Percipient StorAGe for Exascale Data Centric Computing

FETHPC1 - 671500

WP3 Services and tools

D3.5 HSM for SAGE: Validation Readiness Report

SAGE_WP3_HSM_for_SAGE_Validation_Readiness_v1.0

Scheduled Delivery: 01.03.2017
Actual Delivery: 01.03.2017

Version 1.0

Responsible Partner: CEA

Revision history:

Date	Editor	Status	Version	Changes
04.01.2017	Thomas Leibovici	Draft	0	Work in progress
23.01.2017	Thomas Leibovici	Draft	0.1	Initial version
27.01.2017	Thomas Leibovici	Draft	0.2	Included comments from Philippe Deniel
08.02.2017	Thomas Leibovici	Draft	0.3	New composite layout API. Included comments from J.C. Lafoucrière
10.02.2017	Thomas Leibovici	Review	0.4	Ready for SAGE internal review
01.03.2017	Thomas Leibovici	Final submission	1.0	Final edits before submission

Authors
Thomas Leibovici (CEA), Henri Doreau (CEA), Philippe Deniel (CEA), J.-C. Lafoucrière (CEA), Sining Wu (Seagate)

Internal Reviewers
Nikita Danilov (Seagate)

Acknowledgements

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 671500

Glossary of Acronyms

Acronym	Definition
D	Deliverable
DRS	Document Review Sheet
EC	European Commission
FDMI	File Data Manipulation Interface (MERO component)
FOL	File Operation Log (MERO component)
HPC	High Performance Computing
HSM	Hierarchical Storage Management
I/O	Inputs and Outputs of computation (data)
IT	Information Technology
LSTM	Long Short-Term Memory
NN	Neural Network
NVRAM	Non-Volatile Random Access Memory
NVMe	Non-Volatile Memory express
RNN	Recursive Neural Network
SAGE	Percipient StorAGe for Exascale Data Centric Computing
SSD	Solid State Disk
WP	Work Package

Table of Contents

List of Figures

List of Tables

1. Executive Summary

Hierarchical Storage Management (HSM) makes it possible to build cost-effective storage systems that take advantage of various storage technologies: it combines the performance of expensive storage technologies, and the large capacity that can be afforded using cheaper storage. WP3.1 aims to implement an HSM solution that can manage deep storage hierarchies at extreme scales.

Previous deliverable D3.1 described the overall architecture of the HSM solution. It identified the main components that have to be implemented in the SAGE project, and the topics that should be studied.

This second report about HSM summarizes the detailed design of these components, the progress of their development, and the possible future improvements.

Partner	Contribution in Person Months
CEA	13PM

Table 1 Partner Contributions for Deliverable

2. Introduction

Hierarchical Storage Management (HSM) is a key feature of SAGE, which aims to implement an integrated support of deep storage hierarchies. Thanks to this feature, the SAGE platform will be able to handle multiple storage technologies from NVMe to archive disks, and combine their strengths to provide both a performant and high-capacity storage.

The goal of WP 3.1 is to design and implement Hierarchical Storage Management for SAGE.

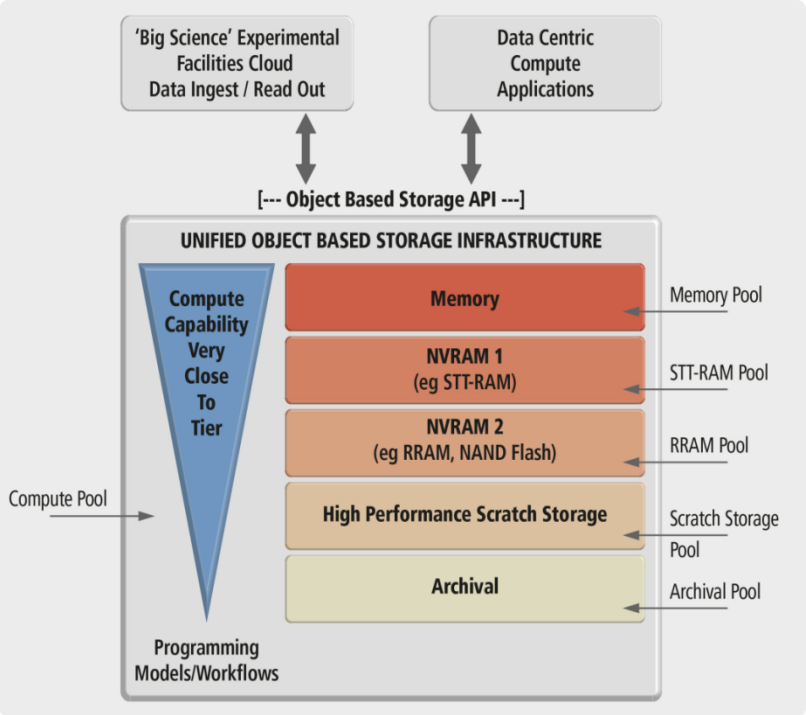

Figure 1: SAGE architecture consists of multiple storage levels

In previous deliverable D3.1, we presented the targeted architecture for the HSM solution in SAGE. We identified 3 main components that had to be implemented:

- A pool selection mechanism: this component is to determine the most suitable technology to store a given data. According to device characteristics, and application requirements in terms of I/Os, it will select the right storage "pool", which is a group of device of a given technology.
- A copy mechanism: this component must be able to copy data between storage levels (or tiers). This data migration must be made transparent to user applications, so it doesn't affect their working, and do not require to change their I/O calls.

- A policy engine: this is the coordinator of the HSM solution. According to the events it collects from the storage system, and policies defined by a system administrator, the policy engine triggers data movements between pools with a goal to optimize the location of data to speed-up application accesses.

This document presents the detailed design and the current progress in the development of these components.

3. Storage Pool Selection

One of the issues of Hierarchical Storage Management is to place data in the appropriate tier according to application needs (in terms of bandwidth, latency, data lifetime...).

This placement decision can occur at different steps of file's life:

- When an object is first written by an application: HSM must select the appropriate storage tier to write data to.
- When an object is moved from one tier to another (policy engine decision): HSM must select the appropriate target tier depending on file characteristics, and expected future accesses.
- When the object is later accessed by an application: HSM must select the appropriate storage tier to access the data.

This section describes the component that has been designed and implemented as part of WP 3.1 to select the most appropriate storage technology to match application needs.

3.1. Principle

HSM storage tiers are implemented in MERO as “storage pools”.

WP2 will provide an interface to select the pool where an object is to be written by specifying a pool identifier (e.g. `m0_object_create(obj_id, …, pool_id)`).

The goal of the HSM pool selection mechanism is to determine the appropriate pool identifier to be passed to MERO for creating an object. It is determined by correlating pool characteristics and application requirements for accessing the data.

Inputs for the pool selection mechanism are:

- **Pool characteristics:** media latency, bandwidth, preferred block size, I/O rate, capability to serve random and sequential I/Os, overall volume...
    - Some of these characteristics can be configured by the storage vendor or the system administrator, according to the know characteristics of devices in the pool.
    - Some characteristics can be extracted from MERO configuration database.
    - Some values can reflect the overall state of the system, and can be updated dynamically. For instance: space used in the given pool, current I/O load, ...
- **Application hints:** information about the way the application will access the object: read/write rate, I/O size, access pattern (e.g. random read, sequential write...), expected object size, file lifetime, etc. Application hints can be partially specified:

application is not obliged to specify all aspects of the I/O pattern. In this case, the pool selection mechanism must still select the pool that best matches the specified hints, or it can select a default pool if application hints are too incomplete to make a decision.

- **Predictions:** a possible enhancement is to complete application hints by a prediction of I/O patterns based on past application runs.

The output is:

- An identifier of the selected pool where the object should be located.

3.2. Design

3.2.1. Pool specification

Pool characteristics are described as a set of tuples `{key, comp_op, value}` that define ranges of acceptable values for I/O operations in this pool.

Example:

```
io_rate <= 1000
io_size >= 1MB
bandwidth <= 100MB/s
latency=1ms
write_random=0
obj_size < pool.space_avail
```

The set of keys is arbitrary, which makes the pool selection mechanism very flexible and expandable with new criteria.

Values can refer to system metrics, like "pool.space_avail" in the example above. Such values are maintained by the pool selection system, or queried to external components (operating system, storage system, …).

A pool specification defines a region in a multi-dimensional space:

- Each key is a dimension (an axis) of this space.
- The associated comparison operator and value define the range of acceptable values in this dimension.

The mechanism is fully configurable. It reads a configuration file that contains the specification of all pools in the system, and various parameters about the matching algorithm (see 3.2.4 and 3.2.5).

3.2.2. Application hints

Application hints are sets of `{key, value}` that describe the expected access pattern.

Example:

```
io_rate=10000, io_size=4KB, random_write=1, bandwidth=1MB/s,
access_time=1day, life_time=1year
```

The set of keys is also arbitrary, but only the keys that are present in the pool specification are taken into account to make the pool selection.

This tuple can be considered as a vector. It defines a point in a multidimensional space.

3.2.3. Pool selection

The goal is to select the pool with the best match to the vector of application hints.

The algorithm consists in selecting the pool with the least distance to the hints vector. As the vector may not specify all the dimensions of the pool characteristics, the distance should be computed as a projection in the space on the subset of dimensions specified by the application.

The selected norm to compute this distance will strongly impact the matching mechanism. It should be made configurable to allow the system administrator to drive the pool selection behaviour. The choice of this norm is discussed in next section.

In the example above, application hints have 3 keys in common with the pool specification: *io_rate*, *io_size*, and *bandwidth* so the distance will be computed in a 3D space. The resulting distance in quite high in this case: 9000 in the *io_rate* dimension, 1020K in the *io_size* dimension, 0 (zero) for *bandwidth*.

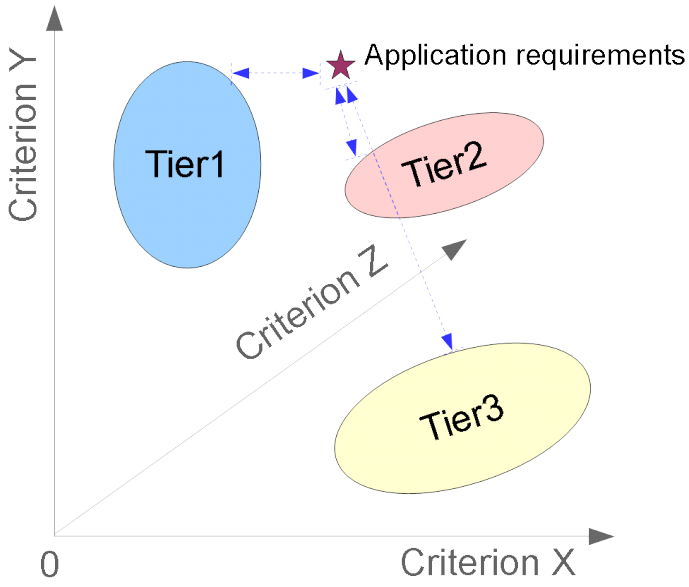

Figure 2 - The library determines the tier that best matches application needs by computing the distance to each tier

3.2.4. Norm choice and parameters

The proposed mechanism consists in computing a distance between a vector (of hints) and a region of space (pool characteristics). For this we need to choose a norm.

There are many possible choices and parameters:

- Each dimension can be associated to a different weight to control the impact of the criterion.
- Range of values can be normalized: values for a given key are normalized across all pool specifications. A range of [min_value ; max_value] is mapped to [0 ; 1] so the distance is always between 0 and 1 – except if the hint from application exceeds the range of values from pool specification. For example, if the specification of 2 pools are ranges [0; 5000] and [5000; 10000], they will be normalized to [0; 0.5] and [0.5; 1]. Then if an application specifies a hint value of 15,000, it will be normalized to 1.5.
- The computed distance between two values in a dimension or multiple dimensions depends on the chosen norm. There is a wide variety of norms. In the next steps, we should study which ones are the best suited for criteria we evaluate.

The best choice may be different for each dimension (e.g. logarithmic for object sizes, binary with a weight for random/sequential I/O patterns…). As a result, the behaviour for each criterion should be made configurable, to achieve a maximum flexibility.

3.2.5. Other parameters

In the design considerations above, pool matching strategy is permissive: even if application hints match no pool specification, the selected pool is the closest one.

In some cases, we may want to strictly forbid a given I/O pattern on a given pool, even if the defined norm designates this pool as the best match.

For example, it is impossible to perform random write on a magnetic tape (tapes are written sequentially). In this case, we would like to enforce the pool criterion. For example:

```
random_write=0:enforce
```

3.3. Implementation

This pool selection mechanism has been implemented as a stand-alone library written in C. This library can then be integrated to any component or SAGE API to benefit from this feature.

This library has been called “fuzzy match” as it allows a partial matching of application hints to the closest pool definition.

The implemented algorithm is very light, so the pool selection can be called in the I/O process without impacting access performances. This section gives more details about its implementation.

```
# Source files of the implemented library:

       -- src
          |-- Makefile.am
          |-- lib
          |   |-- Makefile.am
          |   |-- common.c
          |   |-- fuzzy_match.h
          |   |-- load_cfg.c
          |   |-- match.c
          |   `-- match_engine.h
          `-- tests
              |-- Makefile.am
              |-- hsm.cfg
              |-- test.c
              |-- test.cfg
              `-- xyz.cfg
```

Figure 3 - Source files of the pool matching library

3.3.1. Interface

The library implements and provides two main calls (library calls are prefixed by “fm” that stands for “fuzzy match”):

- **fm_init**: Initializes the library by reading a configuration file. This configuration file contains the pool specification. The interface of this function is represented below.

```
/**
 * Load the list of region definitions from a configuration file.
 * @param[in]  cfg   Path to the region definition file.
 * @param[out] hdl   Handle to the matching resources.
 * @return 0 on success, -errno on error.
 */
int fm_init(void **hdl, const char *cfg);
```

Figure 4 - Definition of function "fm_init"

- **fm_match**: matchs application hints and returns the name of the pool that best matches these hints. The interface of this function is represented below.

```
/**
 * Match a coma-separated list of criteria to the closest region.
 * @param[in] hld  Handle to the library resources given by fm_init.
 * @param[in] str  Input string is of the form "x=1, y=2, z=24".
 * @return The name of the closest region.
 */
const char *fm_match(void *hdl, const char *str);
```

Figure 5 - Definition of function "fm_match"

3.3.2. Pool specification

Pool characteristics are defined in a configuration file which is read at library initialization. For the initial version, the chosen format for this configuration file is the one of “ini” files (like *git* configuration for example). This format has the following advantages:

- This format is human-readable and easy to write.
- Most systems have a parsing library for this format. For example, on RedHat Linux systems, the “libini_config” package is part of the base installation.
- The parsing library is easy to use, which make its integration simple.

Of course, this format could easily be replaced by another format in a later version, to express more complex criteria.

The layout of the configuration file is the following:

- Each section of the “ini” file describes a given pool. The pool name is given between square brackets.
- In a section, each line specifies a criterion. The line format looks like: `crit_name = crit_value`
- Criterion value can be specified as:
  - A single value (possibly with a suffix): e.g. “10M”, “100MB/s”
  - A maximum value: “-10M” means “less than 10 millions”.
  - A minimum value: “+1G” means “more than 1 billion”.
  - A range: “10k-100M” is the range between 10k and 100M.

The following configuration example defines two pools: one pool of SSDs, and another pool of standard disks:

```
[pool0_ssd]
io_rate = 10k
sync = 1
random_io = yes
r_speed = 2GB/s
w_speed = 1GB/s
rewrite = 1
volume = 1-100GB

[pool1_disk]
io_rate = -1k
io_size = +512
r_speed = 100MB/s
w_speed = 80MB/s
rewrite = 1
volume = -4T
```

Figure 6 - Examples of pool specification

3.3.3. Pool matching

As mentioned above, there can be many ways to compute the distance between application hints and pool specifications.

The initial version implements a simple scoring method:

- If an application hint matches a pool criterion, it gets a positive score for this criterion (e.g. +1).
- If a hint doesn't match a pool criterion, the related pool gets a negative score (e.g. -0.3).
- No score is given (0) if no value is specified for a given criterion.
- The selected pool for a set of hints is the one that got the highest score.

As specified in 3.2.3, the library should be enriched to make it possible to use other matching methods (various norms).

Examples

In the examples below, the test program "match_pool" wraps the library we just described. We pass it a configuration file that contains the pool specifications (`hsm.cfg`), and we give it a string that represents some application hints. The program then displays the pool that better matches the input hints. In the ouput below, we also left some debug traces to understand how the pool selection decision was made:

```
> match_pool  hsm.cfg  random_io=1,io_rate=1000,r_speed=10M

REGION SCORES:
     pool0_flash: 1.90
     pool1_disk: 0.90
     pool2_archive: -0.30
best match: pool0_flash

> match_pool  hsm.cfg  random_io=0,io_size=10M,volume=1T

REGION SCORES:
     pool0_flash: -0.20
     pool1_disk: 2.00
     pool2_archive: 3.00
best match: pool2_archive
```

Figure 7 - Demonstration of pool selection

3.4. Status and Future Work

An initial version of the pool matching library for SAGE HSM is now implemented and gives its first results, as shown in the previous example (Figure 7).

In the next phase of the project, the work on this component will consist in:

- Adding new norms and matching parameters to the library (as described in §3.2.4).
- Evaluating various combinations of criteria, pool definitions, application hints, norms, matching methods, to determine the most relevant parameters resulting in the optimal pool selection.
- Integrating library calls to the application workflow. For instance, this could be implemented by a specific HSM call to create an object. This call would wrap the Clovis API.

4. Copy between Storage Tiers

4.1. Principle

One of the main aspects of Hierarchical Storage Management is to move data between storage tiers:

- If a data located on a fast device is no longer used, it should be moved to a slower storage to make room for newly accessed data in the fast tier.
- If a data is planned to be accessed with a heavy I/O pattern, it can be worthwhile loading it to a fast tier.

Although moving data from a source object to another is a basic operation that can be done through the common API (i.e. Clovis API), the difficult aspects for HSM implementation are:

- Making the data movement transparent to applications and users, even if they are currently accessing the data.
- Ensuring object consistency and change durability while the copy is taking place:
  - No object modification must be lost while an object or extent is moved from one tier to another.
  - Read must be directed to the right data. In particular, it must reflect previous writes.
- Limiting the impact on access performance, by reducing the need and the duration of critical sections that must be protected by a lock.
- Managing different granularities: moving parts of objects, single objects, or groups of objects.

4.2. Composite Layout API

In the initial phase of the SAGE project, we identified that the concept of “Composite Layout” was well suited to manage HSM data locations and movements.

The goal is to maintain a list of data fragments located in the different storage tiers (MERO pools). Composite layouts make it possible to maintain such information by representing an object as a combination of sub-objects – each one located in a different pool. These sub-objects are also called “layers”.

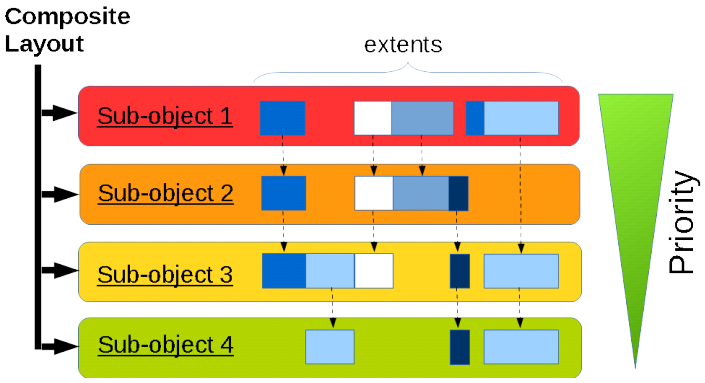

Figure 8: Composite Layout describes data location across tiers

In the next phase, WP2 and WP3 co-designed the API to manage the Composite Layouts.

This API defines the following data types:

- A composite layout associates multiple *sub-objects* to a single Clovis object.
- Each sub-object resides in a particular *pool*.
- Sub-objects are standard objects, and can be accessed as any other standard object in MERO.
- These sub-objects are ordered: when processing a read or write operation, Clovis will look for data available in sub-objects according to their order in the list.
- Two masks are associated to each sub-object. These masks are specified as a list of extents (data ranges). Each extent is described by a start offset and end offset.
    - A *write mask* (represented as a list of extents): it indicates the data ranges that can receive data in the sub-object.
    - A *read mask* (represented as a list of extents): it indicates the ranges of valid data that can be read by applications.

- To process **write** operations, Clovis should write to the first sub-object(s) whose the write-mask matches the written data range. Newly written extents should be added to the layer *read mask* by the Clovis write operation. If the *write mask* of a layer partially matches the written range, the matching range is written and the rest of the data range is searched in next layers.
- When handling **read** operations, Clovis searches the matching extent(s) starting from the first sub-object of the list. Only the readable extents are taken into account, according to the *read mask* attached to each sub-object. If the read mask of a layer partially matches the read range, the matching range is read and the rest of the data range is searched in next layers.

The API provides the following operations to handle these entities:

- **Adding** or **removing** a layer (a sub-object) in a composite layout. In particular, it must allow inserting a layer between two existing layers.
- **Iterating** on the layers of a composite layout.
- **Adding** or **removing** extents from read-mask and write-mask of a layer.
- **Searching** for extents that match a particular range of data in the read-mask or write-mask of layers.
- **Merging** or **splitting** extents in those masks.

4.3. How HSM Operations use Composite Layouts

4.3.1. Ordering of layers

As described previously, sub-objects in Composite Layouts are ordered. This ordering indicates the priority for accessing the sub-objects when an application reads or writes to a composite layout.

This priority number can be used by HSM to reflect the localisation of data in the storage hierarchy, and specify the access priority between multiple objects located in the same pool.

HSM must properly control the ordering of the layers depending on their location in the storage hierarchy, and the mandatory requirements to be satisfied, e.g. an application must read the last data it wrote. In this section, we describe the parameters that must be taken into account to determine a layer position in the layer list to match these requirements.

With HSM, a given storage tier somehow works as a moving walkway: the new data progressively replaces the old data while older data is migrated to slower tiers. To manage that, new data needs to be created with a higher access priority i.e at the head of the layer list, while previously created data is in next positions of the layer list and can be progressively migrated to slower tiers. This way, applications always access the most recent data which is the first in the layer list. Thus, the position in the layer list must reflect the age of the sub-object data. To achieve that, HSM will maintain a **generation number** associated with each sub-object of a composite layout. This number indicates the age of the data the object contains. HSM uses this generation number to determine the position a sub-object must have in the layer list.

Nonetheless, ordering layers based on the age of the data is not enough. If the same data is replicated between several tiers, applications should preferably access the data which is located on the fastest one. To manage that, HSM needs that an order is defined on storage tiers. This order defines the storage hierarchy and is typically from the lowest to the highest

performance. This order can be specified by the system administrator, or it can be automatically determined by the system depending on object type, or a planned access pattern. The important thing to properly manage HSM priorities is that this order remains constant for a given object, and doesn't change in the object lifetime.

Once the tier ordering is defined, we associate a **priority** to each tier of the storage hierarchy, starting from the lowest level, so the fastest tier is associated to the highest priority (see example in Table 2). Notice that the resulting numbering is the opposite of the usual tier indexing.

Table 2 - Example of tier priorities in the SAGE platform

Tier	H/W Tier	Storage Technologies	Storage Interface	Storage Enclosure	Tier priority value
NVRAM	1	NVRAM (NVDIMM / new tech)	CPU/PCIe NVMe		4
SSD	2	NVRAM – FLASH 2.5" SSD	SAS		3
Disks	3	Fast 3.5" SAS disk	SAS	 Fast disk	2
Archive	4	Cost optimised 3.5" SATA disk	SATA (over SAS)	 Archive	1

To order the layers of a composite layout, HSM cleverly computes a "priority" which is the combination of the 2 parameters descrobes above: **generation number** and **tier priority**, as illustrated on Figure 9. With a priority value encoded on 64 bits, the tier number could be

stored on 8 bits, which allows managing up to 256 storage tiers, which seams enough for all imaginable storage architectures. The generation number will be stored on the remaining 56 bytes, which allows managing 72 millions of billions of object versions. The priority of the generation number is more important than the pool priority: priority is to read the most recent data – rather that the data located on the fastest tier. So the generation number is stored in the high weight bits of the priority value. In the rest of this document, we will note priorities as dot-separated values: <generation>.<tier_prio>.

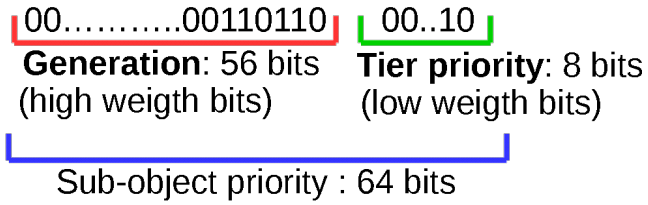

Figure 9 - Definition of sub-object priorities for HSM

Finally, HSM will order the sub-objects according to this priority number. First layers of the list are associated with higher priority values, and last layers have lower priority values.
By ordering entries like this, we ensure:

- User applications always access the most recent data, wherever the data is located in the storage hierarchy.
- If a data is present in multiple tiers, the application will access the one located in the faster tier.

HSM maintains this priority value per object (generation+tier priority), possibly in a specific index “hsm_prio”. This index associates a given object identifier (fid) to an HSM priority value. This value can also be completed with extra metadata about this particular sub-object (creation date, statistics about HSM movements...)

4.3.2. Creating an object in a pool

One of the requirements of HSM feature is to be able to locate a data in a particular storage tier (pool).

This can be achieved by implementing a specific HSM call to do so.

This call should:

1. Create a new composite layout.
2. Create an object in the desired pool.
3. Add the object to the composite layout using the “add layer” call of the composite layout API.

4. The write-mask of this object is set to [0-inf), so all newly written data is directed to the selected pool. The read-mask is initially empty. It will be extended when applications write to this object.
5. The priority of this sub-object is computed according to the pool position in the storage hierarchy, as explained in §**Error! Reference source not found.** (**Error! Reference source not found.**). This value is stored in the *hsm_prio* index.

User application can then use standard Clovis calls to access this composite layout. In this case, any access by the application is directed to the only sub-object of the composite layout.

The extent list of this sub-object is updated to take newly written data into account.

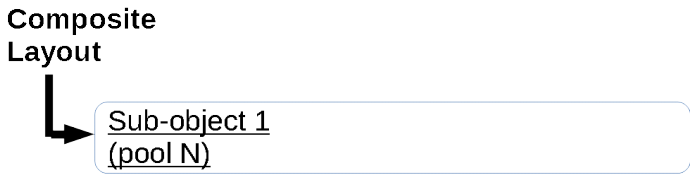

Figure 10 - Just created composite object in a particular pool
A sub-object is created in the selected pool (pool N). It is initially empty (no readable extent).
The composite layout consists of this single layer.
All subsequent I/Os are directed to this sub-object,
until the object (or a part of it) is moved to another tier.

4.3.3. Copying data to a lower tier

Principle

Copying data from one tier to another can be made with the granularity of an extent. If an extent is too big to be copied in a reasonable time, it could be split into smaller extents using the “split” operation mentioned in 4.2 (Composite Layout API). Max extent size for the copy should be configurable by the system administrator.

To prevent from loosing changes, we need to ensure an object or an extent is not modified while it is being copied from one tier to another. If it is modified, we need to keep track of changes, so they can later be applied in the lower tier.

To achieve this, the idea is to create a new sub-object (let’s call it O1) at the head of the list (with a higher generation number) where all new writes are directed, while the previous data (sub-object object O0) remains unchanged and is copied to a lower tier.

Once the copy completed, we can safely delete the source sub-object O0. If a write operation was done during the copy, it has been applied to the new sub-object O1 with the higher priority.

So, to prepare the copy operation from a fast tier to a slower one, the first step is to "freeze" the current version O0 of the data in the source tier.

Algorithm

HSM operation: copy from a fast tier (poolA) to a slower tier (poolB).

PoolA is assigned to tier priority A.

PoolB is assigned to tier priority B.

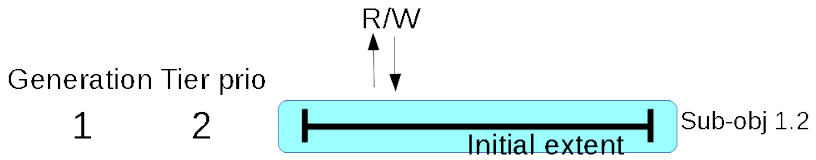

Figure 11 - Example of initial state

Composite object consists of a single sub-object in a tier of priority 2.
All application accesses are directed to this sub-object.

I - Prepare copy:

1. Retrieve the priority of the source sub-object (<Gsrc>.<A>).
2. To ensure the source sub-object won't be modified while we copy it to poolB, a writeable object must be present in previous layers. So, check if the source sub-object is the first writeable object of the layer list. If so, execute step 3 (In this case, it is expected that G*max*=G*src*). Else, retrieve the generation value of the first writeable object of the list (*Gmax)* . Skip next step (step 3) in this later case.
3. Create a new sub-object in the pool of the copy source (poolA) with priority <Gmax+1>.<A>. This object will receive all new write from user applications. This ensures the source sub-object is not modified while we read from it. This new layer is inserted in the list according to the computed priority value.
4. Source sub-object is tagged read-only, by empting its write-mask.

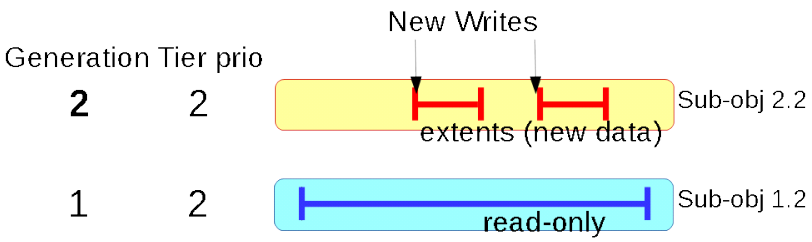

Figure 12 - Preparing copy-down

An object with incremented generation has been created in the same tier to receive newly written data.

Older data of previous generation becomes read-only.

5. Create a sub-object in the target pool (poolB). The priority of this sub-object is set to <Gsrc>.<B>.
   Skip this step if an object with this priority already exists in the Composite Layout. Note: if there is already another version (a different generation number) of the object in this pool, its data is not overwritten. Instead, a new object is created with the generation number *Gsrc*. Nonetheless, overlapping extents of older generations can optionally be removed from the target pool when the copy terminates (see Finalize step).

II – Copy extents:

For each extent of the source sub-object:

1. Copy the related data range from the source sub-object to the target sub-object.
2. Add the extent to the list of readable extents of the target sub-object (possibly merge it with contiguous extents).
3. Drop the source extent:
   a. If the caller wants to release space from poolA: remove the extent from the source sub-object. Read-mask of the source layer must be updated accordingly.
   b. If the caller still wants to to serve efficiently next read requests on the data range, it can keep the extent in the source tier Later, to determine if the extent can be removed from the layer, HSM can check if an object with the same or a higher generation number is present in other layers, and matching the given extent (must be readable).

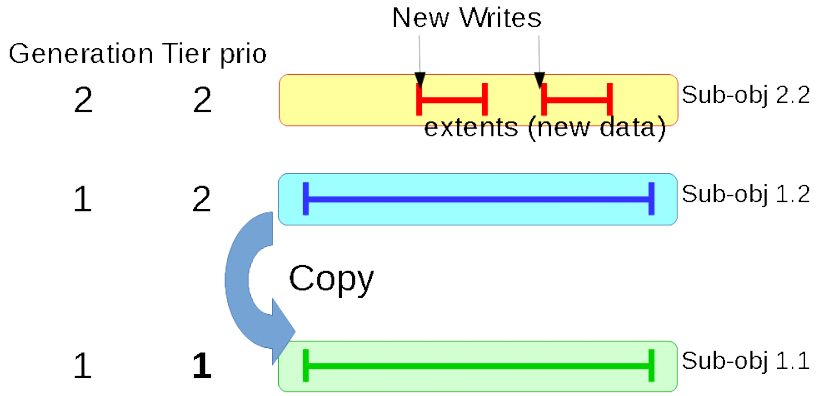

Figure 13 - Example of copy operation

A target sub-object has been created in the lower tier of priority 1.
Extents from source sub-object are copied to it.

III - Finalize:

1. If no readable extent remains in the source layer:
    a. Remove the source sub-object from the composite layout.
    b. Delete the source sub-object.
2. Garbage collection of older copies:
    a. If older versions of data were present in the target tier, previous overlapping extents can be removed. These older copies have a priority number <G>.<B> such as G < Gsrc. After cleaning overlapping extents, delete the related sub-objects if they no longer contain readable extents.
    b. We could also make it possible to keep older versions of data by keeping overlapping extents at this step. This behavior should be configuration-driven.

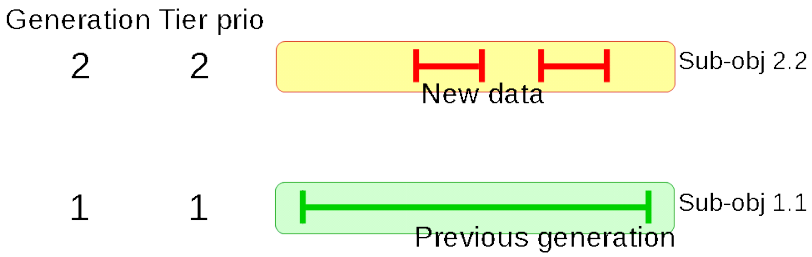

Figure 14 – Example: state after copy completion
Once the copy completed, the source sub-object can be deleted:
All its data has been copied to sub-object 1.1 and potential new changes have been made in sub-object 2.2.

The figure below illustrates what would happen next in case a new copy operation is triggered:

- A new sub-object is created in top tier with increased generation (3).
- A new sub-object is created in lower tier to receive the data of generation 2.
- Extents of generation 2 are set read-only and copied to the lower tier.
- Optionally, overwritten extents of generation 1 can be removed from sub-object 1.1.

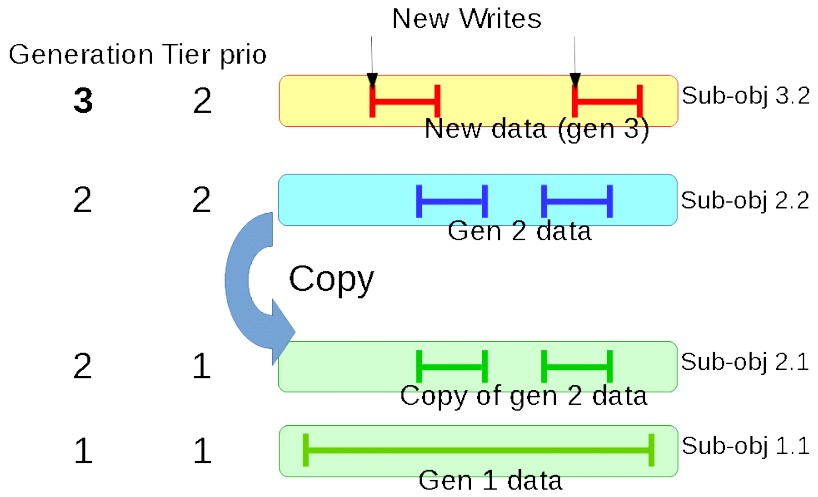

Figure 15 - Example of second copy operation

Configurable/policy-driven behaviors:

Notice that the following points are configurable independently for each copy operation:

- [Optional step III.1] Removing the source sub-object from source pool: if the data is likely to be re-read in the short term, it is worthwhile keeping it in a fast tier. Otherwise, it is better to drop this source copy.
- [Optional step III.2] Removing overwritten extents from previous generations of objects: the proposed mechanism makes it possible to keep previous version of object data, by keeping overwritten extents in older sub-objects. This enables using HSM for backup purpose. Otherwise, cleaning overwritten extents reduces the space consumption to the strict minimum.

4.3.4. Preloading data to a fast tier

As for the copy operation described in §4.3.3, we need to create a particular sub-object at the head of the layer list to receive newly written data from user applications, while other data is staged from the down tier.

HSM operation: copy data from a slow tier (poolB) to a faster tier (poolA).
PoolA is assigned to tier priority A.
PoolB is assigned to tier priority B.

I - Prepare copy:

1. Determine the source sub-objects where data will be read from. There can be several of such sub-objects if the data range to be copied covers multiple extents in

multiple sub-objects. Determine the max generation number of this set of sub-objects: *Gsrc_max*.

2. To keep track of new changes while we copy older data to poolA, a writeable object must be present in previous layers. So, check if there is a writeable object in the previous layer of the list (before all of the source layers). If not, execute step 3.
3. Get generation value for the first layer of the list (from *hsm_prio* index). Create a new sub-object in poolA (or an upper one) with priority <Gmax+1>.<A>. This object will receive all new write from user applications. This ensures the target sub-object won't be modified while we copy to it.
4. If it doesn't already exist, create a new sub-object in poolA with the priority <Gsrc_max>.<A>. This object will be the copy target where the data is staged to.

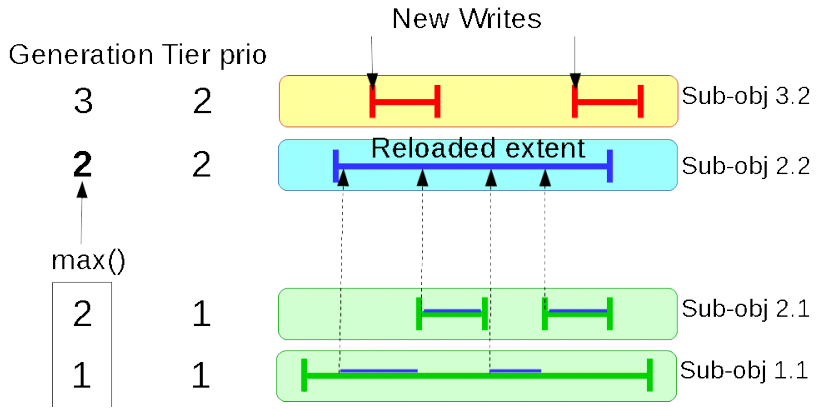

Figure 16 - Example of reloading data to a faster tier
A sub-object is created in the faster tier to copy older data to.
Its generation is max(source sub-oibjects).
The sub-object with higher generation (3) receives newly written data,
so there is no conflict between application writes and the copy operation.

II – Stage extents:

For each extent to be staged:

1. Copy the related data range from the source sub-object to the target sub-object.
2. We are sure nobody will read from it as long as the extent is not present in the target mask of readable extents.
3. Add the extent to the read-mask of the target layer (possibly merge it with contiguous extents).

4. [Optional] Remove the extent from the source tier. This may be desirable if the extent is will be modified in the short-term.

III – Finalize:

- If source object is empty (due to optional step II.3, all its extents have been copied and removed), the following operations are done:
    1. Remove the source sub-object from the composite layout.
    2. Delete the source sub-object.

Notice that the following point is configurable independently for each copy operation:

- [Optional step II.3] Removing the source sub-object from the "archive" tier: if the data is likely to be modified in the short-term and the administrator doesn't want to keep older versions of data, the source copy can be dropped from the lower tier to release space in it.

4.3.5. Releasing space from a storage tier

If space is needed in a storage pool, HSM can release space by deleting "archived" sub-objects. A sub-object (or a part of it) can be removed from a layer if all its data (or a more recent version of it) is present in other layers.

It is one of the policy engine role to maintain a list of these "releasable" extents in a given pool.

HSM operation: free the space for a given composite object in poolA.

1. Get the sub-objects located in poolA.
2. For each sub-object:
    a. For each extent in the sub-object: if the data of this extent is present in another layer, remove the extent from the sub-object.
    b. If no extent remain in the sub-object, remove the sub-object from the composite layout, and delete the sub-object.

4.4. HSM base functions

The previously described operations constitute base primitives to manage data in the multi-tiered SAGE architecture. They will be available to other SAGE components as the following sets of functions:

- **`hsm_create(pool, &fid)`:** create a new object in a given pool. Output is the identifier (fid) of the created object.

- **`hsm_copy(fid, [src_pool,] tgt_pool, extents, options):`** copy or move data from one pool to another. It can be used to move data from a fast tier to a slower one, or to stage data from a low tier to an upper one. This operation is transparent for other applications that would access the object during the copy:
    - The operation does affect the object identifier: its *fid* remains unchanged.
    - Applications can still read and write to the object while the copy is running. These accesses will be handled properly: accesses will not be blocked, data modifications will be taken into account, and read will reflect last written data.

  src_pool is optional. If it is not specified, the copy will read data from the faster tier where it is available.

  Extents is the list of data regions to be copied from the source tier to the target tier. O*ptions*:
    - **HSM_COPY** vs. **HSM_MOVE:**
        - HSM_COPY is the default behaviour: it results in preserving data in the source pool after the copy, which allows keeping fast access to the data even after copying to a down tier. In this case the source data can be later removed from the source tier using hsm_release() operation if space is needed in the tier. In the case of a staging operation, this option results in keeping an archived copy in the slow tier.
        - HSM_MOVE removes the data from the source tier after the copy completes. For an archive operation (copy to a lower tier), this results in freeing space in the source tier. This has an interest if the data is not expected to be reused in the near-future. For staging operations (copy to a faster tier), this drops the archived copy, which can have an interest if the data is about to be rewritten and we don't want to waste disk space keeping old data in the lower tier.
    - **HSM_KEEP_ARCHIVE:**
        - When copying data to a lower tier, this results in preserving previous versions of data, even if they are overwritten by newer data written to this tier. This results in higher space consumption, as multiple date versions are kept in the down tier. But this constitute a snapshot mechanism that allows retrieving the past state of an object.
        - If this flag is not specified, the data regions that where overwritten in a given tier are freed after a more recent version of the data has been copied to it.

- **`hsm_release(fid, pool, extents, options):`** this operation releases used space of the given *extents* in the specified *pool*. In order not to loose data changes, an extent cannot be removed from a pool if no copy of it (or new version of it) is stored somewhere else in the storage hierarchy. The following behaviors can be controlled through the *options* parameter:
    - **HSM_IS_UPTIER:** Allow removing an extent if a copy of it is present in upper tiers.
    - **HSM_IS_DOWNTIER:** Allow removing an extent only if a copy of its data is present in a lower tier.
    - **HSM_IS_NEWER:** Allow removing an extent if a new version of it is present in the tiers specified by above options (uptier or downtier). By default, if this option is not specified, allow removing an extent only if the same version of it is present in the designated tiers.

These functions can be called by applications, or other HSM components:

- They can be called by applications to explicitly pre-load data in a fast tier before an intensive I/O access phase, or to archive data in a slow tier after modifying it.
- They can be used by specific cache management systems, like the Semi-Persistent Cache Manager implemented by WP 4.2.
- They will be called by the HSM Policy Engine to trigger data movements according to admin-defined rules, or self-learned rules.

4.5. Status and Future Work

So far, the effort has been put on defining an API that provides a powerful and flexible way to manage data location in deep storage hierarchies: the Composite Layout API. Now this API has been specified, its effective implementation can start. Its detailed interface has already been defined, so other components of SAGE can start considering how to use it.

The “copy” feature of HSM relies on the Composite Layout API. At this stage, the implementation of copy mechanisms using this API have been designed in details, even if it needs to be completed by additional aspects like locking, to manage concurrency between applications and HSM mechanisms. Now the Composite Layout interface is specified, it becomes possible to start implementing the copy primitives. These mechanisms will be validated once the Composite Layout API is functional.

Once implemented these base HSM functions can be used by applications, cache managers, or HSM policy engine to manage objects stored across multiple storage tiers.

5. Policy Engine

5.1. Role

In the SAGE HSM architecture, the role of the Policy Engine is to trigger data movements between storage tiers.

The Policy Engine collects information from the system. There are multiple types of collected data:

- Data access information makes it possible for the Policy Engine to maintain usage statistics for the objects it monitors.
- Resource usage information indicates the Policy Engine if data movements are necessary, e.g. if a storage tier is full.
- Information from other components of a computing center, like a job scheduler. For example, this can lead it to preload data in a fast tier before a compute job is started.

This information constitutes the material to take movement decisions. Using this data, the Policy Engine must be able to determine the characteristics of the data movements: when data should be moved? What data should be moved? Where it should be moved from, where it should be moved to…

Decisions are made according to admin-defined rules, or automatically determined using prediction algorithms. Decision can also be influenced by applications.

The picture below summarizes the interactions of the policy engine with other components and actors of the SAGE environment:

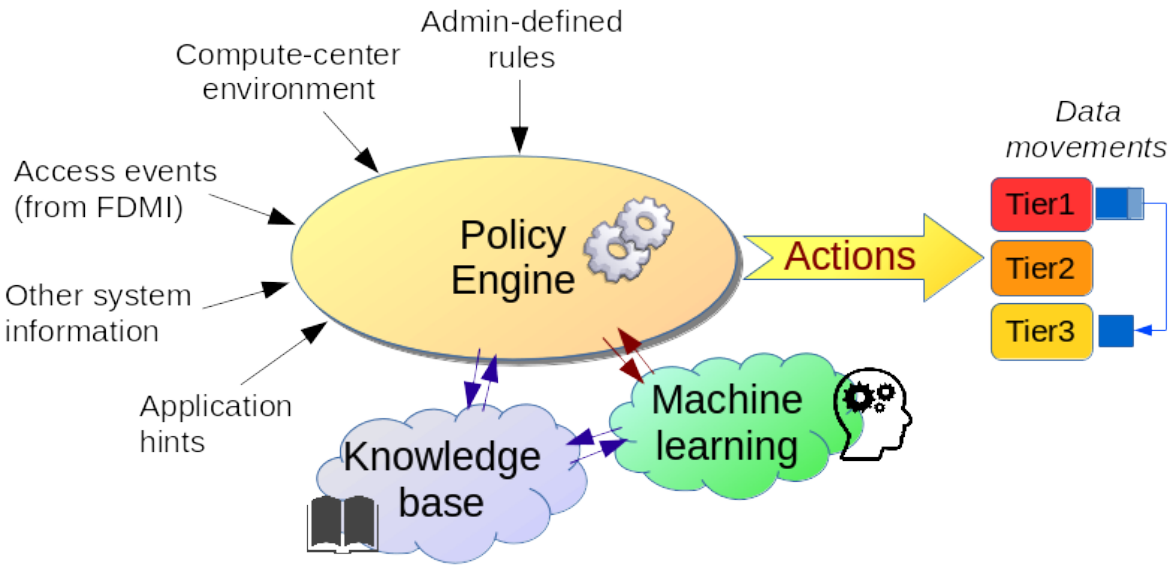

Figure 17 - Interactions between the HSM Policy Engine and other components

5.2. Design and Implementation

5.2.1. Adapting Robinhood Policy Engine

Working basis

The SAGE HSM Policy Engine can use the open-source software "Robinhood Policy Engine" [RHPE] as a working basis for the SAGE work. This software already implements similar features than the one listed in 5.1, for Lustre filesystems:

- It collects information about data accesses in Lustre using its "Changelog" feature.
- It precisely monitors resource usage in the file system by querying each Lustre server independently ("lfs df").
- It can trigger data movements between Lustre and an external storage, thus implementing a 2 levels Hierarchical Storage system.
- System administrator can specify conditions on file attributes to determine if an entry is eligible for such data movement.
- It can release disk space from Lustre servers when its usage exceeds a configurable threshold.

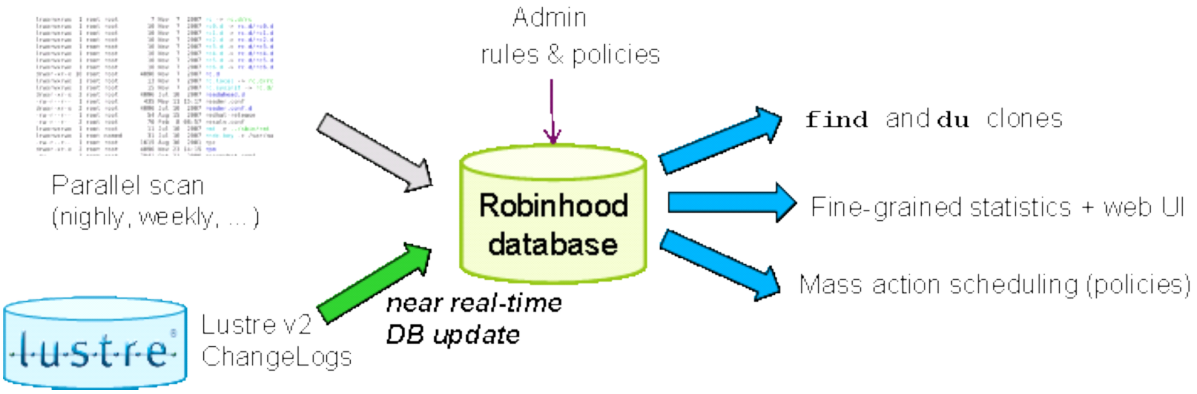

Figure 18 - Robinhood Policy Engine: big picture

Required adaptations

To satisfy the requirements of the SAGE project, Robinhood Policy Engine needs to be adapted to the MERO storage framework. The following changes have been identified as requirements for this evolution:

- Dropping the dependency to the POSIX namespace: Robinhood Policy Engine has been initially implemented for POSIX file systems. MERO does not implement a similar namespace, so Robinhood must be adapted to deal with that. In particular, it must be able to handle and address objects by their identifier, instead of using their path in a POSIX namespace.
- The access log MERO provides is significantly different from the information Lustre Changelogs provides. In MERO, this reporting is done by FDMI. So, an FMDI reader must be implemented in Robinhood to collect this access information from MERO. The processing of the collected information is also to be reconsidered as the reported information greatly differs.
- For now, Robinhood maintains POSIX attributes of entries (owner, access rights, size, access time, modification time…). MERO does not implement these attributes, but there is still the need to maintain object-specific information, like access times, containers… Also, applications can store arbitrary metadata to the key-value store MERO provides. A conceivable change is to make Robinhood manage an association between MERO objects and the related metadata, so it can manage these custom attributes.
- Robinhood used to implement POSIX-specific and Lustre-specific actions: *unlink*, *rmdir*, *hsm_archive*, *hsm_release*… Actions should be made configurable so Robinhood can trigger customizable actions like moving data between SAGE tiers.
- To query or receive information from external components like a job scheduler, new mechanisms are to be designed in Robinhood.

- Instead of handling Lustre striping information, Robinhood has to manage Composite Layouts described in 4.2.
- Instead of querying Lustre OST usage, it should query MERO pool usage.
- For now, Robinhood stores the metadata it maintains in a common relational database (MariaDB). Using MERO key-value store to save this information would be more appropriate to achieve the extreme performance which is targeted by the SAGE project.

The work on these topics is described in the following sections.

5.2.2. Managing custom attributes

A new mechanism of plugin has been implemented in Robinhood. A plugin is a dynamic library loaded at run-time, according to the policy engine configuration. To manage custom attributes, we completed these plugins to allow defining arbitrary attributes in them, in a way they are fully integrated to the Robinhood working, like any other currently managed attribute (e.g. like POSIX attributes):

- They can be referenced in the policy rules specification (in configuration), to specify the candidate entries to run a particular action (e.g. data movement).
- Their value is set when an entry is discovered (when scanning the storage system, or processing an access event about it). This requires the plugin also provides a specific function to retrieve the value of this attribute.
- The Robinhood database is automatically adapted to store this custom information.
- This information is available in robinhood reporting commands:
  - "`rbh-report -e entry_id`" : displays all attributes saved in robinhood database for a given entry, including custom attributes.
  - "`rbh-find -printf format`" supports displaying custom attributes using the syntax "`%Rm{plugin_name.attr_name}`"

Such an attribute is easy to define as a simple C structure defined in Figure 19. Figure 20 gives an example of custom attribute definition in a single line of C.

```
/** descriptor of specific info */
struct sm_info_def {
    const char      *user_name;     /**< Full name for user interface (config,
                                       display...) */
    const char      *db_name;       /**< Short name for db field */
    db_type_e        db_type;       /**< Type in database */
    unsigned int     db_type_size;  /**< Size for strings */
    db_type_u        db_default;    /**< Default value */
    cfg_param_type   crit_type;     /**< Type of criterion (size, count, …).
                                       * Indicates how the value should be parsed
                                       * or displayed.
                                       */
};
```

Figure 19 - Custom attribute type implemented in Robinhood v3.0

```
/** definition of a custom attribute: 'last_read' */
attr = {"last_read", "lstrd", DB_UINT, 0, {.val_uint = 0}, PT_DURATION};
```

Figure 20 – Easy definition of a custom attribute in a single line of C

Now this mechanism is implemented and fully functional, it can be used to store MERO-specific information that needs to be attached to objects. To complete this work, a MERO plugin is to be created. This plugin will define the information that needs to be maintained to manage SAGE HSM policies.

5.2.3. Triggering data movements

Robinhood already implements needed features to trigger movements between a Lustre filesystem and a storage backend, according to admin-defined rules.

To adapt this feature to MERO, the main work has been to make it possible to trigger other operations than Lustre's "hsm_archive", in particular to trigger the copy mechanisms that were previously described in chapter 4.

Completing the concept of plugin just described in §5.2.2, we made it possible for a plugin to implement custom actions, so they can be used in Robinhood Policies instead of previously "hard-coded" Lustre actions. An example of configuration is given in Figure 21. Additionally, we also made it possible to execute external commands by specifying a command line to call them.

```
declare_policy m0_copy {
      default_action = mero.copy ;
      …
}
```

Figure 21 - Example of policy definition using a MERO-specific "copy" function, provided by a "mero" plugin

Actions executed by policies have been made fully configurable, so the system administrator can specify what action is to be done, and specify custom parameters to these actions depending on objects characteristics: target pool, extent size, … These parameters are defined by configuration as an "action_params" section (see example in Figure 22).

```
m0_copy_rules {
      rule copy2disk {
            target_fileclass = objects_on_flash;
            action_params {
                target_pool = "disk";
                max_extent_size = 1G;
                extent_list = mero.unused_extents;
                …
            }
            condition {last_access > 1h
                        and last_copy > 1h}
      }
}
```

Figure 22 - Example of policy rule to copy data from one tier (flash) to another (disk)

5.2.4. Interacting with external components

The plugins mentioned in the two previous sections can communicate with external components, using specific APIs or communication. Thus, the two mentioned features (custom attributes and custom actions) can be used to interact with external components:

- Custom attributes can be used to query external components, and using some result of these queries as values in policy rules. For instance, we could imagine a SLURM plugin that would provide the priority of a compute job, like in this policy rule example:

  ```
  mero.pool == flash and slurm.job_priority > 10
  ```

- Custom actions make it possible to trigger external commands. This mechanism can be used to notify an external component about a particular event on an object. In this example, it advises the pNFS server (developed in WP 3.2) that the data is ready to be accessed in a fast tier:

  ```
  action = pnfs.notify;
  action_params {
        entry_id = mero.fid;
        event    = "data loaded";
        pool     = mero.top_pool;
  }
  ```

5.2.5. Collecting information about system activity

MERO implements a scalable publish-subscribe mechanism to collect information from storage servers, called FDMI (File Data Manipulation Interface). The principle of FDMI is represented by Figure 23. This generic framework can handle any type of information thanks to its self-described records. In its current implementation, it can deliver File Operation Log records (FOL), or records from Analytics and Diagnostics DataBase (ADDB).

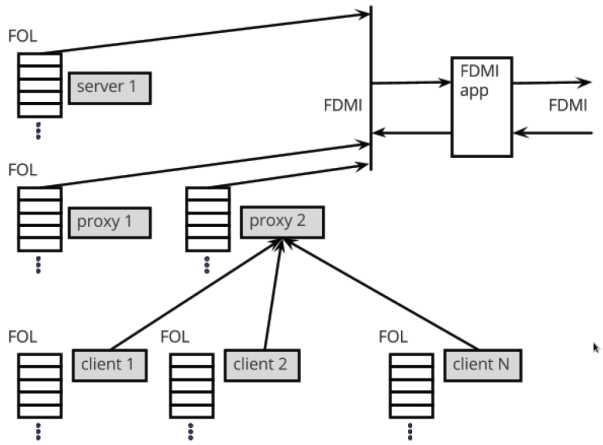

Figure 23 - FDMI scalable architecture

By subscribing to FOL records, the policy engine can receive information about accessed data. In particular, FOL records are emitted when extents are written, which make it possible to have a fine-grained tracking of data accesses in a pool, and even in objects.

Moreover, FDMI makes it possible to filter records on arbitrary criteria. So the policy engine could limit its scope to a particular set of objects, container, pool,...

This FDMI framework is currently at a development phase. Once completed, we can implement a FDMI reader for Robinhood so it can collect information about accesses in MERO.

5.2.6. Maintaining access information

To determine the source pool and target pool of HSM movements, and the ranges of data to be moved, the policy engine must have a precise knowledge of objects structure, and their access patterns. To determine it, the policy engine can rely on two sources of information:

- FOL records mentioned in previous section. They are expected to provide:
  - An object identifier indicates what object the operation affects.
  - A timestamp that indicates the time of an access operation.
  - A range of accessed data.
  - The type of operation (read, write...).
  - The more information they provide, the better: if the policy engine has more information about an entry, it can make smarter policy decisions.
- If the policy engine requires more information than provided by the FOL in order to take policy decisions about moving extents, it can get it by querying MERO using the

Clovis API. In particular, it can read the *Composite Layout* of an entry, which indicates the location of its data in storage tiers.

The policy engine maintains a "map" of accesses using its own representation stored in its database. To manage tiers of an HSM, a "temperature" counter could be attached to every extent of an object in order to identify the extents to keep in faster tiers and the ones eligible to migration to slower tiers.

5.2.7. Applications of Machine Learning and Deep Learning

In deliverable D3.1 we mentioned that Machine Learning algorithms could bring interesting features to Hierarchical Storage Management. In the recent years, there were great improvements in this fields, and great open-source frameworks are now available to anyone who needs to apply this technics: Scikit-learn [ScikitL] provides a large set of Machine Learning algorithms in python; Caffe [Caffe] and Torch [Torch] provide neural networks implementations to run Machine Learning.

As part of the work of this package, we have studied several solutions in the domains of Machine Learning and Deep Learning that could be applicable to HSM. In particular, sequence predictions could help predicting future events based on past sequences of events.

One application of Deep Learning that came to our attention is guessing a text just by reading its first letters. To achieve that, a Neural Network (NN) is designed to get a sequence of letters as its input layer, and its output is the guess of the next letters, as illustrated in Figure 24. This network is trained with a particular type of text (e.g. Shakespeare text, C code, Wikipedia articles…). Then it becomes capable of guessing a text just from reading the first few letters. [Karp15] illustrates the impressive results of Recursive Neural Networks (RNNs) to achieve that.

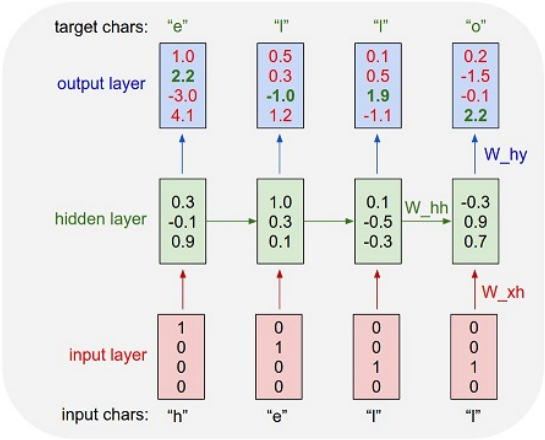

Figure 24 - Example of neural network to predict a sequence of letters

A conceivable application for HSM is to use access events as the alphabet, like in Figure 25. Applying this method, the neural network will be able to predict next accesses of an application, based on what it learned before.

ALPHABET: **A**: create, **B**: write, **C**: read, **D**: delete, …
TEXT: sequence of alphabet items = sequence of access events

Figure 25 - Example of "alphabet" to predict access sequences

As noted in [LSTM97] a particular kind of Recursive Neural Network called “Long Short-Term Memory” are especially effective at capturing temporal dependencies within time series and predict them. Although several models exist, they are all based on a cell that keeps a memory state, and non-linear gates regulating information flowing in and out the cell. It is possible to design the cells such that the network is able to both learn and forget and therefore work continuously [Gers] on sequences that are not divided into clearly bounded subseries.

5.3. Status and Future Work

Needed changes to adapt Robinhood Policy Engine to SAGE have been identified, and several of them have been implemented: support of custom attributes, triggering MERO actions, enabling interactions with other external components. These changes have been integrated to the upstream version of Robinhood in the version 3.0.released in September 2016.

Other adaptions all still required to run Robinhood on MERO, and must be implemented in the next steps, in particular:

- Implementing a FDMI client to collect access information from MERO, and adapt the event processing.
- Removing dependencies to POSIX, and make Robinhood run on Clovis API.
- Replacing Robinhood database backend by MERO key-value store.

6. Conclusion and Next Steps

Great progress was made in specifying and implementing the main components of HSM for SAGE:

- A library has been implemented to drive the system in selecting an appropriate storage level, based on application hints about file lifetime and access patterns. This library can now be integrated to the SAGE platform. As part of the integration process, pool definitions and other library parameters will need to be finely tuned to achieve optimal pool selection with this library.
- The machinery of Composite Layouts has been fully specified and will be the cornerstone of HSM copy mechanism. Both are now entering the implementation phase.
- The HSM policy engine component will rely on the open source Robinhood Policy Engine that is to be adapted to MERO object store. Some of these changes have been done, and this adaptation work is to be continued and completed.

The last steps of the project will consist in putting all these pieces together: applications, HSM libraries and tools, policy engine, and validate their working on the SAGE hardware platform.

These next steps of development, integration, and validation will be finally described in D3.9 (HSM for SAGE: Final Validation Report).

7. References

[RHPE] *Robinhood Policy Engine*, Project Website: http://robinhood.sf.net
[Karp15] *The Unreasonable Effectiveness of Recurrent Neural Networks*, Andrej Karpathy, May 2015, http://karpathy.github.io/2015/05/21/rnn-effectiveness/
[ScikitL] *Machine Learning in Python*, Scikit-learn, http://scikit-learn.org
[Caffe] *Caffe Deep Learning Framework*, http://caffe.berkeleyvision.org
[Torch] *Torch, A scientific computing framework for luajit*, http://torch.ch
[LSTM97] HOCHREITER, Sepp et SCHMIDHUBER, Jürgen. Long short-term memory. Neural computation, 1997, vol. 9, no 8, p. 1735-1780.
[Gers] GERS, Felix A., SCHMIDHUBER, Jürgen, et CUMMINS, Fred. Learning to forget: Continual prediction with LSTM. Neural computation, 2000, vol. 12, no 10, p. 2451-2471.