D5.6
Storage Data Mapping and Memory Integration Implementation for Maestro

Work package:	**WP5 Low Level Middleware**	
Author(s):	Sining Wu, Ganesan Umanesan, Sai Narasimhamurthy	Seagate
Reviewer #1	Dirk Pleiter	JUELICH
Reviewer #2	Maxime Martinasso	ETHZ
Dissemination Level	Confidential	
Nature	Technical	

Date	Author	Comments	Version	Status
02.08.2021	Sai Narasimhamurthy	Table of Contents	V0.1	Draft
04.08.2021	Sining Wu	Draft	V0.2	Draft
27.08.2021	Sining Wu	Updates based on internal review	V1.0	Final

DATA ORCHESTRATION IN HIGH PERFORMANCE COMPUTING
This project has received funding from the European Union's Horizon 2020 research and innovation program through grant agreement 801101.

Glossary

Acronym	Full form
CDO	Core Data Object
PFS	Parallel File System
SNS	Server Network Striping
RAID	Redundant Array of Independent Disks
HSM	Hierarchical Storage Manager
NBA	Non-Blocking Availability
DTM	Distributed Transaction Management
RM	Resource Manager
LM	Layout Manager
NBA	Non-Blocking Availability
Motr	Formerly Mero
Motr API	Formerly Clovis
FOL	File Operation Log
MIO	Maestro IO Interface

This project has received funding from the European Union's Horizon 2020
research and innovation program through grant agreement 801101.

Executive Summary

This deliverable will provide the final implementation and validation of the storage and memory access backend used for Maestro enabled through the Maestro I/O Interface (MIO). Please note that the focus of MIO is on persistent storage backends, which could also include persistent memory used as storage. The deliverable also covers the final APIs available to Maestro (including Guided I/O).

This deliverable follows the high-level architecture of the Mero Object store and the Clovis API which had be provided in D5.2.

Following the open sourcing of Mero & Clovis, Mero has been renamed as Motr, which is the open-source version of Mero, and Clovis has been renamed as Motr API. We will use Mero/Motr and Clovis/Motr API terms interchangeably (as some of the code still may retain the old naming).

Contents

1. Introduction

This project has received funding from the European Union's Horizon 2020 research and innovation program through grant agreement 801101.

The Maestro project is building a data-aware and memory-aware middleware framework that addresses ubiquitous problems of data movement in complex memory hierarchies and at many levels of the HPC software stack.

The Maestro I/O Interface (MIO) has been described in D5.2 as the default interface for doing I/O from Maestro to the different storage back-ends. The focus of this project has been the Mero Object storage with the Clovis API. We described the working of the Mero object store and the Clovis API in D5.2. D5.2 also covered the MIO interface definitions, API specification for Object Manipulations, Object Data Access and Operations (including the Key Value Store). The guided I/O hints interface was also described.

In this deliverable we describe the final validation/working of the MIO interface alongside guided I/O. We also provide links to the final code base.

The following is the tasks in Maestro that link into this deliverable:

Task T5.3: Memory and Storage Integration (Month 1 - Month 36)

This task provides the integration of the memory and persistent storage pools to the Maestro middleware. The Mero object store will be co-designed keeping the requirements of the use cases and the Maestro middleware in mind. The co-design will involve defining and developing the different "extreme scale" components of the Mero object store for scalability, persistence and resilience on top of the basic object store functionalities exposed through the Clovis interface. The guided I/O library that works with Mero will be provided along with the component that interfaces with the job schedulers for appropriate staging of data. The Mero stack as needed for Maestro will then be provided. The Mero object store will be able to work with Memory pools, Flash, non-volatile memory and disk technologies. Access methods to memory, persistent memory, and memory-mapped disk will also be developed in this task. Reading and writing data to memories can involve using the map function of a PDO to access distributed memories efficiently.

MIO (with guided I/O) on top of Mero/Motr was designed, implemented and validated to achieve the outcomes of the task.

Section 2 and 3 explain how the MIO access interface and guided-IO interface were implemented. The MIO source code can be found in [1]: https://github.com/Seagate/cortx-mio . We also show screenshots of MIO tests for its access APIs and a guided-IO based Hierarchical Storage Manager (HSM) example to validate the working of MIO (& guided I/O).

2. MIO Access Interface Implementation

2.1 MIO Instance and Drivers

The Maestro IO Interface (MIO) is designed as an abstraction layer on top of multiple object stores to allow Maestro applications to access different types of object store systems in a uniform way.

MIO maintains a global instance of *struct mio* which holds details about a runtime instance of the MIO interface, such as MIO telemetry. More importantly, it stores information about the physical object store system (*struct mio::m_driver*) that Maestro applications write data

This project has received funding from the European Union's Horizon 2020
research and innovation program through grant agreement 801101.

into and read data from. An MIO instance is initialized with a configuration file by *mio_init()* and finalized by *mio_fini()* respectively.

Each type of object store system supported by MIO (for example, Cortx-motr [3]) is assigned a globally unique ID (*struct mio::m_driver_id*) and is defined using the data structure *struct mio_driver* (see Table 1) which keeps sets of function pointers: driver initialisation/finalisation, operation related functions (called "operations on operation" in MIO), object and key-value set accessing operations etc. These driver-implemented functions define how MIO access the physical object store system.

```
/** Each Maestro instance data structure. */
struct mio {
	enum mio_telemetry_store_type m_telem_store_type;
	char *m_telem_prefix;

	enum mio_log_level m_log_level;
	char *m_log_dir;

	enum mio_driver_id m_driver_id;
	struct mio_driver *m_driver;
	void *m_driver_confs;
};

/**
 * Each Maestro IO driver is defined by a set of operations. Driver
 * initialisation/finalisation and object and key-value set accessing
 * operations must be implemented for a driver, while motr-inspired
 * operations are optional.
 */
struct mio_driver {
	/** Driver's operations, such as initialisation and finalisation. */
	struct mio_driver_sys_ops *md_sys_ops;

	struct mio_op_ops *md_op_ops;

	/** Pool operations. */
	struct mio_pool_ops *md_pool_ops;

	/** Object and key-value set operations. */
	struct mio_obj_ops *md_obj_ops;
	struct mio_kvs_ops *md_kvs_ops;

	/** Motr-inspired APIs. */
	struct mio_comp_obj_ops *md_comp_obj_ops;
};
```

Table 1 MIO Instance and Driver Structures

2.2 MIO Access Interface (Object and Key-value Set)

This section will cover the currently supported MIO access APIs, including object access APIs and key-value set APIs. The MIO access data structures and the details on how a MIO driver implements the supported APIs are explained from section 2.2.2 to section 2.2.4. Section 2.2.2 discusses the definition of MIO operations and how an MIO operation encapsulates multiple driver operations. Section 2.2.3 gives details on how an object is represented and accessed. Section 2.2.4 describes how a key-value set is defined, the 4 queries (GET, PUT, NEXT and DEL) and their corresponding MIO APIs.

This project has received funding from the European Union's Horizon 2020
research and innovation program through grant agreement 801101.

2.2.1 Access APIs

MIO access APIs are built around asynchronous operations which each represents the execution of a request from MIO applications. Table 2 lists the access APIs which all (except the object locking APIs) have a pointer to *struct mio_op* as one parameter. Upon successfully calling MIO access APIs a request is sent to the underneath object store and an in-memory data struct of operation is returned, which is used to track the progress of this operation by polling the operation's state (by calling *mio_op_poll()*). Applications can also set up callback functions for operations (by calling *mio_op_callbacks_set()*) which are triggered when the operation reaches COMPLETION or FAILED, respectively.

Object Access	mio_obj_open mio_obj_close	Open an object and return with an object handle. Close an object handle.
	mio_obj_create mio_obj_delete	Create an object and return with an object handle. Delete an object.
	mio_obj_readv mio_obj_writev	Read data to an object in scatter-gather method. Write data to an object in scatter-gather method.
	mio_obj_sync	Sync data to persistent devices.
	mio_obj_size	Query object size
	mio_obj_lock mio_obj_unlock	Lock or unlock a whole object.
Key-value Set	mio_kvs_create mio_kvs_delete	Create a key-value set. Delete a key-value set.
	mio_kvs_pair_get mio_kvs_pair_next mio_kvs_pair_put mio_kvs_pair_del	Query a key-value set with keys. Retrieve key-value pairs starting with the specifying key. Insert key-value pairs into a key-value set. Remove key-value pairs from the specified set.
Operation	mio_op_poll mio_op_callbacks_set	Query on the state of an operation. Set callback functions for an operation.

Table 2 MIO Access APIs

2.2.2 Operation

struct mio_op (see Table 3) represents the in-memory data structure of an asynchronous MIO operation, in which *struct mio_op::mop_state* keeps the state of the operation. The application's callback functions are stored in *struct mio_op::mop_app_cbs* with one function for operation completion and another one for failure. *struct mio_op::mop_op_ops* is a pointer to the set of driver's "operations on operation" (*struct mio_op_ops*). "Operations on operation" set defines how the driver handles an operation's progress. As discussed above, an MIO application can either wait for the completion of the operation or set callback functions for the operation, these 2 ways of handling an operation are reflected in the member function pointers of *struct mio_op_ops: struct mio_op_ops::mopo_wait* and *struct mio_op_ops::mopo_set_cbs*.

struct mio_op is also a wrapper for driver specific operations, containing a member (*struct mio_op::mop_drv_op_chain*) that keeps the information about the sequence of driver specific

This project has received funding from the European Union's Horizon 2020 research and innovation program through grant agreement 801101.

operations (defined by data struct *struct mio_driver_op_chain*). In many situations, an MIO operation only needs to launch one single driver specific operation, but there exist some cases where an MIO operation translates to a sequence of driver operations. For example, in the object write example of section 2.2.3, a driver key-value PUT operation is executed to update the object size following a successful driver object write operation.

```
struct mio_op {
        uint64_t mop_seqno;

        unsigned int mop_opcode; /* Type of operation. */
        union {                  /* Which object or key-value set.*/
                struct mio_obj *obj;
                struct mio_kvs_id *kvs_id;
        } mop_who;
        int mop_state;           /* Current state of operation. */
        int mop_rc;              /* == 0 when operation successes,
                                 *  < 0 the error code if the
                                 *      operation fails. */

        struct mio_op_app_cbs mop_app_cbs;

        /* See mio_drv_op_chain in mio_inernal.h for explanation. */
        struct mio_driver_op_chain mop_drv_op_chain;

        struct mio_op_ops *mop_op_ops;
};
```

Table 3 MIO Operation Data Structure

2.2.3 Object Access APIs

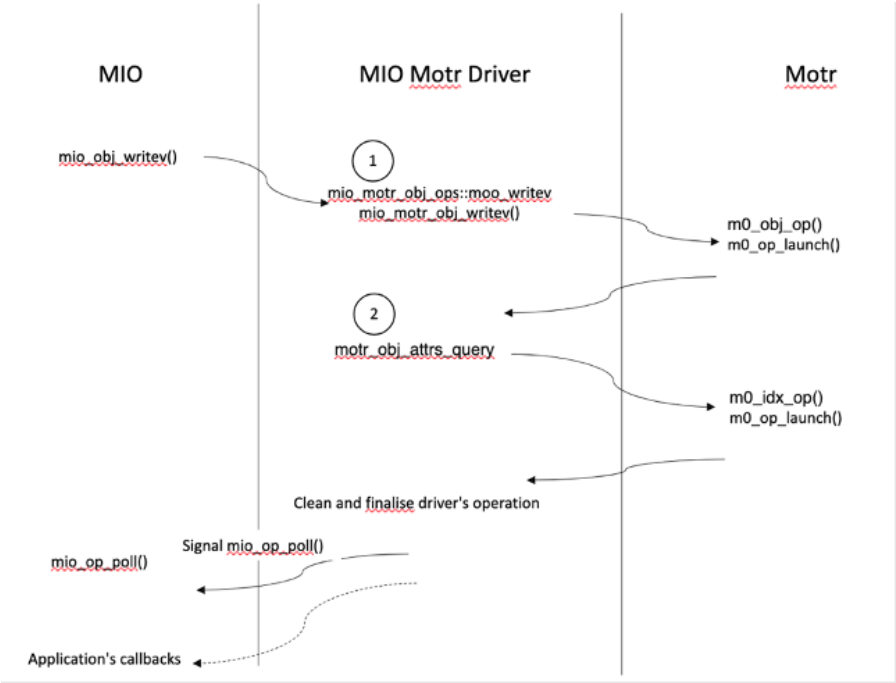

Figure 1 MIO Object Write Operation

A pool (*struct mio_pool* in Table 4) is a group for storing objects. An object store may contain physical pools and each of these physical pools are configured with storage devices of the same tier.

This project has received funding from the European Union's Horizon 2020 research and innovation program through grant agreement 801101.

```
struct mio_pool {
	/** the name by which this layer is referenced by the user */
	char mp_name[MIO_POOL_MAX_NAME_LEN + 1];

	/** Pool id. */
	struct mio_pool_id mp_id;

	enum mio_pool_type mp_type;

	/* Characteristics of the pool. */
	/** Capacity of the layer (not necessarily constant or current) */
	size_t mp_capacity;

	/** Optimised data buffer alignment */
	size_t mp_opt_alignment;

	/**
	 * Preferred block sizes for read and write operations, ordered in
	 * decreasing order of performance. These parameters are
	 * initialised using driver specific experiential `formula`, and
	 * could be updated by monitoring telemetry data.
	 */
	size_t mp_nr_opt_blksizes;
	size_t mp_opt_blksizes[MIO_POOL_MAX_NR_OPT_BLKSIZES];
};
```

Table 4 MIO Pool Data Structure

An object is an array of bytes with a 128-bit identifier and each object contains a set of attributes (*mio_obj::mo_attrs of struct mio_obj_attrs* in Table 5), including object size. *struct mio_obj* defines an object handler and is an in-memory data structure created by opening an object and is released by closing the object. All following operations on the object are performed through the object handler. The actual definition of the object identifier depends on the object store and its MIO driver. Where and how the object attributes are stored are also decided by the object store and its driver. For example, an MIO driver can take advantage of the built-in object attributes if an object store has already defined and exposed the attributes to applications, but if an object store does not expose object attributes, its MIO driver can choose to store the attributes in a key-value set.

struct mio_obj (see Table 5) has a pointer to a set of driver function pointers (*mio_obj::mo_drv_obj_ops of struct mio_obj_ops* in Table 5) to create and launch driver-specific operations. *struct mio_obj_ops* defines the list of currently supported functions, including object creation/deletion, open/close, write/read/sync, locking, pool and hint operations (more details see section 3).

This project has received funding from the European Union's Horizon 2020
research and innovation program through grant agreement 801101.

```
struct mio_obj_attrs {
        uint64_t moa_size;

        /**
         * Object access statistics. MIO shows one usage of
         * this information to calculate object hotness.
         */
        struct mio_obj_stats moa_stats;

        /* Persistent hints only. */
        struct mio_hints moa_phints;
};

struct mio_obj {
        struct mio_obj_id mo_id;
        struct mio_obj_op *mo_op;

        /**
         * Sequence number of an opened object session. This sequence
         * number can be used to associated all operations issued
         * in this session, which can be used in telemetry data
         * analysis.
         */
        uint64_t mo_sess_seqno;

        /** Driver specific object ops. */
        struct mio_obj_ops *mo_drv_obj_ops;

        /** Hints (persistent + session hints) set for this object. */
        struct mio_hints mo_hints;

        /** Associated metadata key-vaule set. */
        struct mio_kvs *mo_md_kvs;

        /** If the object's attributes have been updated. */
        bool mo_attrs_updated;
        struct mio_obj_attrs mo_attrs;

        /** Pointer to driver specific object structure. */
        void *mo_drv_obj;

        /** Pointer to driver specific object lock. */
        void *mo_drv_obj_lock;
};
```

Table 5 In-memory Object Handler Data Structure

This project has received funding from the European Union's Horizon 2020
research and innovation program through grant agreement 801101.

```
struct mio_obj_ops {
        int (*moo_open)(struct mio_obj *obj, struct mio_op *op);
        int (*moo_close)(struct mio_obj *obj);
        int (*moo_create)(const struct mio_pool_id *pool_id,
                          struct mio_obj *obj, struct mio_op *op);
        int (*moo_delete)(const struct mio_obj_id *oid,
                          struct mio_op *op);

        int (*moo_writev)(struct mio_obj *obj,
                          const struct mio_iovec *iov,
                          int iovcnt, struct mio_op *op);
        int (*moo_readv)(struct mio_obj *obj,
                         const struct mio_iovec *iov,
                         int iovcnt, struct mio_op *op);
        int (*moo_sync)(struct mio_obj *obj, struct mio_op *op);
        int (*moo_size)(struct mio_obj *obj, struct mio_op *op);
        int (*moo_pool_id)(const struct mio_obj *obj,
                           struct mio_pool_id *pool_id);

        /**
         * Exclusive whole object (blocking) lock.
         */
        int (*moo_lock)(struct mio_obj *obj);
        int (*moo_unlock)(struct mio_obj *obj);

        /**
         * Load and store persistent hints. Unlike other operations,
         * storing/loading hints are synchronous operations because
         * not all hints are persistent hints, if applications use
         * only session hints, no IO (to retrieve persistent hints)
         * will be launched, thus no 'op' is needed. And as setting
         * hints is rare comapred to object accesses, its overhead
         * is considered small.
         */
        int (*moo_hint_store)(struct mio_obj *obj);
        int (*moo_hint_load)(struct mio_obj *obj);
};
```

Table 6 MIO Object Operations' Data Structure

To show how MIO interacts with the object store driver, we use Figure 1 to illustrate the detailed steps of object write in MIO when Motr is configured as a driver. As described above, an object consists of object data and its attributes, if any data is written beyond the end of the object, the Motr driver updates the object's size in the key-value set that stores the object's attributes. When an application calls *mio_obj_writev()*, the returned MIO operation is actually performed in 2 Motr operations behind the scene as shown in Figure 1. The first Motr object access operation is sent to the Motr data services to write data to devices followed by the second key-value query operation to update the object's size if needed. Once the MIO operation is executed, either successfully or failed, its state is set correspondingly, and MIO will wake up the application from blocking at *mio_op_poll()* or call the corresponding callback function.

2.2.4 Key-value Set APIs

A key-value set stores records, each record consisting of a key and a value. Keys are unique within a set. Keys and values within the same set can be of variable size. Besides operations to create and delete a key-value set (implemented as *mio_kvs_create_set()* and

This project has received funding from the European Union's Horizon 2020
research and innovation program through grant agreement 801101.

mio_kvs_delete_set()), 4 types of basic query operations are defined below. For each key-value set operation, MIO provides a corresponding API which launches an operation after creating, initializing, and setting its parameters. The returned MIO operation can be used to track the progress of the operation.

- GET: given a set of keys, return the matching records from the key-value set. The Get operation is implemented in *mio_kvs_pairs_get()*.
- NEXT: given a starting key, return as many key-value pairs starting from the specified key as requested by the application. The NEXT operation is implemented in *mio_kvs_pairs_next()*.
- PUT: given a set of records, place them in the specified set, overwriting existing records if necessary, inserting new records otherwise. The PUT operation is implemented in *mio_kvs_pairs_put()*.
- DEL: given a set of keys, delete the matching records from the key-value set. Its corresponding API is *mio_kvs_pairs_del()*.

A key-value set is represented as *struct mio_kvs* which has a globally unique ID (*128-bit ID*) and a pointer to a set of function pointers (*struct mio_kvs_ops*) each implementing one of the 4 operations described above using the driver's key-value store functionalities. The interaction between MIO and its driver is similar to the object write explained in section 2.2.3 and Figure 1.

```
struct mio_kvs {
        struct mio_kvs_id mk_id;
        /** Driver specific key-value set data structure. */
        void *mk_drv_kvs;
        struct mio_kvs_ops *mk_ops;
};

struct mio_kvs_ops {
        int (*mko_get)(struct mio_kvs_id *kvs_id,
                       int nr_kvps, struct mio_kv_pair *kvps,
                       int32_t *rcs, struct mio_op *op);

        int (*mko_next)(struct mio_kvs_id *kvs_id,
                        int nr_kvps, struct mio_kv_pair *kvps,
                        bool exclude_start_key, int32_t *rcs,
                        struct mio_op *op);

        int (*mko_put)(struct mio_kvs_id *kvs_id,
                       int nr_kvps, struct mio_kv_pair *kvps,
                       int32_t *rcs, struct mio_op *op);

        int (*mko_del)(struct mio_kvs_id *kvs_id,
                       int nr_kvps, struct mio_kv_pair *kvps,
                       int32_t *rcs, struct mio_op *op);

        int (*mko_create_set)(struct mio_kvs_id *kvs_id,
                              struct mio_op *op);
        int (*mko_del_set)(struct mio_kvs_id *kvs_id,
                           struct mio_op *op);
};
```

Table 7 Data Structures MIO Key-value Set and Operations

This project has received funding from the European Union's Horizon 2020 research and innovation program through grant agreement 801101.

2.3 MIO Access Interface Tests

To demonstrate the functionalities of the MIO access interface, we developed a test suite: each test represents a few use cases of different aspects of MIO access interface, including object creation/deletion and object IOs, key-value set queries, pool querying and object hints. The test suite also demonstrates how MIO can be used in multi-threading and multi-processing applications.

Figure 2 shows the output of each MIO test. The detailed output of each test was recorded into a log file and is displayed from Figure 3 – 5. As explained above, an application can either block on the operation (called *sync mode* in the MIO test suite) by calling *mio_op_poll()* or set call-back functions (called *async mode*), these 2 modes for object IO are shown in Figure 3. Figure 4 shows the result for MIO object locking test. Test for multi-threading IO is shown in Figure 5. The results for the MIO key-value set tests can be seen in Figure 6.

```
[sining@s3dev tests]$ sudo ./mio_run_tests.sh
MIO tests start:
Test log will be stored in /home/sining/maestro/mio_github/cortx-mio/tests/mio_test_sandbox_2021-07-14-0
7-12-13/mio_test_2021-07-14-07-12-13.log.
[T0] Object creation and deletion tests
        io_create_delete_test:  passed
[T1] Object IO tests
        io_test_with_sizes_of_multi_4KB:  passed
        io_test_with_sizes_of_multi_non_4KB:  passed
        io_test_in_async:  passed
        io_test_obj_rw_with_lock:  passed
        io_test_with_multi_threads:  passed
        io_test_with_multi_procs:  passed
[T2] Composite object tests
        comp_obj_test:  passed
[T3] Key/value set (KVS) tests
        kvs_create_query_delete_test:  passed
[T4] Pool tests
        pool_query_test:  passed
[T5] Object hint tests
        obj_hint_test:  passed
mio_run_tests: test status: SUCCESS
[sining@s3dev tests]$
```

Figure 2 MIO Tests

```
[30] /home/sining/maestro/mio_github/cortx-mio/examples//mio_cat -s 31744 -c 16 -o 1:12345678 -y /home/s
ining/maestro/mio_github/cortx-mio/tests/mio_config.yaml /home/sining/maestro/mio_github/cortx-mio/tests
/mio_test_sandbox_2021-07-14-07-12-13/507904.out
        io_test_with_sizes_of_multi_non_4KB:  passed

[46] /home/sining/maestro/mio_github/cortx-mio/examples//mio_cat -s 16384 -c 16 -o 1:12345678 -a -y /hom
e/sining/maestro/mio_github/cortx-mio/tests/mio_config.yaml /home/sining/maestro/mio_github/cortx-mio/te
sts/mio_test_sandbox_2021-07-14-07-12-13/262144.out
        io_test_in_async:  passed
```

Figure 3 MIO Object IO Tests in Sync and Async Modes

```
[48] /home/sining/maestro/mio_github/cortx-mio/examples//mio_rw_lock -s 4096 -c 16 -o 0:12345678 -t 4
      -y /home/sining/maestro/mio_github/cortx-mio/tests/mio_config.yaml &>> /home/sining/maestro/mio_gi
thub/cortx-mio/tests/mio_test_sandbox_2021-07-14-07-12-13/mio_test_2021-07-14-07-12-13.log
[0]     WRITER  3db1fac0f1343d3a43bf4f8b54cc8c16
[1]     WRITER  1f5eacc8bc7db97b89c9edf0436e20ce
[2]     WRITER  dc6ce9adaada838b4aa6e28fc9aa55ae
[3]     READER  dc6ce9adaada838b4aa6e28fc9aa55ae        MATCHED
[4]     WRITER  59594802e6fa6cfbdc50728983daeda2
[5]     READER  59594802e6fa6cfbdc50728983daeda2        MATCHED
[6]     WRITER  51b0bc9ab2c31939e5eaee301a2115e4
[7]     READER  51b0bc9ab2c31939e5eaee301a2115e4        MATCHED
[8]     READER  51b0bc9ab2c31939e5eaee301a2115e4        MATCHED
```

Figure 4 MIO Object Locking Test

This project has received funding from the European Union's Horizon 2020
research and innovation program through grant agreement 801101.

```
[49] /home/sining/maestro/mio_github/cortx-mio/examples//mio_rw_threads -s 4096 -c 16 -o 0:12345678 -t 2
             -n 10 -y /home/sining/maestro/mio_github/cortx-mio/tests/mio_config.yaml &>> /home/sining/
maestro/mio_github/cortx-mio/tests/mio_test_sandbox_2021-07-14-07-12-13/mio_test_2021-07-14-07-12-13.log
 Object ID                          WRITE MD5                                READ MD5
8:12345678      5c50ee814c3d01d91ca86d762b9dc56b        5c50ee814c3d01d91ca86d762b9dc56b
9:12345678      1b6f3625bd8d301241230fec04144ea6        1b6f3625bd8d301241230fec04144ea6
6:12345678      a6634c13d0ee70bf11644da15527d477        a6634c13d0ee70bf11644da15527d477
7:12345678      72bc1dc05c134f5eb67498f8a69b0b7f        72bc1dc05c134f5eb67498f8a69b0b7f
0:12345678      77f2d50f35a1056df969f1273ac6638e        77f2d50f35a1056df969f1273ac6638e
1:12345678      3640e06cfa1be0469bd0845ad9c06a57        3640e06cfa1be0469bd0845ad9c06a57
2:12345678      b22087e53251e55bd0f9e9e822e5cb35        b22087e53251e55bd0f9e9e822e5cb35
3:12345678      d090193de69da5f4ef620c2f62aa7d7a        d090193de69da5f4ef620c2f62aa7d7a
4:12345678      220cde3dd6ceeff9b7a91b67c9e7b00c        220cde3dd6ceeff9b7a91b67c9e7b00c
5:12345678      641ce7a90835cdae0cb0168507f29187        641ce7a90835cdae0cb0168507f29187
[Final Report]  Objects TODO: 10        Completed: 10   Failed: 0       Matched: 10

        io_test_with_multi_threads:  passed


[54] /home/sining/maestro/mio_github/cortx-mio/examples//mio_unlink -o 1:1002100 -y /home/sining/maestro
/mio_github/cortx-mio/tests/mio_config_p2.yaml
        io_test_with_multi_procs:  passed
```

Figure 5 Multi-threading Object IO Tests

```
[1] /home/sining/maestro/mio_github/cortx-mio/examples//mio_kvs_insert -k 7:12345678 -y /home/sining/mae
stro/mio_github/cortx-mio/tests/mio_config.yaml -s 300 -n 20 -l /home/sining/maestro/mio_github/cortx-mi
o/tests/mio_test_sandbox_2021-07-14-07-12-13/kvs_put.log
[2] /home/sining/maestro/mio_github/cortx-mio/examples//mio_kvs_retrieve -k 7:12345678 -y /home/sining/m
aestro/mio_github/cortx-mio/tests/mio_config.yaml -s 300 -n 20 -l /home/sining/maestro/mio_github/cortx-
mio/tests/mio_test_sandbox_2021-07-14-07-12-13/kvs_get.log
[3] /home/sining/maestro/mio_github/cortx-mio/examples//mio_kvs_del_pairs -k 7:12345678 -y /home/sining/
maestro/mio_github/cortx-mio/tests/mio_config.yaml -s 300 -n 20
[4] /home/sining/maestro/mio_github/cortx-mio/examples//mio_kvs_del_set -k 7:12345678 -y /home/sining/ma
estro/mio_github/cortx-mio/tests/mio_config.yaml
        kvs_create_query_delete_test:  passed
```

Figure 6 MIO Key-value Set Tests

3. Guided I/O Implementation

MPI-IO's hints enable MPI applications to select IO preferences and parameters on data access layouts, collective IO optimizations and file system specific configurations, but these hints are generally coarse-grained and only portray a static view on how an application uses IO. When processing hints, MPI-IO performs optimisations according to a set of pre-defined rules without considering the state of underlying file system and the application's workload characteristics, not being able to achieve the best performance.

The missing information of the state of the underlying file system can be gained from telemetry data provided by the I/O software stack or the underlying storage system. A new component, the telemetry processor, has been added in the MIO guided IO framework as illustrated in Figure 7.

This project has received funding from the European Union's Horizon 2020
research and innovation program through grant agreement 801101.

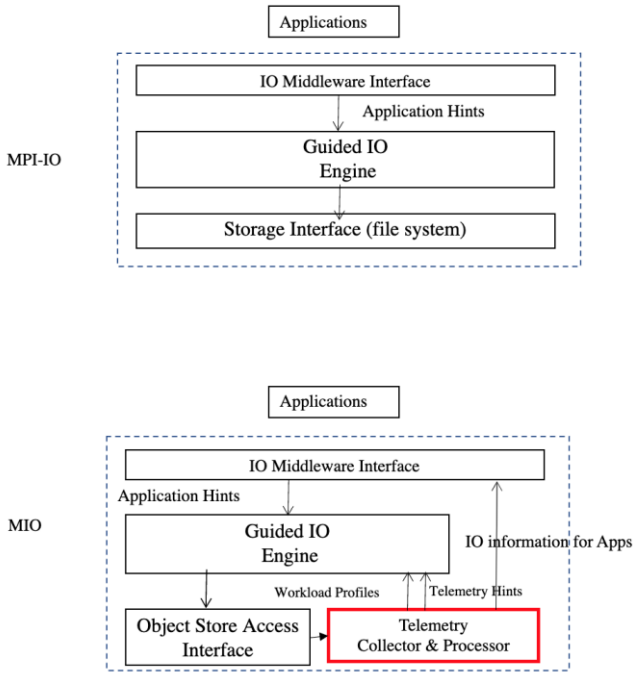

Figure 7 Guided I/O frameworks in MPI-IO and MIO

It accesses and analyses telemetry data to generate hints with richer representation and workload profiles. Unlike the application hints which are usually pre-set, the hints from the telemetry processor are distilled from continually generated telemetry data and will be constantly updated. An example of telemetry hints is to set the default data unit size of the de-clustered parity data layout according to the object IO performance observed through the telemetry data. Another example of a telemetry hint is object hotness which is captured and calculated during object IO operations. We demonstrate the application of object hotness hint using the HSM (Hierarchy Storage Management) system in section 3.3.

Up-to-date workload profiles reflecting characteristics of running applications can also be fed to the guided IO framework. Although the data volume generated by the telemetry sub-system is usually tuned to be modest, the amount of data accumulated every day is still large. Unlike a full-wedged telemetry data analysis system, the telemetry processor component is architected in a similar way to Map-Reduce processing. It enables us to spread most of the heavy-lifted data pre-processing across multiple nodes instead of collecting all raw telemetry data into a shared repository and processing them. This embarrassing parallelism makes fast updates of data possible.

3.1 Hints

The Guided IO interface provides the ability for MIO applications to provide "Hints" to the object store backend on upcoming data usage attributes. Each hint inside MIO is expressed as in a key-value format, *(hint_key, hint_value)*, where *hint_key* is a predefined and unique integer for each hint in MIO. Hints are stored in a data structure defined in *struct mio_hints* in Table 8. Unlike MPI-IO hints which are set for opened files only and the values of hints are lost when a file is closed, MIO differentiates 2 types of hints (*enum mio_hint_type* in Table 8):

- Session hints which are set only for an object that is opened and are kept alive only during the time where the object is open. Session hints will be destroyed when an object is closed, and the hints won't be valid for the next session. Example of session hints: object hotness and pool selection hint.

This project has received funding from the European Union's Horizon 2020 research and innovation program through grant agreement 801101.

- Persistent hints are retrieved and set when an object is opened. The persistent hints are valid for the lifetime of the object. An example of a persistent hint is the object lifetime which can be used as one of the factors in Service Level Agreements (SLAs).

Hints are closely linked to IO optimisations. The current version of MIO focuses on designing and implementing the proposed framework as described in Figure 7, including data structures and APIs for hints. Based on the discussions with Maestro partners about the Maestro core library and its applications, we choose 2 hints, *MIO_HINT_OBJ_WHERE* and *MIO_HINT_OBJ_HOT_INDEX*, to demonstrate the benefit of the Guided IO framework and its usage. *MIO_HINT_OBJ_WHERE* allows users to influence the choice of a pool according to performance requirements, *MIO_POOL_GOLD, MIO_POOL_SILVER or MIO_POOL_BRONZE. MIO_HINT_OBJ_HOT_INDEX* passes information on how frequently the object will be accessed to MIO and MIO internally decides, which pool to store the object according to the hotness value. Section 3.3 will explain it in more details using the HSM example. MIO also defines 2 system level hints, *MIO_HINT_SYS_HOT_OBJ_THRESHOLD* and *MIO_HINT_SYS_COLD_OBJ_THRESHOLD* to set the thresholds of object access frequency for hot and cold objects.

```
/**
 * A hint is represented as a pair of key and value. A set of hints
 * are stored in a map.
 */
struct mio_hints {
    struct mio_hint_map mh_map;
};

/** 3 scopes of hints are defined for object, key-value set and system respectively. */
enum mio_hint_scope {
        MIO_HINT_SCOPE_OBJ,
        MIO_HINT_SCOPE_KVSET,
        MIO_HINT_SCOPE_SYS
};

/** 2 types of hints. */
enum mio_hint_type {
        MIO_HINT_SESSION,
        MIO_HINT_PERSISTENT
};
```

Table 8 MIO Hint Data Structure

3.2 Guided IO APIs

The Guided IO interface provides hint APIs as shown in Table 9 allowing applications to set or query the values of a set of hints or each individual hint. A *struct mio_hints *hints* parameter is added in *mio_obj_create()* to allow applications to set hints for storage pool selection as shown in Table 10.

This project has received funding from the European Union's Horizon 2020
research and innovation program through grant agreement 801101.

```
/**
 * mio_obj_hints_set() sets new values for the hints of the object
 * handler associated with object.
 * mio_obj_hints_get() retrieves object's hints.
 *
 * As potentially there may be many hints for an object, the
 * argument hints allows us to set or get multiple hints in one
 * single call.
 *
 * @param obj Pointer to the object handle.
 * @param hints Hints to set for an object.
 * @return 0 for success, < 0 for error.
 */
int mio_obj_hints_set(struct mio_obj *obj, struct mio_hints *hints);
int mio_obj_hints_get(struct mio_obj *obj, struct mio_hints *hints);

int mio_obj_hint_set(struct mio_obj *obj, int hint_key,
                     uint64_t hint_value);
int mio_obj_hint_get(struct mio_obj *obj, int hint_key,
                     uint64_t *hint_value);

int mio_sys_hint_set(int hint_key, uint64_t hint_value);
int mio_sys_hint_get(int hint_key, uint64_t *hint_value);
```

Table 9 MIO Hint APIs

```
/**
 * Create an object with identifier 'oid'. See mio_obj_open() above for
 * notes on object identifier. Note that an internal key-value set will be
 * created when creating an object, offering a convenient mechanism for
 * applications to store and query customized object attributes.
 *
 * `pool_id` and `hints` together decide which pool to store the object.
 * When `pool_id` isn't NULL, the object will be created in the explicitly
 * specified pool. When `pool_id` is set to NULL, the pool is chosen
 * according to the `hints`. MIO currently offers 2 kinds of hints for pool
 * selection: (1){MIO_HINT_OBJ_WHERE, MIO_POOL_GOLD|SILVER|BRONZE}
 * allows users choose a pool according to performance requirement.
 * (2){MIO_HINT_OBJ_HOT_INDEX, hotname} gives MIO information on how
 * frequent the object will be accessed and MIO then decides which pool to
 * store the object. See example in examples/mio_hsm.c.
 *
 * @param oid The object identifier.
 * @param pool_id The pool where the object is stored to.
 * @param hints Hints about which pool to store the object.
 * @param[out] obj Pointer to the object handle which can be used can be
 * subsequently used to access the object until the object is closed. Note
 * that the object handle must be pre-allocated by application.
 * @return 0 for success, < 0 for error.
 *
 */
int mio_obj_create(const struct mio_obj_id *oid,
                   const struct mio_pool_id *pool_id,
                   struct mio_hints *hints,
                   struct mio_obj *obj, struct mio_op *op);
```

Table 10 MIO Object Creation API with Hint

This project has received funding from the European Union's Horizon 2020
research and innovation program through grant agreement 801101.

3.3 Mini-HSM Example

As the number of objects stored in an object store system grows, it will at some point be necessary to migrate objects into a different tier. This is facilitated by the HSM (Hierarchical storage management) depending on the hotness of an object. Hot objects are moved into a better performing pool and cold objects into a lower performing pool that provides higher capacity. How to identify hot and cold objects is difficult. Object access frequency provides with an index showing how active an object is during a specified period. The object access frequency is captured by the MIO telemetry collector/processor and is stored as part of the object attributes (*struct mio_obj::mo_attrs::moa_stats*). The current version of MIO determines if an object is hot (or cold) by simply examining whether the access frequency is higher (or lower) than *MIO_HINT_SYS_HOT_OBJ_THRESHOLD (or MIO_HINT_SYS_COLD_OBJ_THRESHOLD)*, a parameter that is set by the application.

The Mini-HSM example makes use of the MIO multi-pool feature to create and access (read or write) objects in different pools (for example, pools of NVM, SSD and HDD tiers). It is also shipped with a workload generator to simulate object access pattern. While the workload generator is running, object access frequency is collected and used as the object's hotness as explained above.

HSM often needs to move objects between pools of different storage tiers. Typical steps to move an object when combining MIO guided IO are: (1) Query hotness of an object by calling MIO guided IO API (*mio_obj_hint_get() with hint's key MIO_HINT_OBJ_HOTNESS* and *the object's handler*). (2) The hotness can be used as the hint when creating a new object to which the original object will be moved. MIO internally decides which pool to create this new object based on the object's hotness hint. Hot objects will be created in a better performance pool, while a cold object is created in a lower performance pool; (3) The old object is copied to the new object if a different pool is selected for the new object. Currently the thresholds for hot and cold objects are set as system level hints as explained above.

The screenshot in Figure 9 shows the steps of HSM example running on the Sage cluster at Jülich Supercomputing Centre (JSC) [2]. 3 pools as shown in Figure 8 are used: the first pool is constructed using NVME devices with a 128-bit ID *0x6f00000000000001:0x46d*, while the second pool with ID *0x6f00000000000001:0x47a* is built with SSD devices and the third pool with ID *0x6f00000000000001:0x487* is configured with HDD devices. The third pool is configured as the default pool for objects.

```
[wu5@client-24 tests]$ hctl status
Data pools:
    # fid name
    0x6f00000000000001:0x46d 'tier1-nvme-p1'
    0x6f00000000000001:0x47a 'tier2-ssd-p2'
    0x6f00000000000001:0x487 'tier3-hdd-p3'
```

Figure 8 Sage Cluster Pool Settings

Figure 9 shows the screenshot of the running Mini-HSM. The Mini-HSM emulates common commands of an HSM system: creating objects, reading/writing objects and moving objects between tiers. To demonstrate how to move objects according to object hotness hint, we also add a few commands to display object's pool, set and get hotness thresholds for hot and cold objects and generate emulated IO workload. In the simple example, we first create 5 objects and populate them with data. At the end of the IO workload command (*workload*), the hotness for each object is displayed. Using the command *show*, we can clearly see that all objects are stored in the HDD pool. We also run the command *move* on 3 objects: the hottest object (*object 4*)t is moved to the pool with the best performance (NVME pool); the cold object (*object 0*) doesn't move as the

This project has received funding from the European Union's Horizon 2020
research and innovation program through grant agreement 801101.

object is created at the default HDD pool which is the worst performing pool, the *object 2's* hotness is between cold and hot thresholds and is called warm object in MIO, it is moved to one of the pools with medium performance, in our case it is the SSD pool. The pools for each object can be verified running the command *show.*

```
[wu5@client-24 tests]$ ../examples/mio_hsm -s 4096 -c 10 -o 1:62345678 -n 5 -y ./mio_config_hsm.yaml
hsm> help
 actions:
    create
    write <index> <number of objects> <number of blocks>
    read <index> <number of objects> <number of blocks>
    move <index> <number of objects>
    show <index> <number of objects>
    set_hot_thld <hot object threshold>
    set_cold_thld <cold object threshold>
    get_thlds
    workload
hsm> create 0 5
[hsm_create_objs] create 5 objects ...
hsm> write 0 5 10
WRITE Time: 1 secs 19156 usecs
hsm> workload
OBJ0 Hotness: 10
OBJ1 Hotness: 150
OBJ2 Hotness: 210
OBJ3 Hotness: 340
OBJ4 Hotness: 600
hsm> show 0 5
Object     POOL_ID
[Object 0] (6f00000000000001:487)
[Object 1] (6f00000000000001:487)
[Object 2] (6f00000000000001:487)
[Object 3] (6f00000000000001:487)
[Object 4] (6f00000000000001:487)
hsm> set_hot_thld 500
hsm> set_cold_thld 50
hsm> move 4 1
hsm> move 0 1
Object stays in the same pool, not moving :)
hsm> move 2 1
hsm> show 0 5
Object     POOL_ID
[Object 0] (6f00000000000001:487)
[Object 1] (6f00000000000001:487)
[Object 2] (6f00000000000001:47a)
[Object 3] (6f00000000000001:487)
[Object 4] (6f00000000000001:46d)
hsm>
```

Figure 9 Mini-HSM Example Using MIO Guided IO APIs

4. Conclusion

We have described the final APIs of MIO used for mapping persistent storage resources behind the Maestro Middleware and Guided I/O as well as its implementation and validation using a simple HSM. The storage and memory integration has been achieved with the help of MIO (with Motr as the backend object store) - with the focus being on persistent storage. It is to be noted that MIO could also be implemented for integration of other backend object stores such as Ceph, which will be discussed separately as an addendum to D6.4.

Demonstrations alongside Maestro applications and workflows will be covered in the WP6 deliverables.

This project has received funding from the European Union's Horizon 2020
research and innovation program through grant agreement 801101.

5. References

[1] MIO source code. https://github.com/Seagate/cortx-mio

[2] Sage2 Project. https://sagestorage.eu/

[3] Cortx-motr. https://github.com/Seagate/cortx-motr

This project has received funding from the European Union's Horizon 2020
research and innovation program through grant agreement 801101.