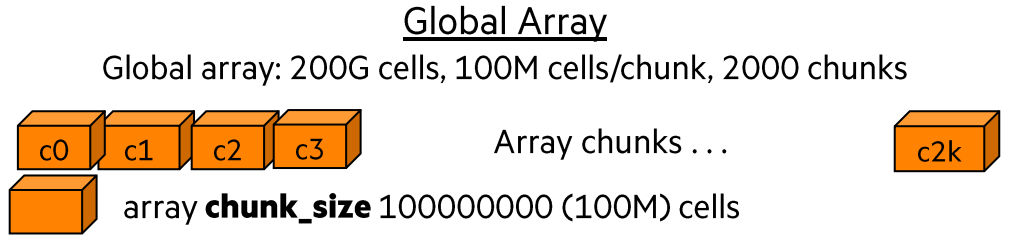
```
int daos_array_get_size(daos_handle_t oh, daos_handle_t th, daos_size_t *size, …);
int daos_array_set_size(daos_handle_t oh, daos_handle_t th, daos_size_t size, …);
int daos_array_get_attr(daos_handle_t oh, daos_size_t *chunk_size,
                        daos_size_t *cell_size);
```

DAOS Array Object – Example

```
/** create array - 1 byte/cell, NCELLS=100 million cells per chunk */
daos_array_create(coh, oid, DAOS_TX_NONE, 1, 100000000, &array, NULL);
```

`

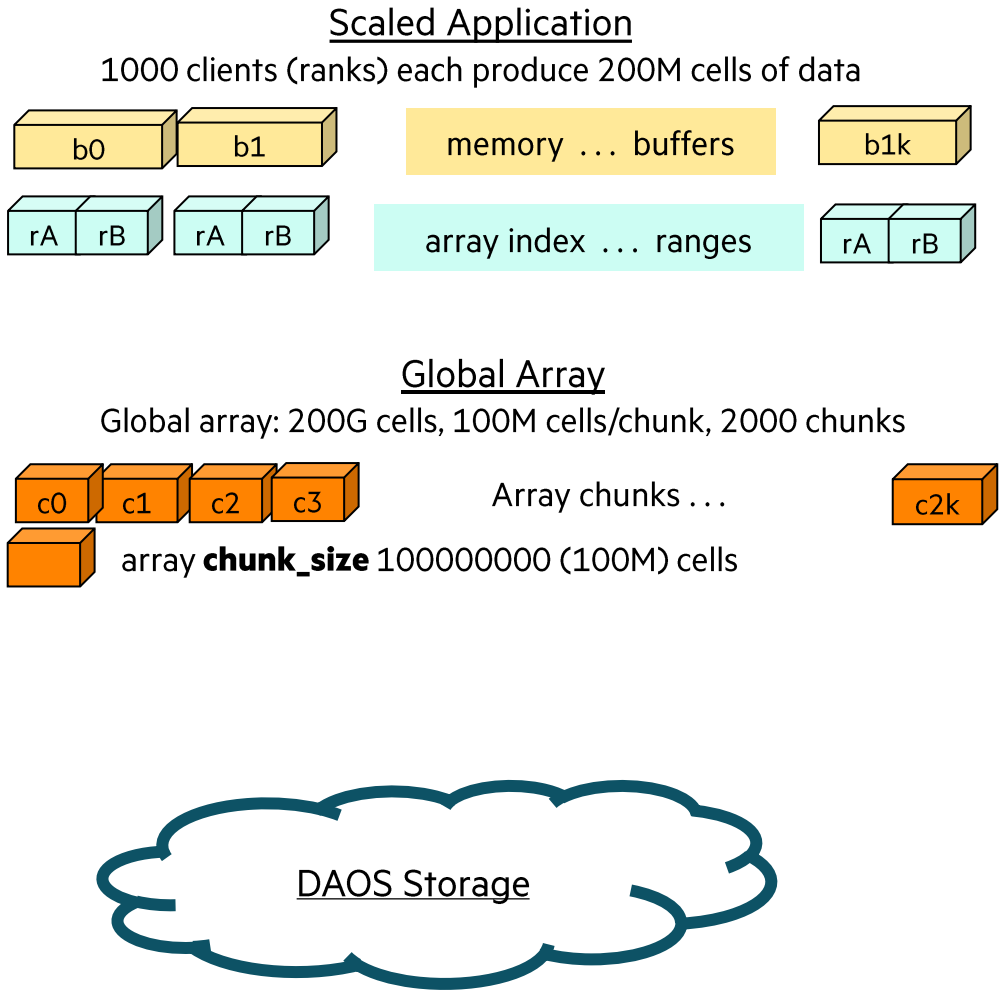




DAOS Array Object – Example

```
/** create array - 1 byte/cell, NCELLS=100 million cells per chunk */
daos_array_create(coh, oid, DAOS_TX_NONE, 1, 100000000, &array, NULL);

d_sg_list_t      sgl;    /* memory: scatter/gather list of iovecs */
d_iov_t          iov;    /* memory (iovec): 1 buffer (ptr, bytes) */
daos_array_iod_t iod;    /* array IO descriptor - array ranges */
daos_range_t     rgs[2]; /* array ranges(start index, num cells) */

/** set memory location, each rank writing BUFLEN */
sgl.sg_nr = 1;                  /** one memory buffer iovec */
d_iov_set(&iov, buf, BUFLEN); /** one buffer/ptr, BUFLEN=200M bytes */
sgl.sg_iovs = &iov;

/** specify this client's particular array (sub)ranges */
iod.arr_nr    = 2;                /** two array (sub)ranges */
iod.arr_rgs   = rgs;              /** the list of two ranges */
ra_start      = rank * NCELLS*2;  /** array ranges start indices */
rb_start      = ra_start + NCELLS;
rgs[0].rg_idx = ra_start;         /** (and rgs[1] from rb_start) */
rgs[0].rg_len = NCELLS;           /** length (and rgs[1] len=NCELLS) */
```

`

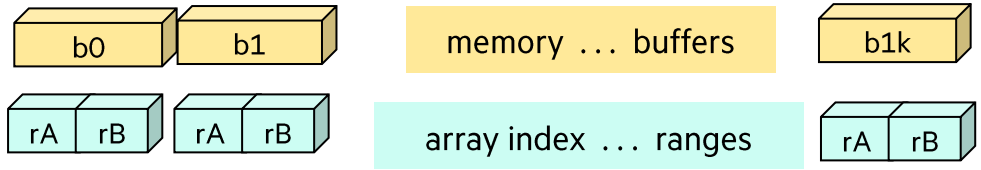


DAOS Array Object – Example

```
/** create array - 1 byte/cell, NCELLS=100,000,000 cells per chunk */
daos_array_create(coh, oid, DAOS_TX_NONE, 1, NCELLS, &array, NULL);

d_sg_list_t      sgl;    /* memory: scatter/gather list of iovecs */
d_iov_t          iov;    /* memory (iovec): 1 buffer (ptr, bytes) */
daos_array_iod_t iod;    /* array IO descriptor - array ranges */
daos_range_t     rgs[2]; /* array ranges(start index, num cells) */

/** set memory location, each rank writing BUFLEN */
sgl.sg_nr = 1;                  /** one memory buffer iovec */
d_iov_set(&iov, buf, BUFLEN); /** one buffer/ptr, BUFLEN=200M bytes */
sgl.sg_iovs = &iov;

/** specify this client's particular array (sub)ranges */
iod.arr_nr    = 2;                /** two array (sub)ranges */
iod.arr_rgs   = rgs;              /** the list of two ranges */
ra_start      = rank * NCELLS*2;  /** array ranges start indices */
rb_start      = ra_start + NCELLS;
rgs[0].rg_idx = ra_start;         /** (and rgs[1] from rb_start) */
rgs[0].rg_len = NCELLS;           /** length (and rgs[1] len=NCELLS) */

/** write array data to DAOS storage, and read back */
daos_array_write(array, DAOS_TX_NONE, &iod, &sgl, NULL);
daos_array_read(array, DAOS_TX_NONE, &iod, &sgl, NULL);
daos_array_close(array, NULL);
```

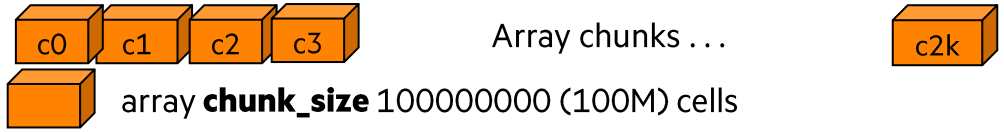
Scaled Application

1000 clients (ranks) each produce 200M cells of data



Global Array

Global array: 200G cells, 100M cells/chunk, 2000 chunks

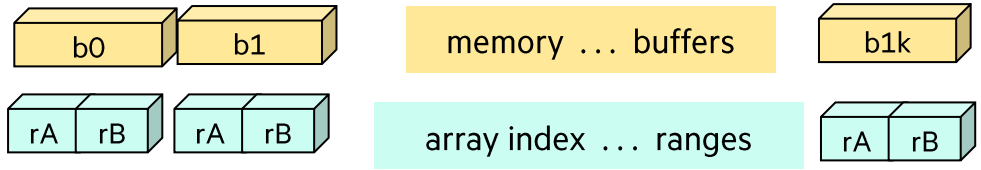


DAOS Array Object – Example

```
/** create array - 1 byte/cell, NCELLS=100 million cells per chunk */
daos_array_create(coh, oid, DAOS_TX_NONE, 1, 100000000, &array, NULL);

d_sg_list_t      sgl;    /* memory: scatter/gather list of iovecs */
d_iov_t          iov;    /* memory (iovec): 1 buffer (ptr, bytes) */
daos_array_iod_t iod;    /* array IO descriptor - array ranges */
daos_range_t     rgs[2]; /* array ranges(start index, num cells) */

/** set memory location, each rank writing BUFLEN */
sgl.sg_nr = 1;                  /** one memory buffer iovec */
d_iov_set(&iov, buf, BUFLEN); /** one buffer/ptr, BUFLEN=200M bytes */
sgl.sg_iovs = &iov;

/** specify this client's particular array (sub)ranges */
iod.arr_nr    = 2;                /** two array (sub)ranges */
iod.arr_rgs   = rgs;              /** the list of two ranges */
ra_start      = rank * NCELLS*2;  /** array ranges start indices */
rb_start      = ra_start + NCELLS;
rgs[0].rg_idx = ra_start;         /** (and rgs[1] from rb_start) */
rgs[0].rg_len = NCELLS;           /** length (and rgs[1] len=NCELLS) */

/** write array data to DAOS storage, and read back */
daos_array_write(array, DAOS_TX_NONE, &iod, &sgl, NULL);
daos_array_read(array, DAOS_TX_NONE, &iod, &sgl, NULL);
daos_array_close(array, NULL);
```

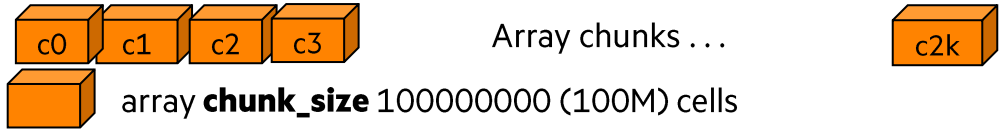
Scaled Application

1000 clients (ranks) each produce 200M cells of data



Global Array

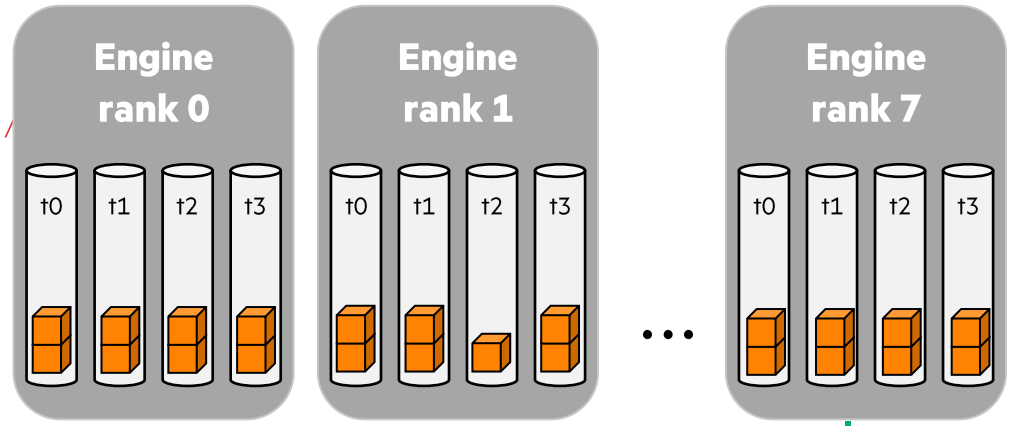
Global array: 200G cells, 100M cells/chunk, 2000 chunks



DAOS Storage

Stored in <num_daos_engines_in_pool> x tgts/daos_server

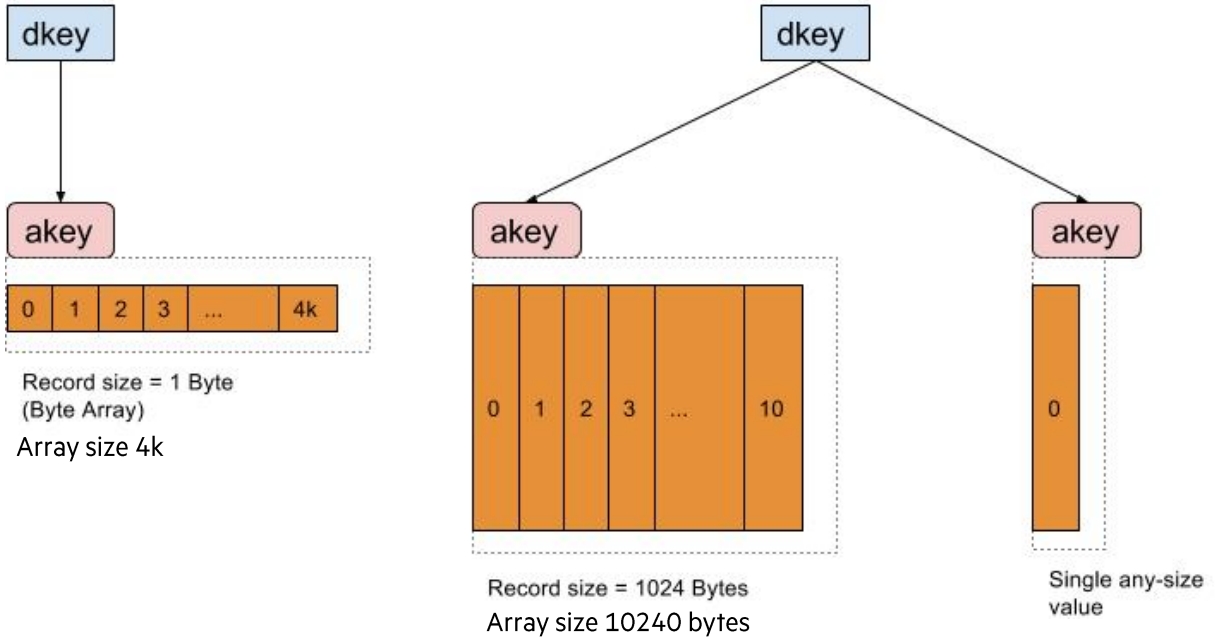
Ex: 8 servers x 4 tgts/server = 32 targets



Multi-Level KV Object

- Two-level key:
  - Distribution Key - Dkey (collocate all entries under it), holds multiple akeys
  - Attribute Key - Akey (lower level to address records)
  - Both are opaque (support any size / type)

- Value types (under a single akey):
  - Single value: one blob (traditional value in KV store)
  - Array value:
    - Array of fixed-size cells (records) that can be updated in a fine-grained manner via different range extents
    - This is different than a DAOS (global/distributed) array



- **Intentionally very flexible, rich API**
- **(at the expense of higher complexity for the typical user)**

Multi-Level KV Object – Management Operations

```
int daos_obj_open(daos_handle_t coh, daos_obj_id_t oid, unsigned int mode,
                  daos_handle_t *oh, daos_event_t *ev);
int daos_obj_close(daos_handle_t oh, daos_event_t *ev);

int daos_obj_punch(daos_handle_t oh, daos_handle_t th, uint64_t flags, /* ev */);
int daos_obj_punch_dkeys(daos_handle_t oh, daos_handle_t th, uint64_t flags,
                         unsigned int nr, daos_key_t *dkeys, daos_event_t *ev);
int daos_obj_punch_akeys(daos_handle_t oh, daos_handle_t th, uint64_t flags,
                         daos_key_t *dkey, unsigned int nr, daos_key_t *akeys, …);
```

API:

- input coh from daos_cont_open()
- Input oid from daos_obj_generate_oid()
- output object handle (oh)

Multi-Level KV Object – Access Operations (Update, Fetch)

```
int daos_obj_update(daos_handle_t oh, daos_handle_t th,
      uint64_t flags, daos_key_t *dkey, unsigned int nr,
      daos_iod_t *iods, d_sg_list_t *sgls, daos_event_t *ev);
```

```
daos_key_t iod_name;      /* akey */
daos_iod_type_t iod_type; /* value type (single value or array value) */
daos_size_t iod_size;     /* SV: value size, array: record size */
uint32_t iod_nr;          /* SV: 1, array: number of record extents */
daos_recx_t *iod_recxs;   /* SV: NULL, array: (offset, length) pairs */
      uint64_t rx_idx, rx_nr;
```

```
uint32_t sg_nr;
uint32_t sg_nr_out;
d_iov_t *sg_iovs;
```

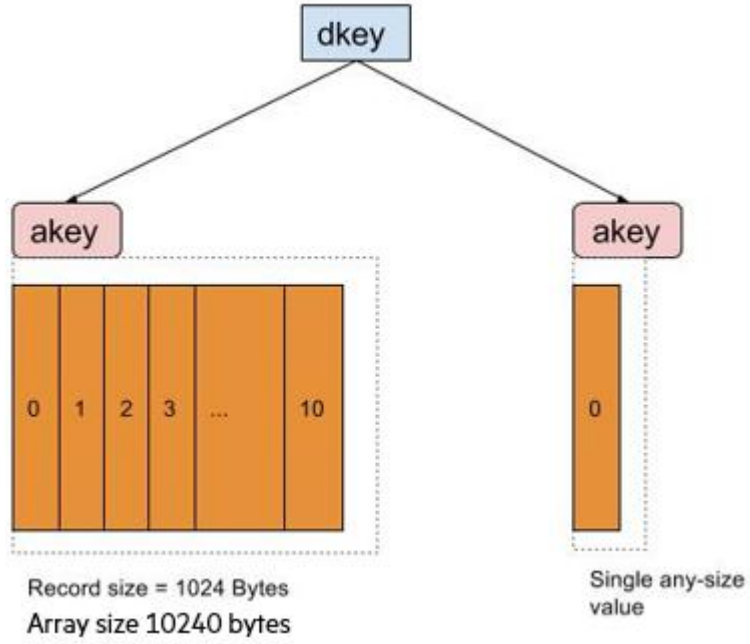
```
int daos_obj_fetch(daos_handle_t oh, daos_handle_t th, uint64_t flags,
      daos_key_t *dkey, unsigned int nr, daos_iod_t *iods,
      d_sg_list_t *sgls, daos_iom_t *ioms, daos_event_t *ev);
```

Multi-Level KV Object – Access Operations (List)

```
int daos_obj_list_dkey(daos_handle_t oh, daos_handle_t th, uint32_t *nr,
      daos_key_desc_t *kds, d_sg_list_t *sgl, daos_anchor_t *anchor,…);

int daos_obj_list_akey(daos_handle_t oh, daos_handle_t th,
      daos_key_t *dkey, uint32_t *nr, daos_key_desc_t *kds,
      d_sg_list_t *sgl, daos_anchor_t *anchor, daos_event_t *ev);
```

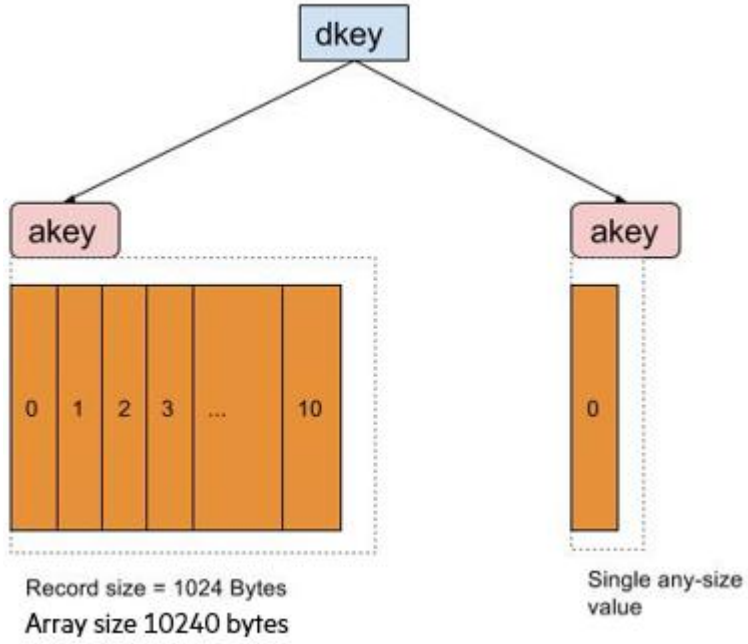
Multi-Level KV Object – Update Example

```
daos_obj_open(coh, oid, DAOS_OO_RW, &oh, NULL);

/* application buffers */
const char *buf2 = "single_value, my string";
d_iov_set(&sg_iovs[0], buf1, BUF1LEN);      /* 10240 byte array val: (dkey1,akey1) */
sgls[0].sg_nr = 1;
sgls[0].sg_iovs = &sg_iovs[0];
d_iov_set(&sg_iovs[1], buf2, strlen(buf2)); /* string val: (dkey1,akey2) */
sgls[1].sg_nr = 1;
sgls[1].sg_iovs = &sg_iovs[1];

/* keys */
d_iov_set(&dkey, "dkey1", strlen("dkey1"));
d_iov_set(&iods[0].iod_name, "akey1", strlen("akey1"));
d_iov_set(&iods[1].iod_name, "akey2", strlen("akey2"));

/* IODs for each akey */
iods[0].iod_type = DAOS_IOD_ARRAY;
iods[0].iod_size = 1;                   /* 1 byte/array cell */
recx.rx_idx = 0;                        /* array index range (0, BUF1LEN) */
recx.rx_nr = BUF1LEN;
iods[0].iod_nr = 1;
iods[0].iod_recxs = &recx;
iods[1].iod_type = DAOS_IOD_SINGLE;
iods[1].iod_size = strlen(buf2);
iods[1].iod_nr = 1;                     /* iod_recxs=NULL for SV */

daos_obj_update(oh, DAOS_TX_NONE, 0, &dkey, 2, &iods[0], &sgls[0], NULL);
```



Multi-Level KV Object – Fetch Example

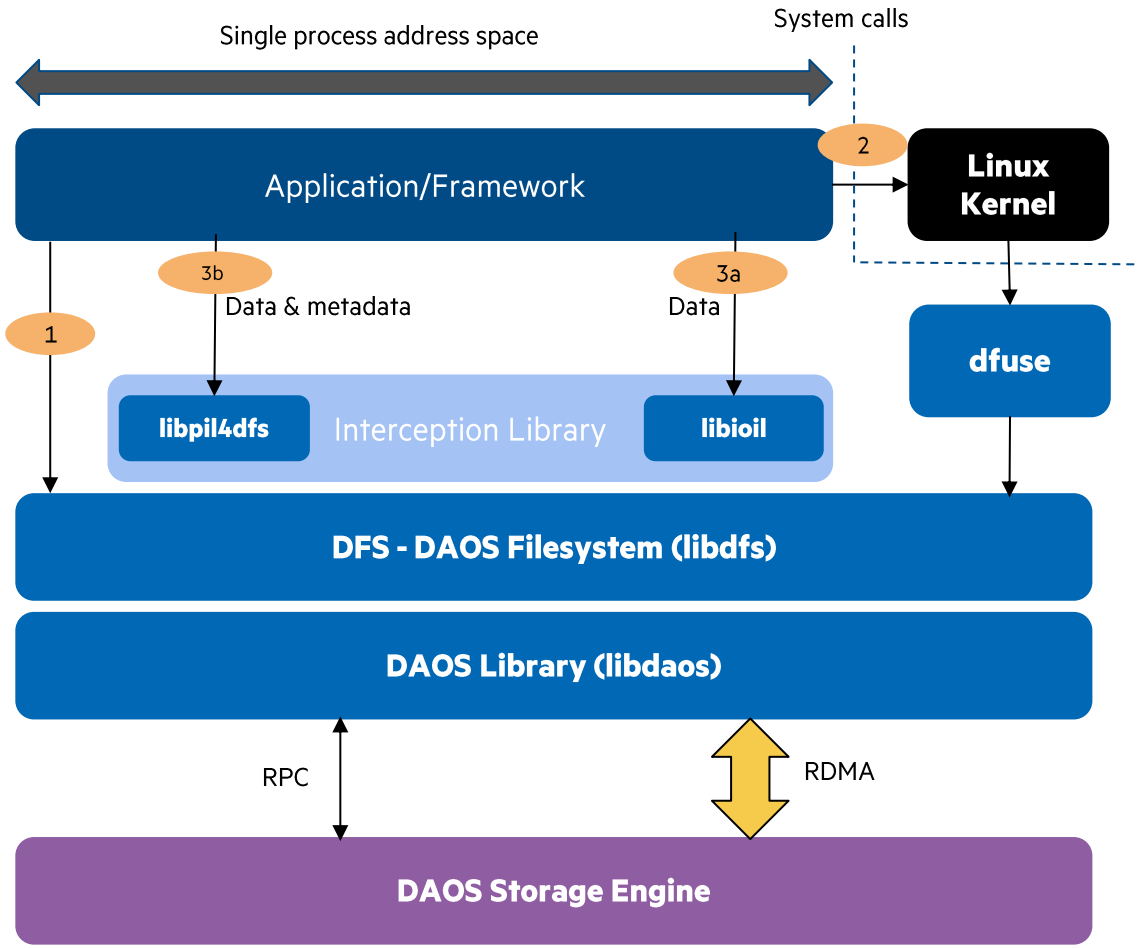
```
daos_obj_open(coh, oid, DAOS_OO_RW, &oh, NULL);

/* application buffers */
char rbuf2[128];
d_iov_set(&sg_iovs[0], rbuf1, BUF1LEN);     /* 10240 byte array val: (dkey1,akey1) */
sgls[0].sg_nr = 1;
sgls[0].sg_iovs = &sg_iovs[0];
d_iov_set(&sg_iovs[1], rbuf2, strlen(buf2)); /* string val: (dkey1,akey2) */
sgls[1].sg_nr = 1;
sgls[1].sg_iovs = &sg_iovs[1];

/* keys */
d_iov_set(&dkey, "dkey1", strlen("dkey1"));
d_iov_set(&iods[0].iod_name, "akey1", strlen("akey1"));
d_iov_set(&iods[1].iod_name, "akey2", strlen("akey2"));

/* IODs for each akey */
/** If iod_size is unknown: specify DAOS_REC_ANY, NULL sgl */
iods[0].iod_type = DAOS_IOD_ARRAY;
iods[0].iod_size = 1;                  /* 1 byte/array cell */
recx.rx_idx = 0;                       /* array index range (0, BUF1LEN) */
recx.rx_nr = BUF1LEN;
iods[0].iod_nr = 1;
iods[0].iod_recxs = &recx;
iods[1].iod_type = DAOS_IOD_SINGLE;
iods[1].iod_size = strlen(buf2);
iods[1].iod_nr = 1;                    /* iod_recxs=NULL for SV */

daos_obj_fetch(oh, DAOS_TX_NONE, 0, &dkey, 2, &iods[0], &sgls[0], NULL, NULL);
```



More Examples

- https://github.com/daos-stack/daos/blob/master/src/tests/simple_obj.c
- https://github.com/daos-stack/daos/blob/master/src/tests/simple_dfs.c

POSIX Support & Interception



1

Userspace DFS library with API like POSIX

- **Requires** application changes
- Low-latency and high-concurrency
- No caching

2

DFUSE daemon to support POSIX API

- **No changes** to application
- VFS mount point and high-latency
- Caching done by Linux kernel

3

DFUSE + Interception Library

- **No changes** to application
- 2 flavors, choose with LD_PRELOAD

3a

- Libioil
  - Intercept data only - (f)read/write
  - Metadata handled by dfuse / kernel

3b

- Libpil4dfs
  - Intercept data and metadata
  - Aim to deliver same performance as DFS (#1)
  - mmap() and binary execution via fuse

POSIX – How to Use DFS API (From a Modified Application)

- You should have access to a pool (identified by a string label).

- Create a POSIX container with the daos tool:
  - `daos cont create mypool mycont --type=POSIX`
  - Or: use API to create a container to use in your application (if using DFS and changing your app).

- Open the DFS mount:
  - `dfs_connect (mypool, mycont, O_RDWR, .. &dfs);`
  - `dfs_disconnect (dfs);`

POSIX – DFS API

POSIX	DFS
mkdir(), rmdir()	dfs_mkdir(), dfs_rmdir()
open(), close(), access()	dfs_open(), dfs_release(),dfs_lookup()
pwritev(), preadv()	dfs_read/write()
{set,get,list,remove}xattr()	dfs_{set,get,list,remove}xattr
stat(), fstat()	dfs_stat(),ostat()
readdir() …	dfs_readdir() …

- Mostly 1-1 mapping from POSIX API to DFS API.
- Instead of File & Directory descriptors, use DFS objects.
- All calls need the DFS mount which is usually done once (initialization time).

POSIX – DFUSE (With Unmodified Applications)

- To mount an existing POSIX container with dfuse, run the following command:
  - `dfuse mypool mycont -m /mnt/dfuse`
  - No one can access your container / mountpoint unless access is provided on the pool and container (via ACLs)

- Now you have a parallel file system under /mnt/dfuse on all nodes where that is mounted
  - "Easy path" for unmodified apps and daos – access files / directories as a namespace in the container

- dfuse + Interception Libraries:
  - Approach: intercept POSIX I/O calls, issue I/O directly from application through libdaos (kernel bypass)
  - To use: set LD_PRELOAD to point to the shared library in the DAOS install dir
    - (newer approach – metadata+data intercept) LD_PRELOAD=/path/to/daos/install/lib64/pil4dfs.so
    - (original approach – read/write only intercept) LD_PRELOAD=/path/to/daos/install/lib64/libioil.so

POSIX – Best Practices: Redundancy Factor (rd_fac) Container Property

- The number of (not yet rebuilt) concurrent failures container objects are protected against (without loss)
  - A number in the range 0-5

- Production systems recommendation: rd_fac:2 (which is a default value if not specified)
  - `daos cont create –type=POSIX –properties=rd_fac:2 <pool> <container>`

- Note: all objects must use a class with at least this degree of protection. Some legal examples:

	rd_fac:0	rd_fac:1	rd_fac:2
No Protection Classes	OC_S<*>	None	None
Replication Classes	Any	OC_RP_2G<*> OCP_RP_3G<*> …	OC_RP_3G<*> OC_RP_4G<*> …
Erasure Code Classes	Any	OC_EC_8P1G<*> OC_EC_16P1G<*>	OC_EC_8P2G<*> OC_EC_16P2G<*>

POSIX – Best Practices: Object Class Data Protection

- Recall: data protection is part of an object's "object class" – None, Replication, or Erasure Code

- Erasure Code:
  - Best for large IO access patterns.
  - Full stripe write: 12%-33% lower performance (vs. no data protection).
  - Partial stripe write: 66% lower performance (vs. no data protection).
  - Read performance should be the same.
  - Not supported for directory objects

- Replication:
  - Best for metadata objects (directories) and small files (<= 16k).
  - Write IOPS: slower (than no data protection) by the number of replicas created.
  - Read IOPS: equal or better (than no data protection) – more shards to serve concurrent requests.

POSIX – Best Practices: Object Class Striping (Wide or Narrow)

`daos cont create –type=POSIX –dir-oclass=<OC> --file-oclass=<OC>`

	rd_fac:0	rd_fac:1	rd_fac:2
Defaults - Widely-striped (“X”) objs for: - Large files (GBs), Lean dirs. (<10k ent) - Single-shared access, high BW required	File: SX Dir : S1	File: EC_16P1GX Dir : RP_2G1	File: EC_16P2GX Dir : RP_3G1
Small-stripe (1/2/4/16/32) objs for: - Something in-between huge and tiny files - File per process to large files.	File: S32 (S1/2/…/32) Dir : S1	File: EC_16P1_G32 (G1/2/…/32) Dir : RP_2G1	File: EC_16P2_G32 (G1/2/…/32) Dir : RP_3G1
One-stripe objs for: - tiny files, more IOPS required	File/Dir: S1	File/Dir: RP_2G1	File/Dir: RP_3G1

Recall: Sample Object Classes (`daos_oclass_id_t`)

```
/* Explicit layout, no data protection
 * Examples: OC_S1, OC_S2, …, OC_S32, OC_SX
 * S1 : shards=1, S2 shards=2, SX shards=all tgts
 */

/* Replicated object (OC_RP_), explicit layout:
 * <number of replicas> G<redundancy groups>
 * Ex OC_RP_2G1, 2G2..32 2GX, 3G1..32 3GX, …
 * 2G1 : 2 replicas group=1
 * 3G2 : 3 replicas groups=2, ...
 * 6GX : 6 replicas, spread across all targets
 */

/* Erasure coded (OC_EC_), explicit layout:
 * <data_cells>P<parity_cells>G<redun_groups>
 * Ex: EC_8P2G1, EC_8P2G<2..32>, EC_8P2GX,
 *     EC_16P2G1, EC_16P2G<2..32>, EC_16P2GX,
 * - 8P2G2:   8+2 EC object, groups=2
 * - 16P2GX: 16+2 EC object, all targets in pool
 * - 2P1G1:   2+1 EC object, group=1
 * - 4P2G8:   4+2 EC object, groups=2
 */
```

POSIX – Best Practices: Object Class Striping – Tradeoffs

```
daos cont create –type=POSIX –dir-oclass=<OC> --file-oclass=<OC>
```

	rd_fac:0	rd_fac:1	rd_fac:2
Defaults - Widely-striped ("X") objs for: - Large files (GBs), Lean dirs. (<10k ent) - Single-shared access, high BW required Tradeoffs: - If used with file-per-proc, non-scalable pool connect slows pool service. - Slow file stat(), remove, directory listing – RPC to all engines, query all targets.	File: SX Dir : S1	File: EC_16P1GX Dir : RP_2G1	File: EC_16P2GX Dir : RP_3G1
Small-stripe (1/2/4/16/32) objs for: - Something in-between huge / tiny files - File per process to large files. Tradeoffs: - Faster stat() and directory listing - Limited bandwidth to number of targets - Benchmarking file create/remove/stat could benefit from widely-striped dirs.	File: S32 (S1/2/…/32) Dir : S1	File: EC_16P1_G32 (G1/2/…/32) Dir : RP_2G1	File: EC_16P2_G32 (G1/2/…/32) Dir : RP_3G1
One-stripe objs for: - tiny files, more IOPS required	File/Dir: S1	File/Dir: RP_2G1	File/Dir: RP_3G1

Recall: Sample Object Classes (`daos_oclass_id_t`)

```
/* Explicit layout, no data protection
 * Examples: OC_S1, OC_S2, …, OC_S32, OC_SX
 * S1 : shards=1, S2 shards=2, SX shards=all tgts
 */

/* Replicated object (OC_RP_), explicit layout:
 * <number of replicas> G<redundancy groups>
 * Ex OC_RP_2G1, 2G2..32 2GX, 3G1..32 3GX, …
 * 2G1 : 2 replicas group=1
 * 3G2 : 3 replicas groups=2, ...
 * 6GX : 6 replicas, spread across all targets
 */

/* Erasure coded (OC_EC_), explicit layout:
 * <data_cells>P<parity_cells>G<redun_groups>
 * Ex: EC_8P2G1, EC_8P2G<2..32>, EC_8P2GX,
 *     EC_16P2G1, EC_16P2G<2..32>, EC_16P2GX,
 * - 8P2G2:   8+2 EC object, groups=2
 * - 16P2GX: 16+2 EC object, all targets in pool
 * - 2P1G1:   2+1 EC object, group=1
 * - 4P2G8:   4+2 EC object, groups=2
 */
```

POSIX – Best Practices: – EC Properties for Performance

- DFS Chunk Size, default 1MiB (`daos container create –chunk_size=`)
  - DAOS splits file data across dkeys in chunk size units

- `ec_cell_sz` container property
  - DAOS splits application buffer into `ec_cell_sz` byte parts (16 parts for EC_16P2, 8 parts for EC_8P2, etc.)

- Full (vs. Partial) Stripe Write – application buffer chunk is an even multiple of `ec_cell_sz` (or not).
  - Full stripe write is more efficient

EC Object Class	EC Cell Size	DFS Chunk size	Full or Partial Write?
16P2	128k	1m	Partial: 1m gets divided into 8 128k data parts which < 16 data shards of the object class used
8P2	128k	1m	Full
16P2	128k	2m, 4m, 8m, etc	Full
16P2	256k	2m	Partial
16P2	256k	4m, 8m, etc.	Full

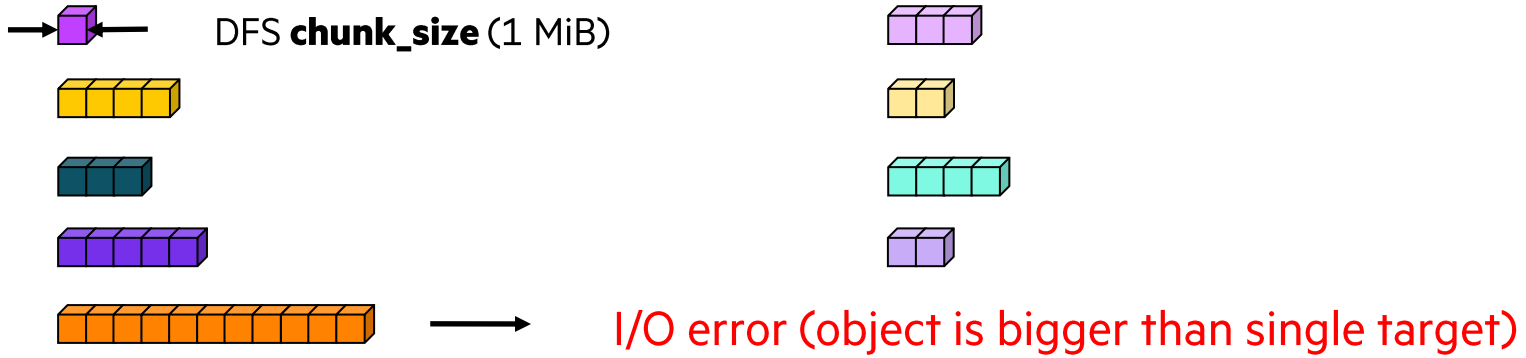
Thank you

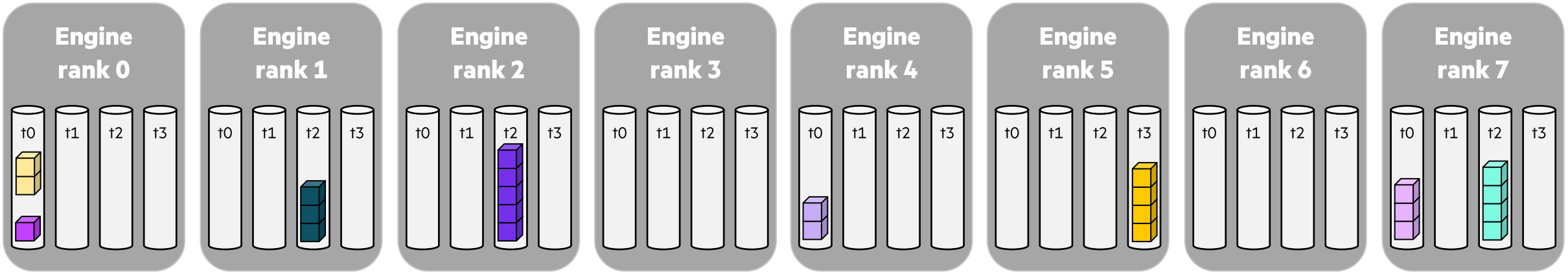
kenneth.cain@hpe.com
a.jackson@epcc.ed.ac.uk

mohamad.chaarawi@hpe.com
jerome.soumagne@hpe.com

Data Distribution and Data Protection

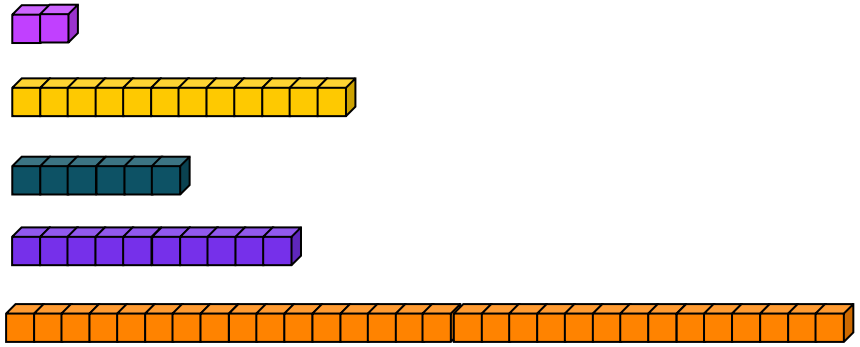
Sharding, object class S1

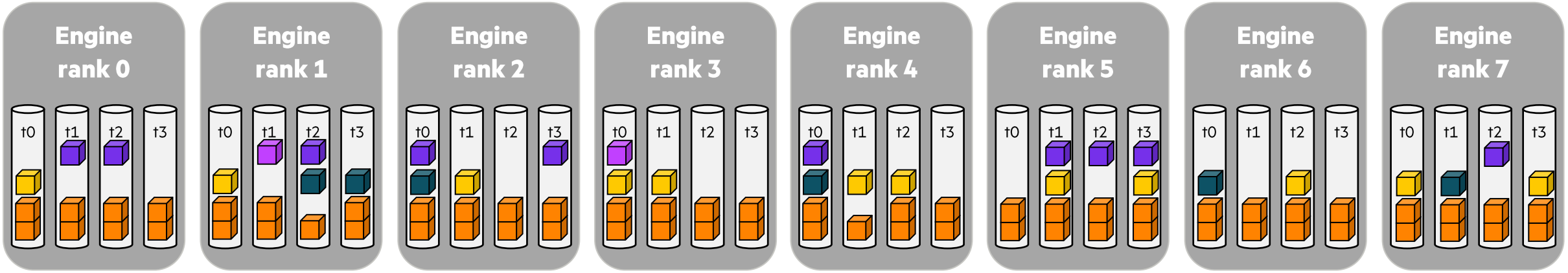




Data Distribution and Data Protection

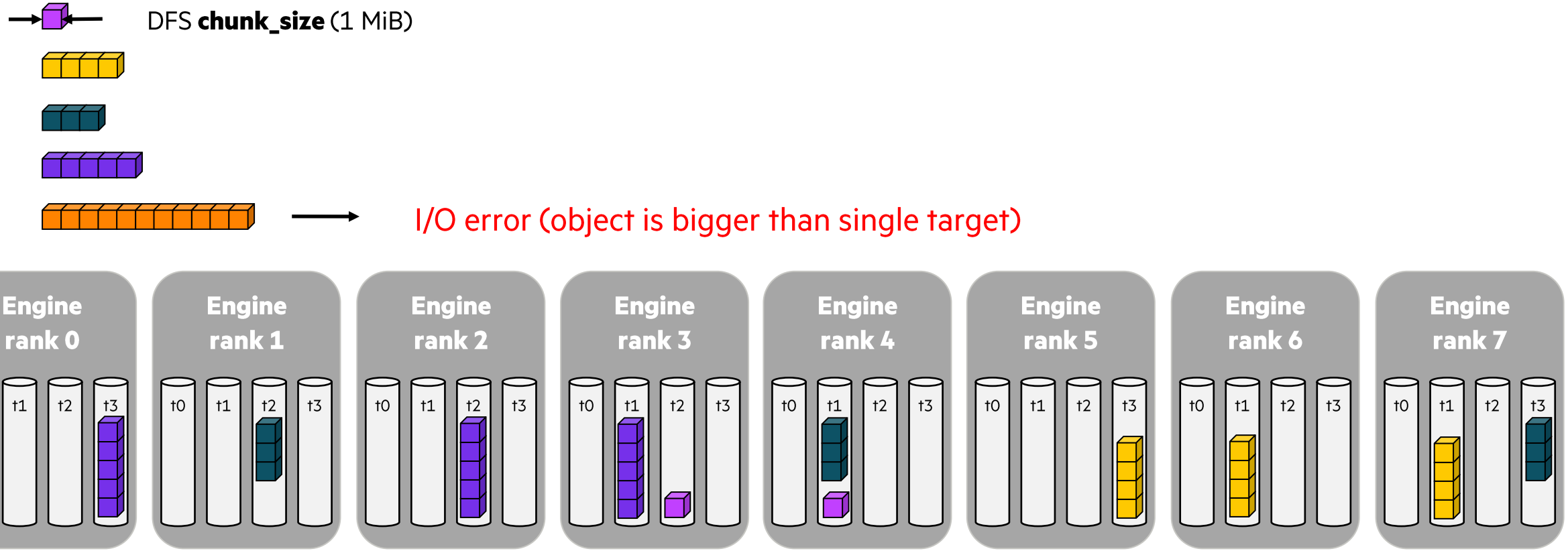
Sharding, object class SX





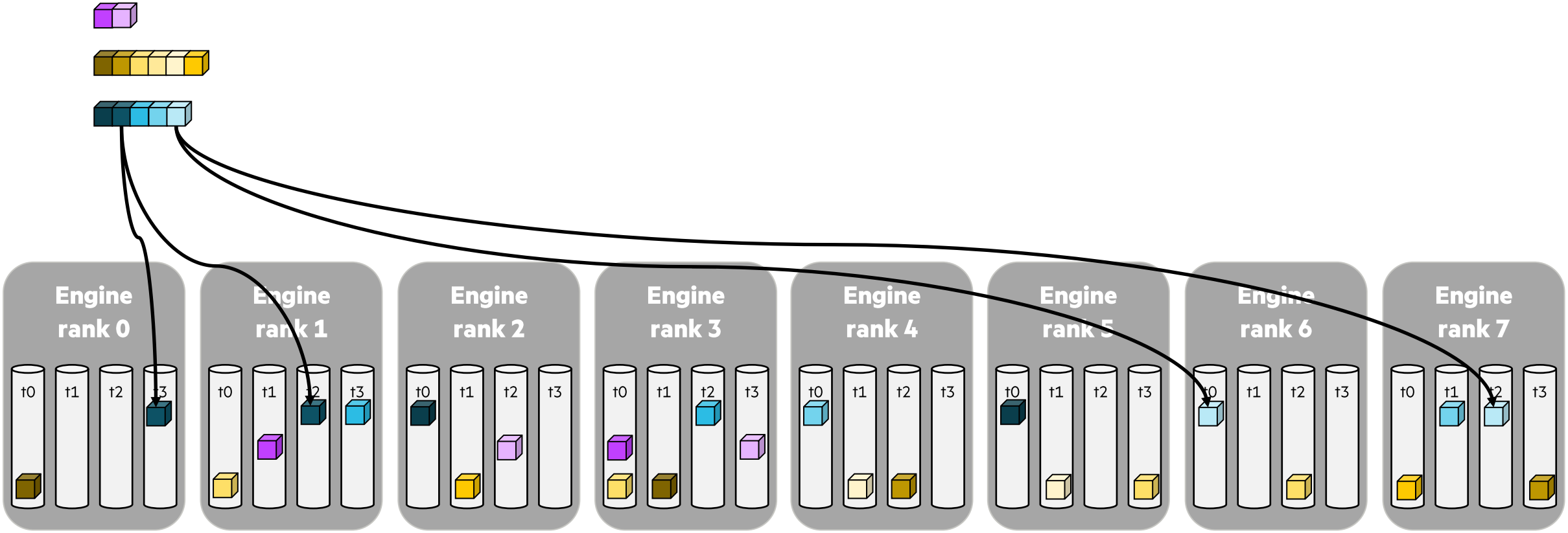
Data Distribution and Data Protection

Replication, object class RP_3G1 (rf_lvl=engine)



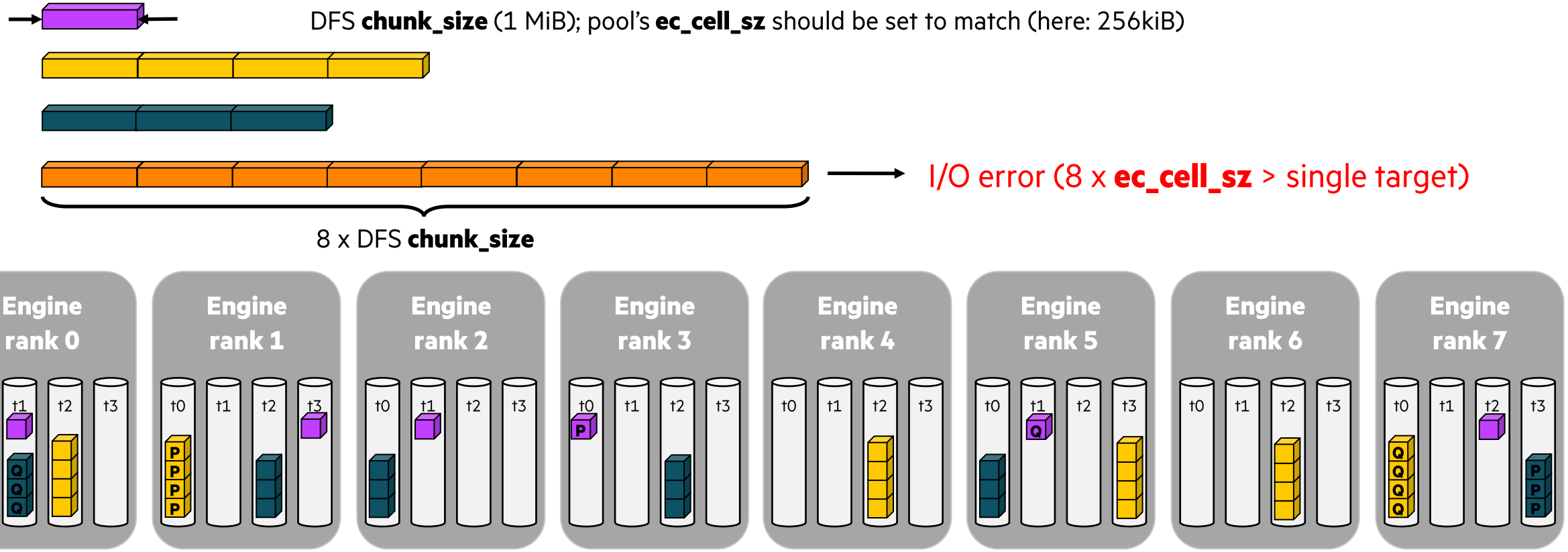
Data Distribution and Data Protection

Replication, object class RP_2GX (rf_lvl=engine)



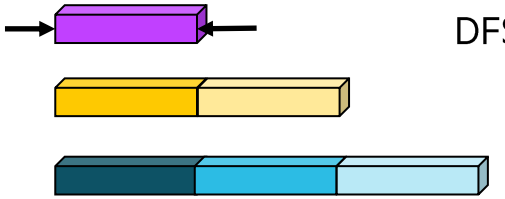
Data Distribution and Data Protection

Erasure Coding, object class EC_4P2G1 (rf_lvl=engine)

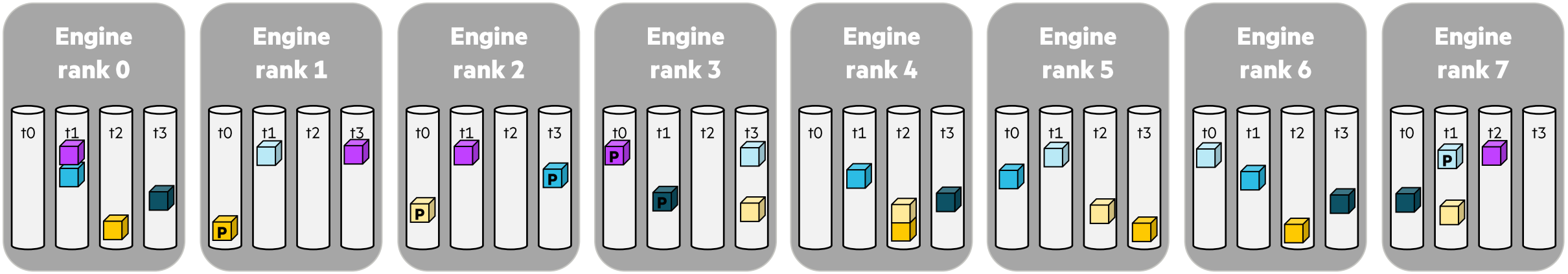


Data Distribution and Data Protection

Erasure Coding, object class EC_4P1GX (rf_lvl=engine)



DFS **chunk_size** (1 MiB); pool's **ec_cell_sz** should be set to match (here: 256kiB)



DAOS Object - Old Update Example

```
daos_obj_open(coh, oid, DAOS_OO_RW, &oh, NULL);
d_iov_set(&dkey, "dkey1", strlen("dkey1"));

d_iov_set(&sg_iov, buf, BUFLEN);
sgl[0].sg_nr = 1;
sgl[0].sg_iovs = &sg_iov;
sgl[1].sg_nr = 1;
sgl[1].sg_iovs = &sg_iov;

d_iov_set(&iod[0].iod_name, "akey1", strlen("akey1"));
d_iov_set(&iod[1].iod_name, "akey2", strlen("akey2"));

iod[0].iod_nr = 1;
iod[0].iod_size = BUFLEN;
iod[0].iod_recxs = NULL;
iod[0].iod_type = DAOS_IOD_SINGLE;

iod[1].iod_nr = 1;
iod[1].iod_size = 1;
recx.rx_nr = BUFLEN;
recx.rx_idx = 0;
iod[1].iod_recxs = &recx;
iod[1].iod_type = DAOS_IOD_ARRAY;

daos_obj_update(oh, DAOS_TX_NONE, 0, &dkey, 2, &iod, &sgl, NULL);
```

Multi-Level KV Object – Old Fetch Example

```
daos_obj_open(coh, oid, DAOS_OO_RW, &oh, NULL);
d_iov_set(&dkey, "dkey1", strlen("dkey1"));

d_iov_set(&sg_iov, buf, BUFLEN);
sgls[0].sg_nr = 1;
sgls[0].sg_iovs = &sg_iov;
sgls[1].sg_nr = 1;
sgls[1].sg_iovs = &sg_iov;

d_iov_set(&iod[0].iod_name, "akey1", strlen("akey1"));
d_iov_set(&iod[1].iod_name, "akey2", strlen("akey2"));

iod[0].iod_nr = 1;
iod[0].iod_size = BUFLEN; /** if size is not known, use DAOS_REC_ANY and NULL sgl */
iod[0].iod_recxs = NULL;
iod[0].iod_type = DAOS_IOD_SINGLE;

iod[1].iod_nr = 1;
iod[1].iod_size = 1; /** if size is not known, use DAOS_REC_ANY and NULL sgl */
recx.rx_nr = BUFLEN;
recx.rx_idx = 0;
iod[1].iod_recxs = &recx;
iod[1].iod_type = DAOS_IOD_ARRAY;

daos_obj_fetch(oh, DAOS_TX_NONE, 0, &dkey, 2, &iods[0], &sgls[0], NULL, NULL);
```

SUMMARY

Hewlett Packard
Enterprise

epcc

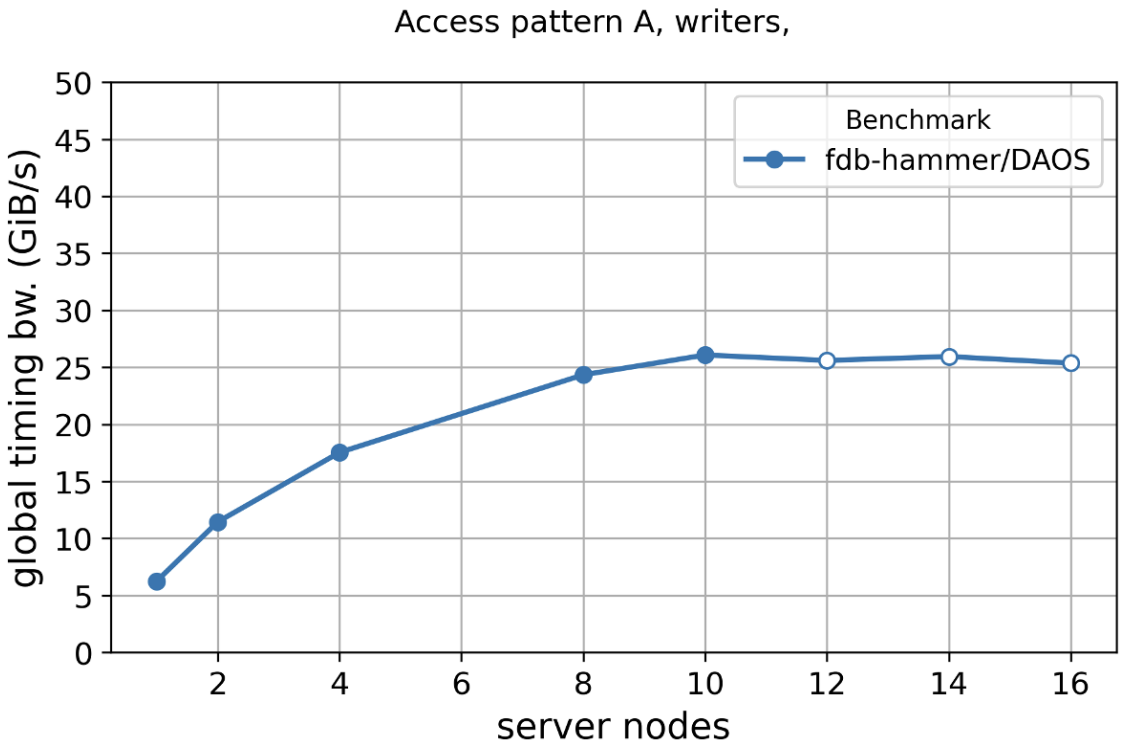
Aside: DAOS Fortran interfacing

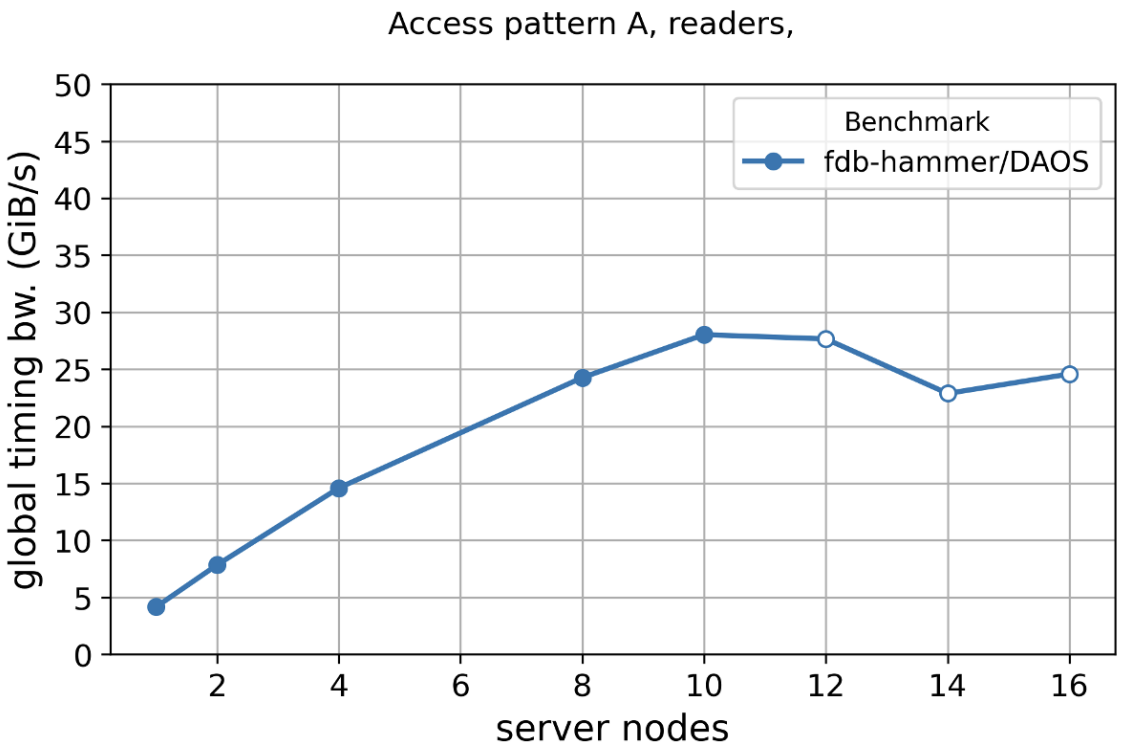
```
type, public, bind(c) :: daos_array_stbuf_t
     integer (kind=daos_size_t) :: st_size
     integer (kind=daos_epoch_t) :: st_max_epoch
  end type daos_array_stbuf_t


interface


   integer(kind=c_int) function daos_array_create(coh, oid, th, cell_size, chunk_size,
oh, ev) bind(c,name="daos_array_create")
     import :: c_int
     import :: daos_handle_t
     import :: daos_obj_id_t
     import :: daos_size_t
     import :: daos_event_t
     type(daos_handle_t), value, intent(in) :: coh
     type(daos_obj_id_t), value, intent(in) :: oid
     type(daos_handle_t), value, intent(in) :: th
     integer(kind=daos_size_t), value, intent(in) ::cell_size
     integer(kind=daos_size_t), value, intent(in) ::chunk_size
     type(daos_handle_t), intent(inout) :: oh
     type(daos_event_t), intent(inout) :: ev
   end function daos_array_create
```

Hewlett Packard Enterprise

epcc

Evaluate performance/approach

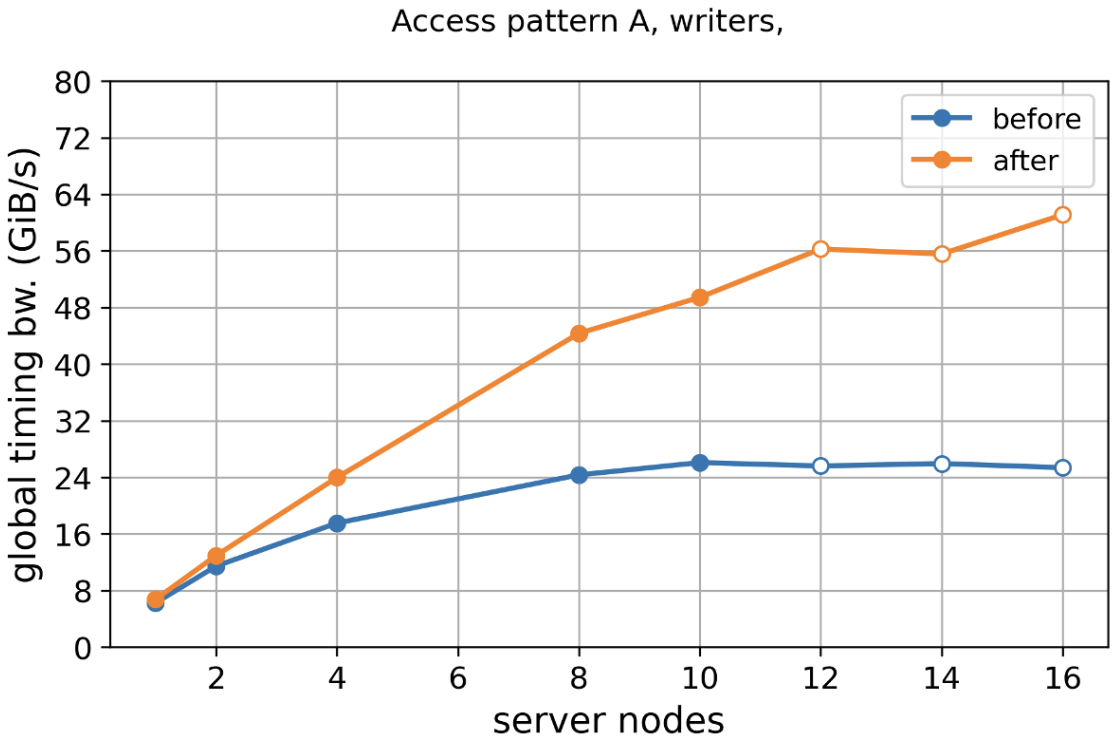


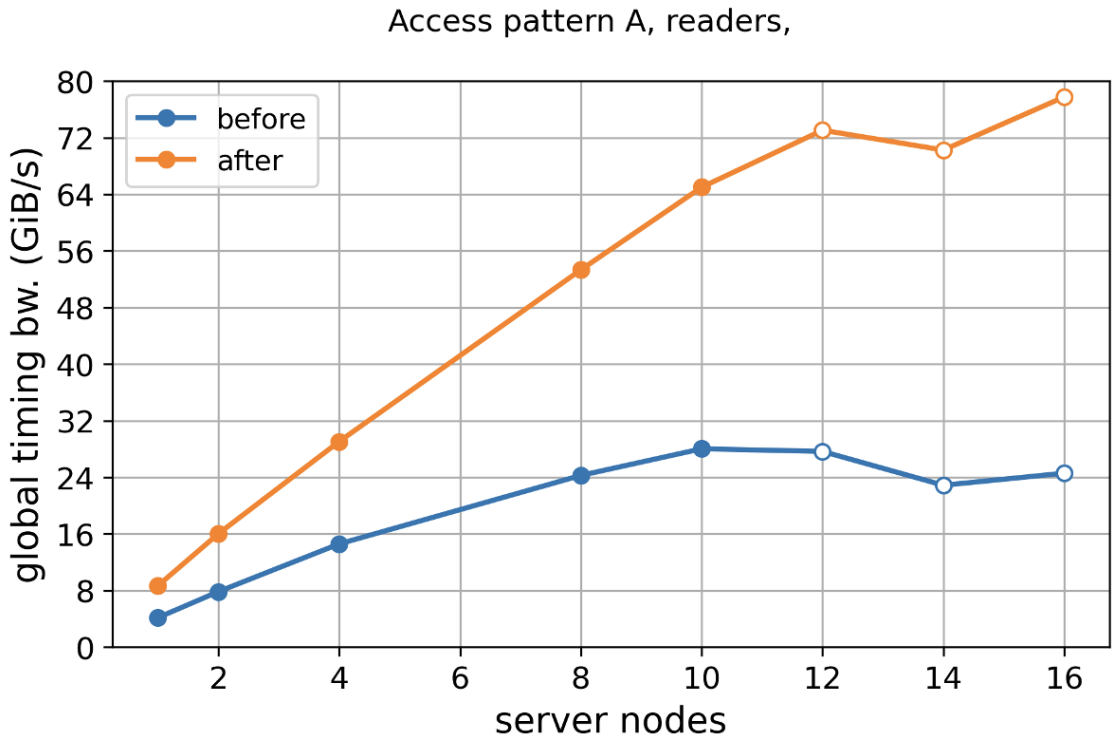


Hewlett Packard Enterprise

epcc

Optimised performance



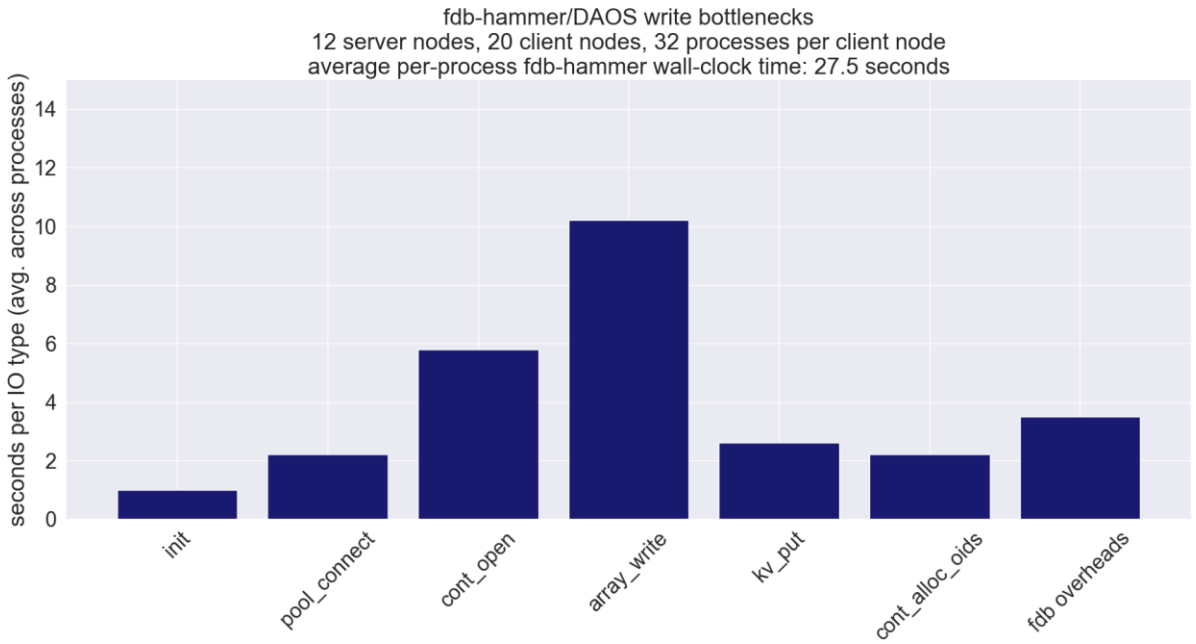


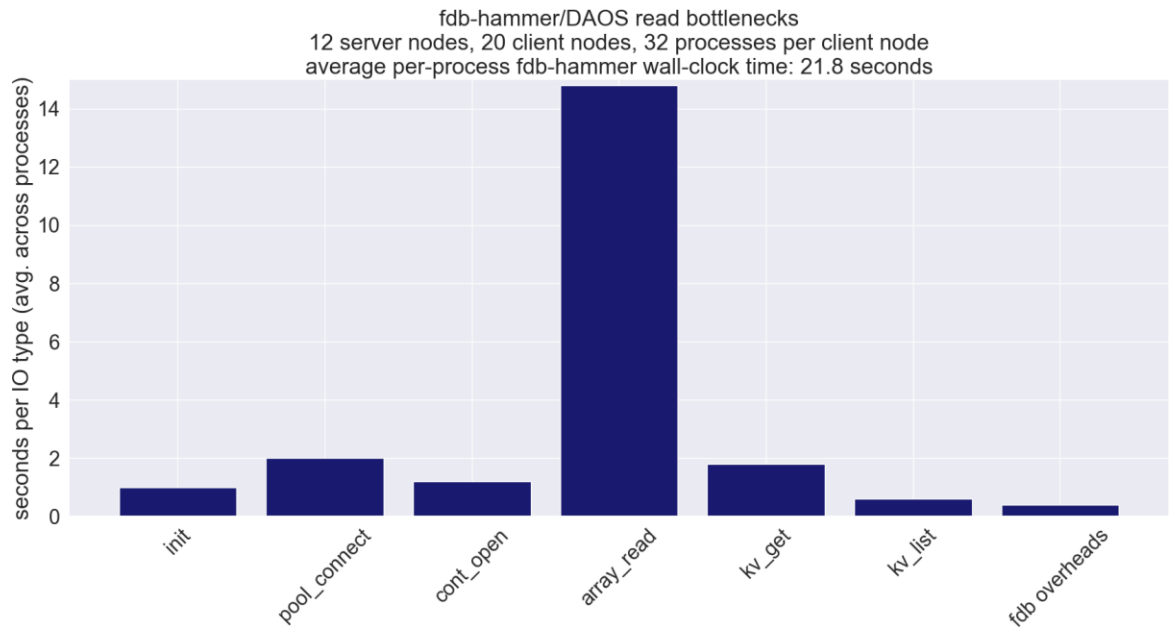
Hewlett Packard Enterprise

epcc

Profiling

- Example breakdown of where time is being spent
  - Manual profiling





Approach/recommendations

- Key-Value contention
- For a specific benchmark run configured with contention across processes on indexing Key-Values:
  - 20 GiB/s write
  - 13 GiB/s read
- Tweaking the benchmark configuration to have all processes operate on a separate Key-Values:
  - 35 GiB/s write
  - 68 GiB/s read
- This may not be trivial or possible for all applications, but if design can achieve it then this improves performance

Approach/recommendations

- Avoid communications on/with the server where possible
- Cache objects locally in DRAM if possible
- Use `daos_array_open_with_attr` to avoid `daos_array_create` calls
  - Only supported for `DAOS_OT_ARRAY_BYTE`, not for `DAOS_OT_ARRAY`
  - Warning: the cell size and chunk size attributes need to be provided consistently on any future `daos_array_open_with_attr` to avoid data corruption
- `daos_array_get_size` calls can be expensive
  - Can store array size in our indexing Key-Values
  - Can manually calculate
  - Also possible to infer the size by reading with overallocation:
    - use `DAOS_OT_ARRAY_BYTE`, over-allocate the read buffer, and read without querying the size. The actual read size (`short_read`) will be returned
- `daos_cont_alloc_oids` is expensive, call it just once per writer process
  - Required to generate object ideas to use in calls but can generate many at one

Approach/recommendation

- Creating several containers (starting at ~300) in a DAOS pool reduces performance
- Opening the same container from all processes is expensive
  - this happens even if only a few containers exist in the DAOS pool
  - e.g. out of 20 seconds taken by a process to write 2000 fields, 1.5 seconds were spent just to open one container
  - we observed this starting at ~200 parallel processes
  - Sharing handles using MPI is the way to fix this
- Opening more than one container per process is very expensive
  - e.g. out of 30 seconds taken by a process to read 2000 fields, 6 seconds were spent just to open two containers

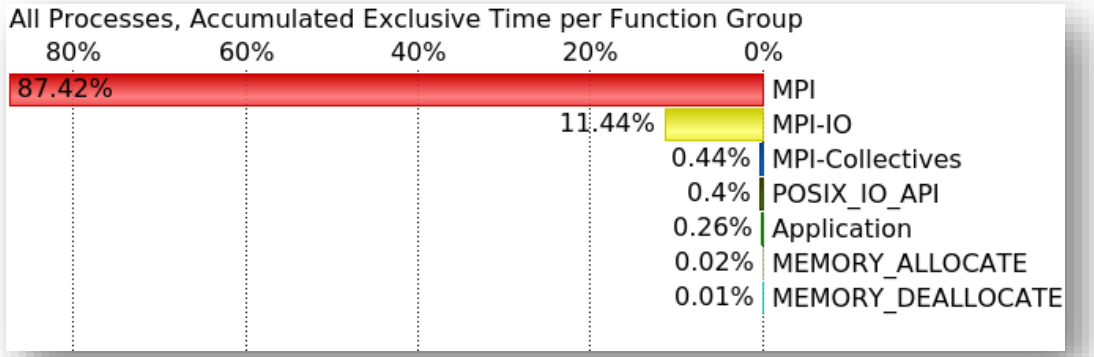
Approach/recommendations

- `daos_key_value_list` is expensive
- `daos_array_open_with_attrs`, `daos_kv_open` and `daos_array_generate_oid` are very cheap (no RPC)
- Normal `daos_array_open` is expensive
- `daos_cont_alloc_oids` is expensive
- `daos_kv_put` and `_get` are generally cheap
  - Value size impacts this
- `daos_obj_close`, `daos_cont_close` and `daos_pool_disconnect` are cheap
- Server configuration to use available networks/sockets/etc… important for performance
  - Just like any storage system or application

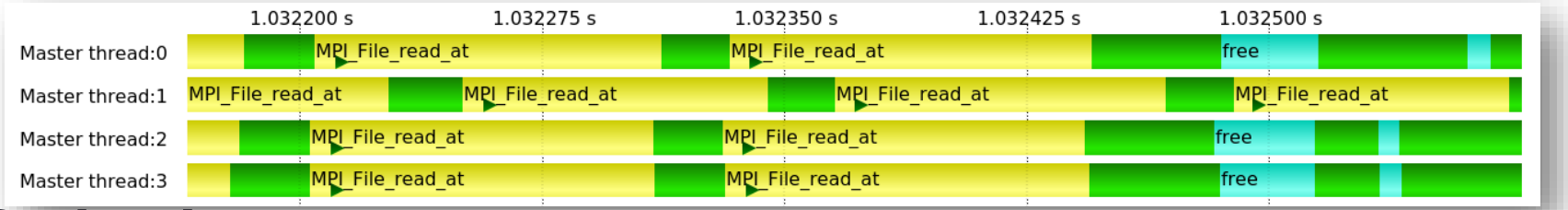
Object store usage design

- Mapping data structures to KV and Array objects is key to getting good performance functionality
- We suggest mapping contiguous chunks of arrays to be stored to single DAOS array object
  - Collect multiple arrays with associated KV to make the whole array
- Can be as extreme as having a single value per KV
  - Significant overheads in this
- Depends on your application data structures you may want to aggregate less data for I/O
  - Group based on meaningful/scientific dimensions
- HDF5 or similar hierarchies could map well to Keys with Arrays
- Adding keys to the array data/values can let data set structure to be created, enumerated, and extended
- See the Exercises/FullApplication in the GitHub repository for the tutorial

Summary



- Object storage can provide high performance
  - DAOS: 90+ GB/s per server is possible
  - Hardware and configuration dependent, just like all I/O
- Built in replication and redundancy under your/user control
- Different interfaces available
  - Filesystem for zero cost porting
  - Simple file like access for slightly improved performance at little effort
  - Programming APIs for full functionality
- Object store interface enables changing I/O granularity/patterns for bigger benefits



Hewlett Packard Enterprise

epcc

Final Summary

- Thanks for attending!
- Happy to take further questions when/if they occur to you
  - Email or come and talk to us
- Tutorial system will stay active for the week
  - Time to complete the exercises/experiment with the technology
  - Any problems email me as well
- Want more help
  - Come and speak to us
  - Happy to collaborate/help with object store usage/porting/etc...

Hewlett Packard Enterprise

epcc