Bachelor Thesis

Distributed storage management policies in JULEA

Julian Benda
julian.benda@st.ovgu.de

March 27, 2022

First Reviewer:
Jun.-Prof. Dr. Michael Kuhn

Second Reviewer:
Kira Duwe

Supervisor:
Jun.-Prof Dr. Michael Kuhn and Kira Duwe

Abstract

A growing variety of storage media and technologies focus on hierarchical storage management and lead to more complex storage layouts. This work aims to develop a mechanism to implement HSM policies in JULEA to provide a streamlined framework that supports this type of research. JULEA is a storage framework for distributed systems, e.g. high-performance computing (HPC) systems. Implementing a modular mechanism to write high-level HSM policies enables extended object management on storage hierarchies, creating an effective environment for research.

For constructing the HSM policy interface, different promising ideas for HSM are analyzed. Based on this interface, the integration in JULEA is designed and implemented. To evaluate the performance various use cases and benchmarks were run with different simple policies, like LRU, and compared against JULEA without hierachy storage management (HSM).

The qualitative analysis shows that the HSM mechanism reduces the overall performance for single tier scenarios up to 20%, due to additional metadata handling and function calls. However, comparing different policies shows that the performance of the lower storage tier can be completely hidden. In conclusion, the implemented mechanism works as intended but should be revisited to minimize the performance impact. In addition, when using HSM, the application characteristics should be analyzed and used as the basis for selecting and parameterizing the policy.

Contents

Chapter 1.

Introduction

A growing variety of storage media and technologies focuses on hierarchical storage management and leads to more complex storage layouts. To support further research in this area, this work wants to provide a streamlined research framework by developing a mechanism to implement HSM policies in JULEA, a research framework for data access on HPC systems. By implementing a modular mechanism for writing high-level HSM policies, JULEA's applicability will be expanded to manage objects on storages hierarchies, creating an effective environment for research.

In this chapter a brief introduction is given. First, the motivation and goal of the thesis will be discussed. Afterward, the general structure will be explained.

1.1. Motivation

Modern HPC (high performance computing) systems contain a vast amount of different storage and memory modules[1](Oukid & Lersch, 2018). They all are mainly different in design but are typically compared by their latency, bandwidth, and capacity. In contrast, the determining factor for limited capacity is mostly the cost[2]. Also, latency and bandwidth typically have a weak correlation, which allows ordering the different technologies in a hierarchy as seen in figure 1.1. Noticeable is the difference in access time between a build-in and external cache, and between memory and storage, which is 5ns to 100ns and 100ns to 100µs, respectively. This gap between storage and memory is, compared with pre-SSD times, relatively small. New technologies like Intel's Knight Landing an NVRAM[3], which introduce even more layers into that hierarchy, closes this gap further.

The prices of different storage technologies are different in multiple orders of magnitude and their energy consumption (Borba et al., 2020; Jiang et al., 2021; Katal et al., 2022). This is not only important for monetary value but also with rising interest and focus on green computing (Hariri et al., 2019; Krish et al., 2016).

Different researches and experiences have shown that better storage does not always result in significantly faster execution(Guerra et al., 2012; Krish et al., 2016; Meng et al., 2014). For workflows requiring large sequential reads, a PFS (parallel file system) with high bandwidth

[1]In this thesis, the term memory is used for volatile medium (like DDR or SRAM) and storage for persistent (like HDD or tape).

[2]With cost typical, the cost per byte is referenced

[3]Non-Volatile random access memory = persistent memory (see fig. 1.1)

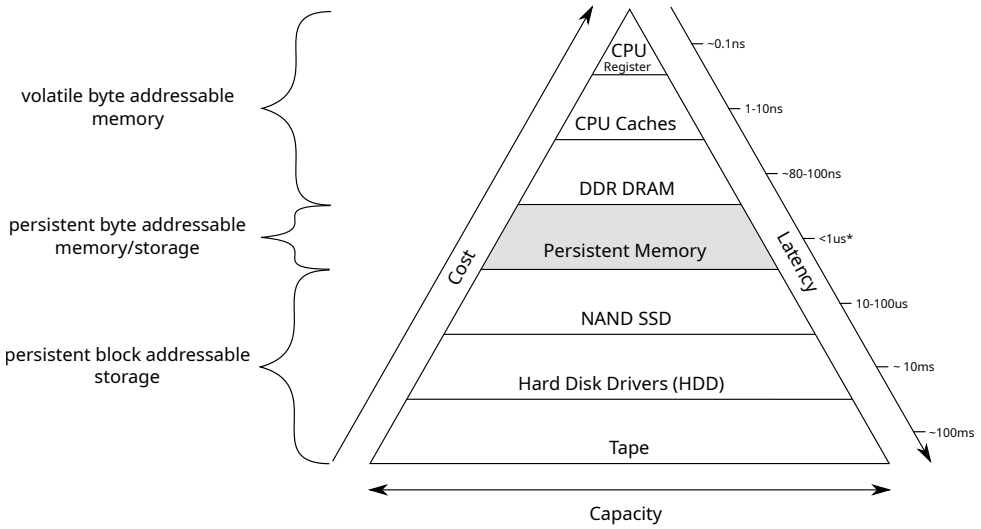


Figure 1.1.: Latency pyramid based on “Developers Embrace Intel”(Rudoff, 2019)

disks may be all that is needed to provide the required data throughput. A typical example for tiered storage is the addition of LFS (linear file system) to traditional HDD disks. In this scenario archived data are stored on durable, cheap, but slow tapes.

CPU caches are the fastest layer in the storage hierarchy. These layers are needed to provide the CPU with instructions on a cycle base, amongst others. However, the built-in CPU caches are SRAM with a high transistor count and high production cost (Oukid & Lersch, 2018), which leads to only small caches being implemented. Because accessing these cache levels takes only a few clock cycles (“How L1 and L2 CPU Caches Work”, 2022), the CPU contains a hardware-implemented caching algorithm to manage the access (Wong, 2013). Furthermore, it is therefore not accessible for software-controlled HSM. On the other hand, LFS has access cycles that can last days because of the mechanical parts, e.g., retrieving the cartridges and loading them into the drive.

To leverage the hardware, data distribution on the different tiers is vital. It would be most sensible for a system with an SSD and high bandwidth HDD to put small data to the SSD. Since access to small data is often random access, they will leverage the small latency of the SSD. In contrast, large data access is typical bandwidth capped. Therefore, the difference between HDD and SSD is minimal (Dong et al., 2016; Zhang et al., 2015). In addition, having large objects on the cheaper medium is also economical.

The optimal usage of this hierarchy is an open research question (see sec. 3). However, different from cache layer management it is possible to implement more sophisticated software, because of the long access times. These software systems are called HSM.

JULEA is a storage framework for researching and teaching. It provides a modular structure to support rapid prototyping and testing new approaches for parallel and distributed file systems (Kuhn, 2017). Therefore JULEA uses a client-server approach and supports multiple storage servers with different backend types like key-value stores(KV-stores) for metadata storage or object-stores for large data segments (see fig. 1.2).

Currently, a single JULEA server can only leverage a storage hierarchy by placing different stores on different levels/tiers (see fig. 2.2). To implement HSM approaches with JULEA, it would be necessary to instantiate multiple server with an object store each and place much

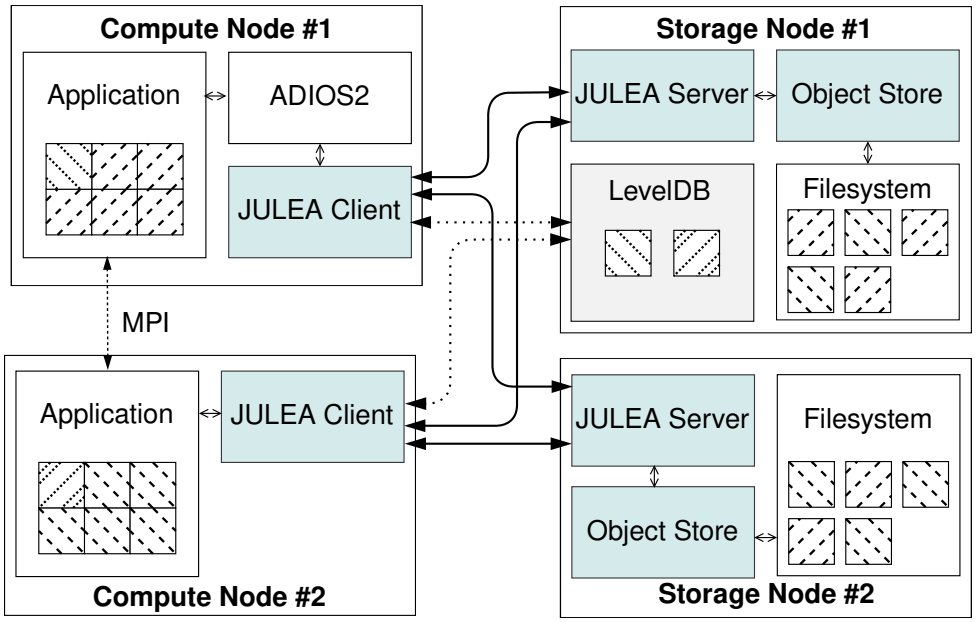


Figure 1.2.: Example setup constructed from JULEA's main components with two clients where one accesses JULEA through ADIOS2 (#1) and the other direct (#2). They communicate with two servers containing an object store and one(#1) a KV-store. Simplification of (Duwe & Kuhn, 2021, fig. 4).

logic on top for accessing the data. This overhead contradicts the demand for a modular and easy-to-use framework for prototyping, research, and teaching.

1.2. Goal

The goal is to introduce a mechanism in JULEA to integrate different HSM policies. The policies should be modular as is JULEA and allow the implementation of a wide variety of policies without reducing the overall performance. A HSM policy is a set of rules that defines when objects are moved between different storage media/tiers. An example policy typically used for CPU caches is the least recently used (LRU) policy. As the name implies, the concept is to demote the least used objects to make room for promotion. Its simplest implementation will promote accessed objects and demote the longest unused objects in exchange.

In this context, the HSM policies will only be implemented for the object backend because the metadata/KV-storage should be used for small information snippets. Small data pieces will lead to many small, mostly random access, and therefore a high tier of storage, like an SSD, is in every case preferable (Duwe & Kuhn, 2021; Iliadis et al., 2015; Zhang et al., 2015).

1.3. Structure

The next chapter (sec. 2) explains central concepts important for the reasoning in the later sections. At first, two topics are about modern storage (sec. 2.1) and then an insight how JULEA manages objects and object backends (sec. 2.2). The storage topics summarize currently available tiers and a digression of why a larger SSD is not the solution for every problem.

Next is an overview of current research about HSM on HPC (sec. 3), how they approach this problem, and the trials they propose. This insight will be used to give directions for possible policies used by JULEA and also to design the policy mechanism in a way so that this approach can be implemented.

Chapter 4 is about the design and implementation of this mechanism. First, the design will be explained started by what the fundamental goals are to archive based on the preliminary findings (sec. 4.1). Based on these goals, the encapsulation of the policy will be described, followed by a listing of different components and how they interact to provide certain functionality.

After that (sec. 4.2), the implementation will be presented. First is the policy infrastructure with the policy's actors and sensors. Last but not least, a peek at how the policy can process the data without slowing down JULEA is taken.

The implementation will then be evaluated (sec. 5). At first, the used hardware and environment will be documented (sec. 5.2). The benchmarks will test different policies (sec. 5.3) implemented with the new mechanism in different scenarios. It will also be measured how large the performance gap between a proxy policy and JULEA without the policy mechanism is. In the last chapter (sec. 6), the insights will be summarized, and directions for further research and work will be proposed.

Chapter 2.

Background

This chapter introduces fundamental concepts important for decisions in this thesis. The first part is about storage properties. Therefore, it first lists currently available storage/memory types and their usage. Then the problem of access contention is elaborated, showing why simple caching is not a solution in every case. Lastly, an in-depth explanation of JULEAs object management follows, which serves as a basis for design decisions later on.

2.1. Modern Memory Concerns

The relationship with storage has shifted in the last decades. In 1995 one concern was that a computer might not support threading and therefore should sequential network communication (Schmidt, 1995). Nowadays, even consumer CPUs have 16 cores. As mentioned by Perarnau et al., dedicated migration threads can boost performance because all threads cannot possibly be delivered with enough bandwidth (Perarnau et al., 2016). However, this also means that more resources are available, reducing the bottleneck through memory management.

Also, new storage and memory technologies have increased, as shown in figure 2.1, which closed the large performance gap between RAM and disk storage. This opens new possibilities and asks for a more detailed evaluation of different module types for different use cases.

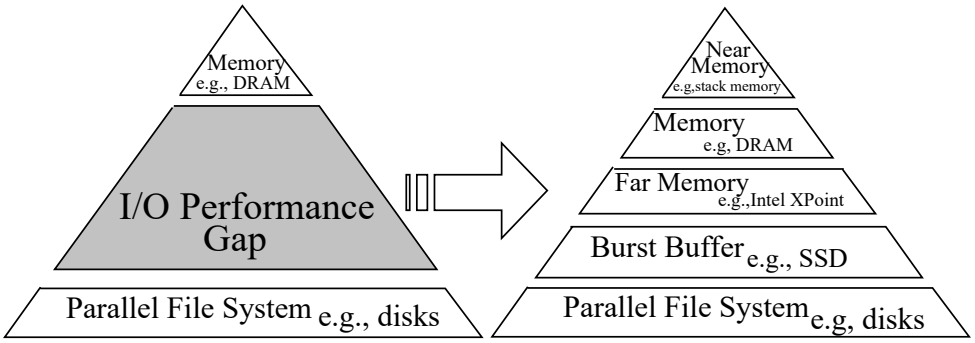


Figure 2.1.: Traditional vs. modern storage hierarchy (Dong et al., 2016, fig. 1). Higher tiers have less capacity but lower latency and higher bandwidth.

2.1.1. Deep Storage Hierarchy

The current systems provide a variety of different memory and storage tiers. The description tier is used because typical storage media have either higher latency or lower bandwidth. Nevertheless, lower tiers become attractive because of cost per bit, energy consumption, storage density, or persistence, as demonstrated in figure 1.1. (Oukid & Lersch, 2018)

Typical, the fastest tier is the in-processor SRAM (static random access memory). This memory is volatile and is commonly known as L1-3[1] cache. Because of its fast access rate [2], this type of cache is directly built in the CPU and managed with hard-wired algorithms (Wong, 2013) and therefore not of interest for most HSM works.

The next tier is the DRAM (dynamic access memory). In contrast to SRAM, it is simpler in construction but needs to be refreshed frequently. The rapid refresh cycle results in higher energy consumption and a limit for storage density. DRAM exists in different specifications with different parameters, Intel's Knight Landing series experimented with a "near"- and "far"-memory. Traditional DRAM, "far-memory," is therefore paired with multi channel DRAM (MCDRAM), "near-memory". As shown by Peranau et al., it improves the total processing time in combination with data migration cores (Perarnau et al., 2016).

NVRAM is the newest tier. It is slightly slower than DRAM but is non-volatile and can achieve much higher storage densities (Kang et al., 2009; Oukid & Lersch, 2018). Especially database applications can profit from this since querying requires many broad spared memory accesses (Lehner, 2017).

The subsequent tiers are long-term storage. The SSD is the fastest among them. Because of the construction, it allows for fast random access compared to an HDD. However, because of higher energy consumption and the lower price per byte of HDDs, the SSD is still not the primary storage medium for HPC (Oukid & Lersch, 2018). Especially for large sequential reads, a good striped PFS with HDDs can archive similar performance to SSDs (Koo et al., 2017).

Last but not least, there is tape storage or LFS. Tape is primarily self-contained storage, because of the mechanic procedure to change the tape in the reader and because of its linear nature has a high latency. Therefore, it is mainly used to archive data, as shown in Koltisdas et al. (Koltsidas et al., 2015). Tape is a very durable and high-density storage medium, but because of its high latency not suitable for storing data in a workflow.

As this overview shows, there are currently many different tiers which different properties and use cases. It also shows that dedicated HSM can leverage multiple storage tiers more than the standard caching approach and that processing power is not the limiting factor for data-intensive workflows.

2.1.2. Access Contention

An intuitive assumption is that if all frequently accessed data fit in a higher tier, it is a good idea to do so. Nevertheless, as shown by Zhang et al. and Perarnau et al., the differences in

[1] L1, L2, L3, and sometimes L4 cache.

[2] On CPU caches have typical access times of a view cycles to a low three-digit number of cycles fig. 1.1, ("How L1 and L2 CPU Caches Work", 2022).

performance between the different tiers are not necessarily high enough to hold this assumption (Perarnau et al., 2016; Zhang et al., 2015). Too much parallel access to the same SSD, for example, can lead to an access contention which means that the controller cannot handle the request simultaneously and therefore introduces a latency for queued requests (see sec. 5.2). Also, a more extensive sequential operation may block multiple channels of the SSD, which can also lead to more latency for other requests or reduced bandwidth.

Therefore, an HSM policy simply putting everything as efficiently as possible at the highest tier is not the best. Iliadis et al. is presenting an approach to factor workload of different tiers in when dividing data around them (Iliadis et al., 2015). It presents an offline method to calculate the distribution because it is NP-hard.

Another way to reduce access contention is, as shown by Zhang et al., to use processing power for complex HSM since the workflow can not use the full processing capabilities because of the limited data throughput (Zhang et al., 2015). In these scenarios, fewer calculation resources can improve the throughput or overall runtime.

2.2. Object Management in JULEA

JULEA is a storage framework that focuses on flexibility to allow research on different parts of storage management without writing a whole system or producing complex code that is not reusable. Therefore it is composed of modules, which provide each a unified interface used to, for example, implement a custom object backend.

An object backend is a library for storing larger data objects. JULEA distinguishes, like many HPC storage systems, between metadata and object data. Furthermore, JULEA allows writing small snippets of data directly to the metadata or dedicated DB nodes. For KV and DB backends, the memory access is vastly different from object backends. Because of the random and small memory accesses, KV-stores and DB can leverage an SSD in nearly every situation.

Each JULEA server currently supports up to one object backend, as seen in figure 2.2. Therefore, to experiment with HSM, a new object backend must be implemented with integrated HSM. An alternative solution would be to have the client communicate with multiple servers, one per storage tier, and manage the data. This would be a client-controlled HSM and is generally not desirable (Devarajan et al., 2020). Implementing a new backend is coupled with many boilerplates and is not transferable for a different backend setups. Therefore JULEA wants to use multiple object stores in one server as pictured in figure 2.2, which will need a HSM module on place the objects to different tiers.

The general structure of the backend interface, seen in the listing 2.1, is a `scope` that will `open/close`. This entry check ensures that when another operation is executed, the necessary rights are provided, and potential serialization steps will happen on a block level, which reduces the chance of an error occurring.

Object backends are managed with modules. Modules are a mechanism to load libraries at runtime. This, in combination with a naming convention, allows modules to be compiled separately and loaded at program start. Also, the module can easily be changed in the configuration file[3] without recompiling.

[3] https://github.com/julea-io/julea/blob/master/doc/configuration.md

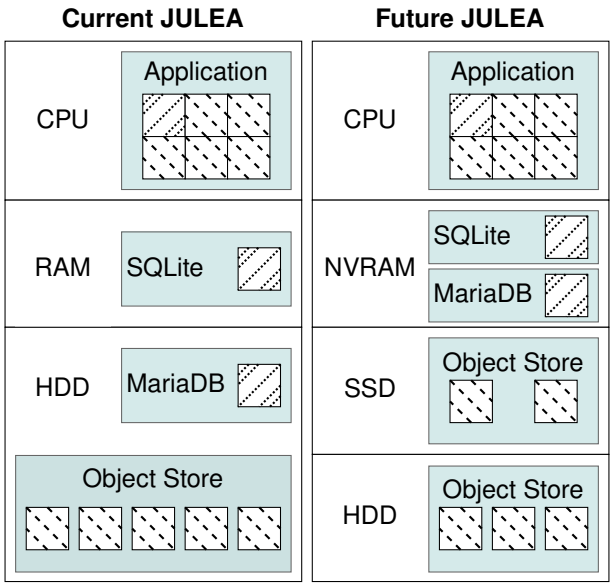


Figure 2.2.: "Different storage components on corresponding hardware layers: [...] JULEA's design makes it possible to easily separate data and metadata and handle it differently. In the current setup, the SQLite database is held in main memory while MariaDB and the object store reside on HDDs [(left)]. The proposed extension will extend JULEA's support for storage hierarchies to be able to make full use of new technologies such as NVRAM. This will also allow us to put parts of the object store onto SSDs for fast access, while the majority can be kept on HDDs (right)." (Duwe & Kuhn, 2021, fig. 6)

Besides `init` and `fini` methods, all functions are of interest for a policy because they inform about interaction with objects and storage. As shown by Meng et al., the combination of different hint types increases the hit ratio for higher tiers (Meng et al., 2014). The policy's access propagation should be as detailed as possible to allow for a wide range of hinting functions.

The scope spanned with `open` and `close` allows for an excellent point to introduce a barrier to coordinate data access from clients and access from the policy for migration purposes. This barrier reduces the chance of implementing race conditions.

Listing 2.1 JULEA module definition for an object backend (`jbackend.h`).

```
136 struct {
137     // instantiate new object backend
138     gboolean (*backend_init)(gchar const*, gpointer*);
139     // deconstruct object backend
140     void (*backend_fini)(struct JBackend*);
141 
142     // create a new object in this backend and open it
143     gboolean (*backend_create)(struct JBackend*, gchar const*, gchar const*,
        ↪ gpointer*);
144     // open a object, may also create it.
145     gboolean (*backend_open)(struct JBackend*, gchar const*, gchar const*,
        ↪ gpointer*);
146 
147     // delete object and associated data, object must be open
148     gboolean (*backend_delete)(struct JBackend*, gpointer);
149     // closes a object
150     gboolean (*backend_close)(struct JBackend*, gpointer);
151 
152     // Gain properties of object, currently total size and last accessed time,
153     // data may be outdated if executed "write" without sync" or "close".
154     gboolean (*backend_status)(struct JBackend*, gpointer, gint64*, guint64*);
155     // Forces data transfer to storage and metadata updater
156     // without closing object.
157     gboolean (*backend_sync)(struct JBackend*, gpointer);
158     // Reads part (offset + size) of object from memory to backend,
159     gboolean (*backend_read)(struct JBackend*, gpointer, gpointer, guint64,
        ↪ guint64, guint64*);
160     // Writes part (offset + size) of object from memory to backend,
161     // extends object if needed.
162     gboolean (*backend_write)(struct JBackend*, gpointer, gconstpointer, guint64,
        ↪ guint64, guint64*);
163 
164     //  Returns a iterator to iterate though every entry at this backend.
165     gboolean (*backend_get_all)(struct JBackend*, gchar const*, gpointer*);
166     // Returns a iterator to iterate though every entry in this backend
167     // with a specified prefix.
168     gboolean (*backend_get_by_prefix)(struct JBackend*, gchar const*, gchar
        ↪ const*, gpointer*);
169     // Increments a entry iterator and return the next entry name.
170     gboolean (*backend_iterate)(struct JBackend*, gpointer, gchar const**);
171 }
```

Chapter 3.

Related Work

This section will look at different approaches using deep storage hierarchies effectively with an offline and online algorithm. This insight will be used to design the HSM policy mechanism for JULEA in a way that would allow implementing this approach as a policy module and use it within JULEA.

3.1. H-Fetch (Devarajan et al., 2020)

With caches and RAM, tiered storage and caching have existed for a long time in computer science. What data should be held in faster memory is also thought about a lot. A common conclusion is that loading the data before it is needed[1] will improve the performance (Devarajan et al., 2020; Ghoshal & Ramakrishnan, 2021; Perarnau et al., 2016). This is why stride detection is already implemented on a CPU level. Nevertheless, with a growing depth of storage hierarchies, the question is not only what to cache but also in which layer. As shown by Devarajan et al., the typical solution does not answer the question of where to prefetch (Devarajan et al., 2020).

Another problematic point is that most solutions use a *client-pull* approach. This means that prefetching is application-oriented and will not coordinate global access. The missing coordination may lead to redundant data or unwanted eviction. A *server-push* approach would analyze global access patterns and provide the data accordingly.

Furthermore, this work emphasizes the importance that the HSM is:

Hierarchy aware The manager should keep a global view and access to the storage tiers. This allows reduced data movement on intermediate tiers.

Application agnostic The manager should be decoupled from running applications and therefore be allowed to fetch data blocks that are not currently accessed.

Their H-Fetch HSM keeps track of file accesses similar to the access types used by JULEAs object backend. These hints are used to create a heat map for the data segments. Further, they found that the frequency of the data placement influences access performance. Therefore, high placement frequency will lead to short read times, but higher latencies cause the data migration to block access to data, as seen in figure 3.1.

[1] This mechanism is called prefetching.

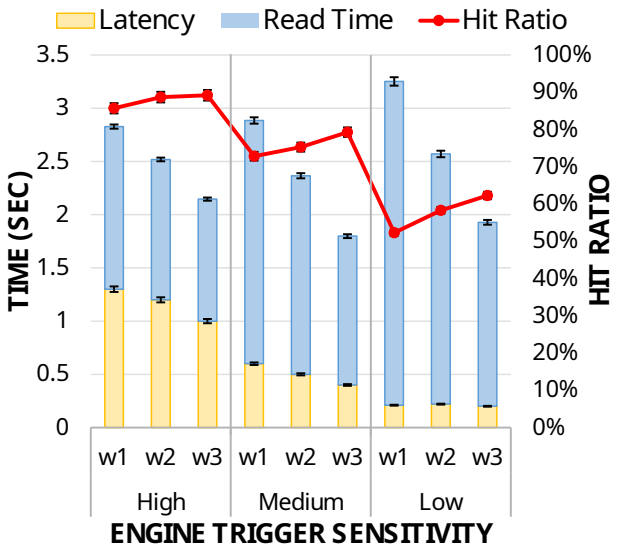


Figure 3.1.: Benchmark for high/medium/low frequent placement of data by HSM for data intensive (w1), balanced (w2) and computation intensive (w3) workflows, as adapted from (Devarajan et al., 2020, fig. 3(b))

3.2. Data Elevator (Dong et al., 2016)

Besides complex optimized solutions, if the focus on writing data, even simple policies can increase the performance significantly in deep storage hierarchies. The *Data Elevator* presented by Dong et al. is an example of a simple but effective tool (Dong et al., 2016).

The *Data Elevator* is a mechanism for more efficient usage of burst buffer (BB). They show that if the PFS nodes directly read data from the BB *contention* it will lead to poor performance of the BB. With the *Data Elevator*, a dedicated process will read memory chunks of the BB to its his local memory, allowing for better throughput utilization, and then place them on the slow tier. The intermediate transfer to the memory blocks the BB less than transferring the data directly to the slow storage and leads to *end-to-end-time* boost of up to 35% (Dong et al., 2016, ch. 7). Data will then be stored from the process to the PFS without slowly fighting for priority on the BB. It should be mentioned that they use BB accessed via the network, which allows other nodes to do this migration task.

This works shows that if the specification is well known for a workflow, simple HSM can increase the performance for real-world examples up to 6 times (Dong et al., 2016, ch. 5). It also emphasizes using dedicated *data-mover* to increase performance, especially for data-intensive workflows.

3.3. Adaptive Storage (Koo et al., 2017)

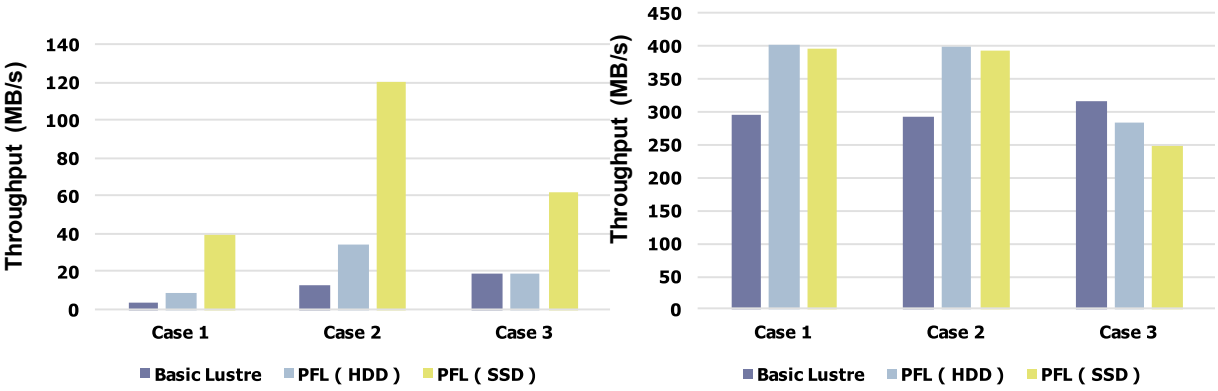
Splitting data between multiple disks is a familiar concept and implemented from RAIDS to modern PFS. Striping allows for faster reading since if two segments are on different disks, they can be read with the combined bandwidth of these disks. Even a sequential read can profit from this because multiple segments can be read in parallel because the network throughput from the computation nodes is much higher then from disk storage.

Koo et al. has compared the fixed striped PFS Lustre with a progressive file layout (PFL) build into it (Koo et al., 2017). The idea is that the first small part of each object would be stored on a different disk, and the remaining data would be striped over the remaining disks. The consequence will be that small files are completely stored on a disk, while larger files are distributed across multiple storage devices to leverage parallel access.

This approach already resulted in higher total throughput because small file accesses will not be compete with efficient large reads for large I/O-operations (see fig. 3.2). If an SSD is used to store the small I/O-fragments, the throughput for these fragments is significantly higher since the random access performance of SSDs is better than that of HDDs (see fig. 3.2a).

Another important observation is that selecting the correct size for *small* files has a massive impact on performance. Setting the size too large will drop the performance for small I/O-operations and increase the contention, which consequently also reduces the large I/O-operation throughput too (see fig. 3.2b).

This leads to the observation that SSD is best leveraged with random access and that mixed I/O-scenarios can significantly improve system throughput. Furthermore, this shows that object sizes are an imported factor for efficient HSM and should be considered.



(a) small file I/O-throughput

(b) large file I/O-throughput

Figure 3.2.: Compare PFS with fixed striping across 4 HDDs (Lustre) / with PFL storing 4MB on dedicated HDD and striping remaining data across 3 HDDs (PFL(HDD)) / with PFL similar properties then PFL(HDD) expect using an SSD for the first 4MB for mixed I/O workloads with large (16GB) files and small files with 2MB/4MB/8MB in case 1/2/3 respective (adapted from (Koo et al., 2017, fig. 8/9)).

3.4. Manage Workflows with HSM (Ghoshal & Ramakrishnan, 2021)

Another approach to address HSM to focus on workflows. It is common to run multiple applications instead of just one, which are interrelated and where each contributes different data. This is also the case when "manual tiering" is executed. Therefore data needed for the workload are stored in advance on higher tiers, and data with less access are explicitly stored on lower tiers.

This process is not only tedious, but it is also error-prone because of changing architecture and high effort to adapt. While there are already many tools that analyze applications and their dependencies and run them accordingly, there are only a few which also consider data migration.

Ghoshal & Ramakrishanan proposed a workflow management that, besides orchestrating applications, moves data asynchronously to maximize the available tier utilization (Ghoshal & Ramakrishnan, 2021). Their approach extends existing workflows with additional *data-mover* tasks (see fig. 3.3). These additional tasks allow, for example, moving data to the archive while it is still being written to use high throughput to minimize waiting time for the application and then free fast memory while the application is calculating. Because the workflow is calculated offline, it can simulate different approaches and select the best.

They concluded that a data-centric managed workflow performs the same tasks with less overall memory consumption because data can be stored earlier and deleted accordingly. Also, data can be persistent while the application is still running without human interaction and overhead.

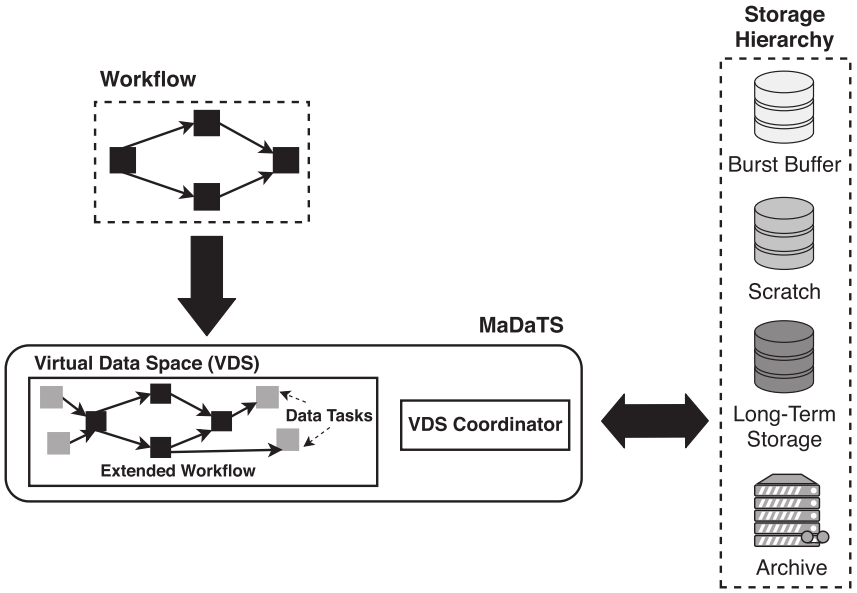


Figure 3.3.: "Users map workflows and data to a virtual data space (VDS) and a VDS coordinator manages the data during workflow execution on tiered storage systems." (Ghoshal & Ramakrishnan, 2021, fig. 1)

Chapter 4.

Design and Implementation

This chapter is about the design and implementation of extending JULEA to run HSM policies on an object backend server. The first section outlines the structure and reasoning behind the design. The second section discusses actual implementation with C in the current JULEA source code.

4.1. Design

An abstraction of a policy is needed to implement and test different HSM-policies in JULEA and keep the modularity. Since JULEA is already an established codebase, the design concepts will be inherited, avoiding confusion. Furthermore, it allows more straightforward modifications later.

JULEA is designed object-oriented. These objects are commonly accessible via functions like in GLib ("GLib-2.0", 2022). JULEA uses `GModules` to avoid recompiling if modules, like the object backend, are changed, swapped, or reconfigured. Modules are a mechanism from GLib to load and use libraries at runtime. That way, only the module/library needs to be recompiled when changes are made.

The following section covers the design goals to be achieve with this implementation and outlines how the policy module needs to be constructed to fulfill these requirements. The last section will discuss the integration in the current JULEA object management ecosystem (see sec. 4.1.3).

4.1.1. Design Goals

The following goals were set to create an extension that will be used in the future.

Minimally invasive The modifications to the current code should be minimal to avoid introducing errors. Further, it should not complicate or interfere with future planned extensions.

Convenience The usage of this extension should be as convenient as possible to allow for faster and easier implementation of policies. A more convenient usage will also reduce the turnover time for new ideas.

Flexible As shown in chapter 3, there are many different approaches for implementing HSM policies depending on the environment and use case. The policy-interface must offer a high degree of freedom of its operations to support a wide range of policies.

Minimal Overhead The mechanism should introduce no relevant latency for data access if not used or if the policy ignores the corresponding action. A small overhead is essential because the latencies on newer tiers will become smaller, and the performance should not be dumped from this extension.

Parameterized Policies A simple mechanism should exist to provide parameters for the policy. Parametrization allows easier testing and adaptation without modifying source code or recompiling.

Configurable via JULEA 's Configuration As extension to *minimal invasive*, the policy's selection and configuration/parametrization should be made with the existing configuration file. That allows for a more straightforward configuration since all information is contained in one file.

Reuse as much as possible Existing JULEA functionality should be used to avoid duplicated systems which are a common error source when the system evolves (Carzaniga et al., 2015). It also reduces the amount of code needed and therefore increases the maintainability.

Transparency Different policy strategies need different data, as shown in chapter 3. Most of that data is related to memory/storage access. The policy should receive all data related to storage access to leverage the available data as much as possible.

Dedicated Computation Resources Data throughput is a massive bottleneck for scientific workflows (Chen et al., 2016). Therefore dedicated *data-mover* processes can reduce the total computation time. Chapter 3 showed that a variety of HSM approaches require dedicated computation resources to be performed efficiently.

Outside Communication Channel An information flow from other applications to the policy should be present. This communication channel can, for example, be used to implement application and data workflows manager (Ghoshal & Ramakrishnan, 2021) or allow the user to specify their data placement directly (e.g., by namespace). It would be possible to integrate them directly into the policy. However, against the background that JULEA already has a communication protocol, it would introduce duplicated systems that would work against the goals *reuse as much as possible* and *convenience*. Therefore the existing messaging system of JULEA should be expanded to allow sending messages to the policy.

4.1.2. Policy Module

In JULEA, modules are typically used in the following way: The module provides a `_info`[1] function. This function instantiate an object which implements the appropriate interface. This interface contains in general an initialization/constructor (`_init`) and finalization/destructor (`_fini`) function to manage the object (see lst. 2.1). This general structure will be adapted for the HSM policy module interface.

[1] E.g. `backend_info` for backend modules

Initialization For the initialization of the policy, a reference to `JBackendStack` gets passed, which is later used to issue migrations. In addition, the policy can access the object storage to write and store data, read other objects or fetch the size of objects. That will allow a *flexible* and *convenient* way to get additional information about objects and modify them.

Hints Most decisions of HSM are based on hints. These hints are created from data access operations. The policy should decide what kind of hints it wants to collect to provide *flexibility*. For deriving hints the policy get signals via `process_access` (see listing 4.1 line 90). Because the function allows ignoring the kind of access, they are *convenient* to use. Moreover, with the data segment containing data depending on the access type[2], it offers *transparency*. To *reduce the overhead* for data accesses, the policy should keep the hinting function as slim as possible. The object creation process is a special case handled with `process_creation` instead of `process_access`. This is necessary because to provide more *flexibility* the policy must map a new object directly to a storage tier. In a simple implementation, this may be a function that returns a fixed value or a more demanding task for scenarios with a low object creation frequency.

Processing Processing should be offloaded to a *dedicated process*. The function `process` passes the resources to the policy (see listing 4.1 line 99). This function will then be called from a dedicated thread to ensure processing time for the policy. This process then can migrate objects between different tiers in an asynchronous manner.

Parametrization An argument list passed from the config file to the policy offers a simple and *convenient* way to *parametrize* the policy, which results in easily modifiable and highly *flexible* usage. If the policy uses instructions to move data, they arguments may contain a path to the movement instructions. Since the policy is able to read from the object storage, the instructions and configuration could also be read from this.

Communication Nevertheless, creating objects just for a one-time read is much overhead and may pollute the object storage namespace. To avoid this, `process_message` provides a way to *communicate* with the policy (see listing 4.1 line 111). A message is composed of a string decoding the message type and a data segment of fixed length. The type string may be omitted but will, if used, provide information to create better readable logs.

[2]For `process_read` this are the offset and size of the read

Listing 4.1 Interface for policy module (`jbackendstack.h`)

```
85  typedef struct
86  {
87          gboolean (*init)(gpointer* this, const JList* args, JBackendStack*
            ↪  backendStack);
88          gboolean (*fini)(gpointer this);
89          /// short processing period to handle data
90          gboolean (*process_access)(gpointer this,
91                                     const gchar* namespace,
92                                     const gchar* path,
93                                     guint obj_id, ///< [in] object index, used for
    ↪  easier recognise same object
94                                     guint tier,         ///< [in] tier where object
    ↪  is currently on
95                                     JObjectBackendAccessTypes access,
96                                     gconstpointer data);
97          /// long processing period for migration and large calculation
98          /** \retval FALSE on error */
99          gboolean (*process)(gpointer this);
100         /// short processing period to match new object to a storage tier
101         /** \retval FALSE on error
102          * \sa j_backend_stack_get_tiers */
103         gboolean (*process_create)(gpointer this,
104                                    const gchar* namespace,
105                                    const gchar* path,
106                                    guint obj_id,      ///< [in] continious object
    ↪  index
107                                    guint* storageTier ///< [out] storage tier ID
    ↪  to store new object
108         );
109         /// process policy custom message
110         /** \retval FALSE on error */
111         gboolean (*process_message)(gpointer this,
112                                     const gchar* type, ///< [in] message type
                                        ↪  encoded as 0-terminated string
113                                     gpointer data,     ///< [in,out] optional
                                        ↪  message data
114                                     guint length       ///< [in] size of data
    ↪  field
115         );
116
117         gpointer data;
118 } JObjectBackendPolicy;
```

4.1.3. Functionality

Currently, the JULEA server directly accesses the object backend. JULEA is already able to instantiate more than one object backend. That allows instantiating an object backend for each tier, which *reuses* existing functionality and reduces error sources.

Managing multiple object backends needs a new structure. Instead of directly accessing the one object backend, the required backend will now be requested from the `JBackendStack`. This *invades* only minimal in existing backend code and provides, in addition, a *convenient* access.

Access control The core functionality of a HSM is the migration of data between tiers. While the same object can exist on two tiers simultaneously, it would be complicated to generalize the complex access management. For that reason, HSM are strategies that do focus on reading, writing or explicitly renounce duplicated data (see sec. 3). To keep the first implementation simpler, it will also refuse duplicated objects.

It is necessary to block object access for that process to prevent race conditions or data corruption during migration. That could be archived by using a mutex for each object, which ensures only one operation at a time. The `open/close` structure of JULEAs object management is suitable to insert this mutex.

It is essential to notice that an object backend may support parallel accesses, for example, parallel writing nonoverlapping parts. Suppose a mutex around each access would not leverage these capabilities because it would block concurrent access. A read-write-mutex can be used to avoid this issue. This kind of mutex can differentiate between two accesses types. Read access can be parallel, whereas write access requires exclusive access. In this case, the read case would be the object access via normal access operations, and the write case the migration from one tier to another.

To track this mutex, `JBackendStack` has similar to `JObjectBackend`, a pair of `begin/end` functions for requesting the corresponding backend. Since the access in JULEAs object backend is object-based, the read-write-mutex locking is simply integrated into the scope (see fig. 4.2). Therefore, the `begin` function will provide the backend/tier that contains the requested object and acquire a *read-lock* on them. This ensures no migration will happen while still allowing for concurrent access.

Metadata For an efficient mapping between object and tier, a KV-store would be ideal. While JULEA already provides a meta-data server with a KV-store, this server might be at a different node and only accessible via the network. That could introduce high latencies, which contradict the goal to *minimize the overhead* on a frequent task like querying an objects tier. Also, this would *invade*/pollute the meta-data namespace, which leads to implicit requirements.

On the other hand, using a hash-table, would introduce a new dependency, reduce the *flexibility* and might lead to race conditions. A compromise between the above two options is to instantiates a local KV-store that is only used from the `JBackendStack` as pictured in figure 4.1. This solution minimizes latencies, avoids namespace pollution, stores the data, and enables quick changes to the KV-store.

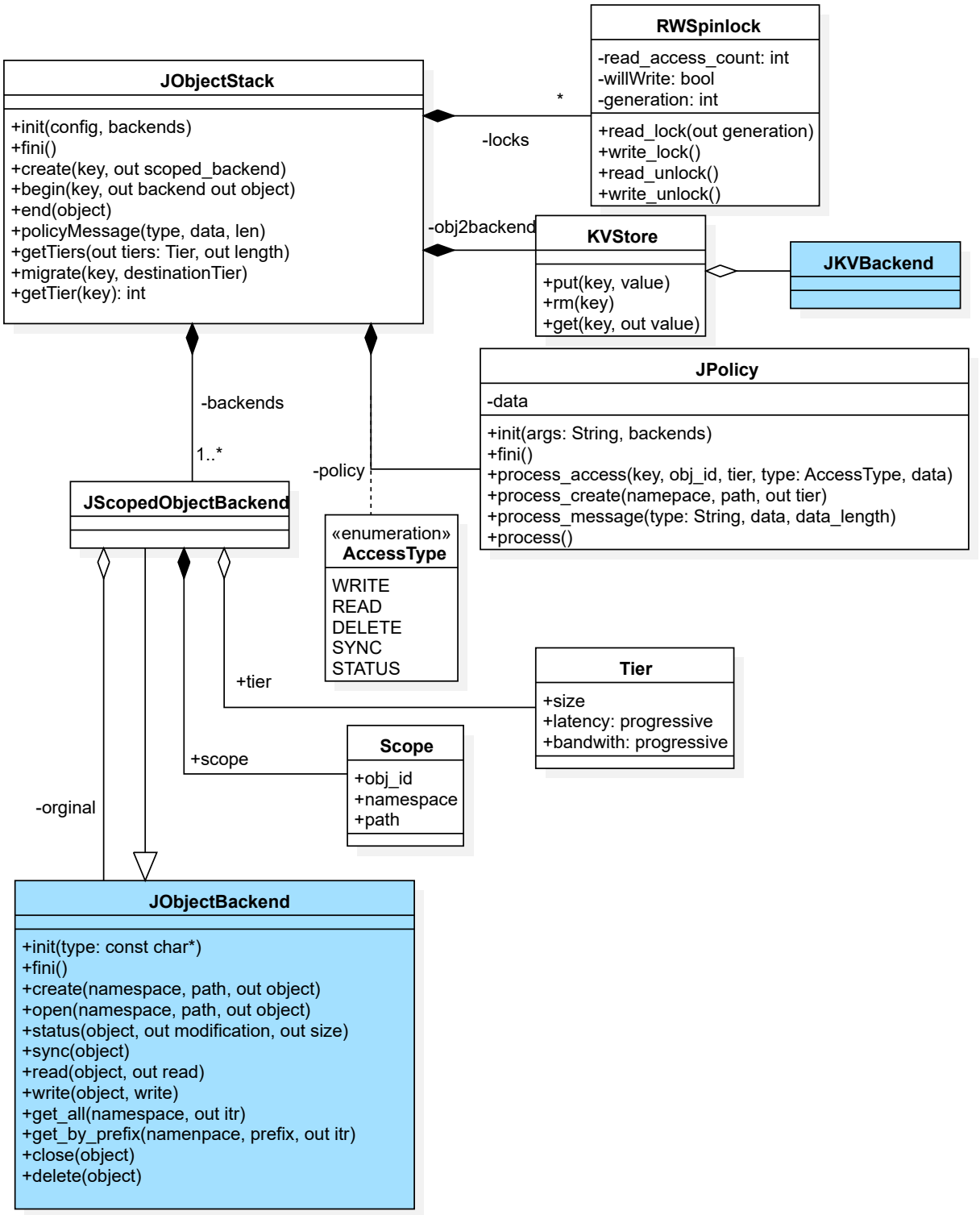


Figure 4.1.: Class diagram for policy module integration. Blue classes already exist in JULEA.

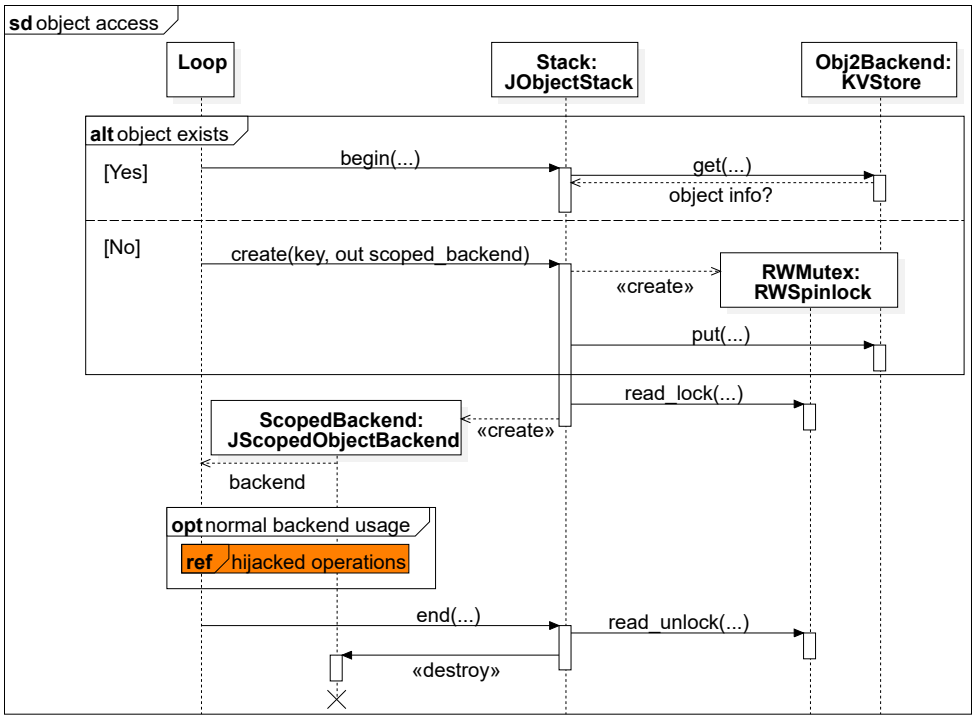


Figure 4.2.: Diagram for accessing object backend through `JBackendStack`, referenced diagram is figure 4.3

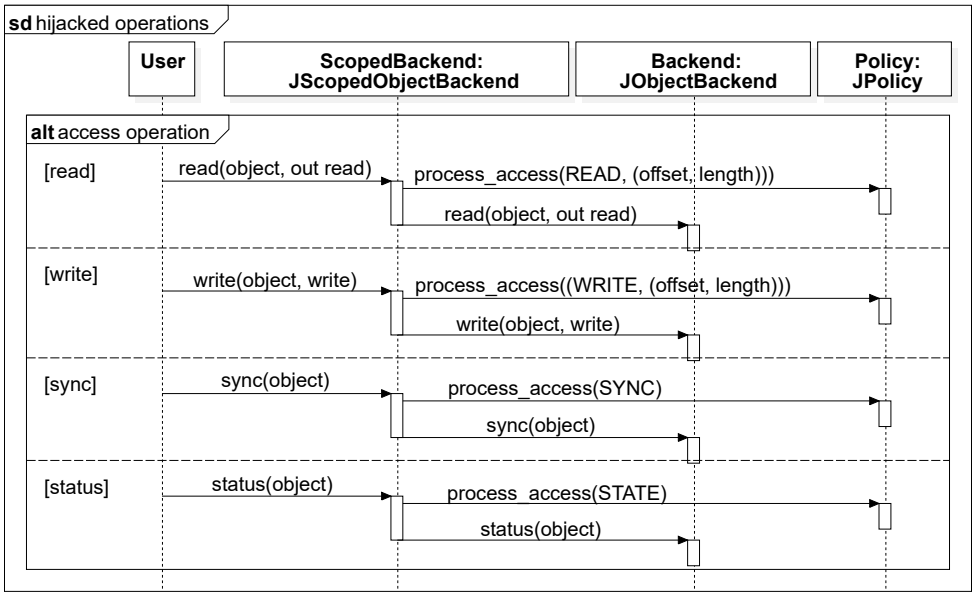


Figure 4.3.: Hijacking scheme for signaling backend operations to Policy.

Abstraction On significant problem that remains is how to issue a `process_access` call for object access while only minimally altering the backend handling. The solution is inheritance. If the `JBackend` functions to track relevant access are overwritten, the server code would be unchanged. To archive, this `JScopedObjectBackend` is a child of `JBackend`. Besides the over-written access functions, this also stores the object key of the currently accessed object to provide it as needed (see fig. 4.1). This over-writing of access functions allows *unchanged* backend code and the *convenience* to add new backend functions without altering the policy mechanism. If new functions are added to a backend that is not directly related to `access`, there is no need to modify policies-related code since that functions will be inherited. The relevant operations are propagated to the backend as displayed in figure 4.3.

An alternative solution could be to introduce a new function for each access type, which calls the intended backend function and the corresponding policy function. However, this method would results in many changes. Every new backend function would have to be implemented in the `JBackendStack`, which would define the same method multiple times and prolong rotation times Any other solution would interfere even more with the existing design and codebase.

Backend iteration The iteration capability of JULEAs object backend allows iterating over all objects stored on an object server while filtering by namespace. Iterating over all objects can be achieved by iterating over each tier since there are no order constraints. Migrating objects during an iteration is problematic because an object could be visited multiple times. However, this iteration only returns the object paths, which are stored as in the local KV-store. An iteration over all objects is the same as an iteration over the keys in the KV-store since JULEAs KV interface *already* has the same iteration functionality. This way, it offers a simple and *efficient* iterator.

An alternative would be to block all migrations for the iteration duration. This solution would keep the changes less invasive since it would only introduce a new scope (migration lock). Nevertheless, this would disable the HSM while iterating. Therefore the iteration via the KV-store is chosen. It is more invasive but circumvents possible latency at operation start if a migration is already running and will not clog the policy execution.

Remaining structures To cover the remaining parts displayed in figure 4.1. `Scope` is meta-data associated with an object access scope, namely the namespace, path, and `ojbect_id`. The `object_id` can identify objects more *conveniently*. Since the index is dense, it can be used to index an array that contains data related to the corresponding object instead of creating a string lookup based on namespace and path.

`Tier` is a collection of data describing a tier because there is no way in JULEA currently to fetch the metadata of an object backend. These arguments must be provided via the configuration file. An alternative is to pass custom metrics via the parameter list directly. The decision to introduce this data object is based on the assumption that bandwidth, latency, and capacity are standard parameters to describe storage and may be available through JULEA someday (Lehner, 2017). Moreover, it creates a better readable configuration file and is an entry point for later universal tier-related data.

It was also considered to measure access timing and deduce tier properties. The problem here was that a sophisticated approach will introduce latencies, and the naive approach needs data

and non-data access to calculate these values. In that case, the data would also not be available from the start, which leads to a new list of problems.

Last is the `AccessType` described in section 4.1.2. It is used to differentiate access types without implementing different functions. Policies can use data detailed as they need and access types of interest.

4.2. Implementation

In the previous section, the design was explained. However, further details need to be specified to implement the HSM-mechanism. This section presents the remaining considerations and their chosen solutions. First, in a top-down manner the integration of the `JBackendStack` and infusion of `JScopedBackend` in the current JULEA codebase is explained. Afterward, the implementation of the policy execution loop is discussed, followed by the challenges of the migration process.

4.2.1. Integration with Current Object Backend

To enable integration with the current object backend changes in the configuration process are necessary to allow the use of more then one object backend. Since GKeyFile files support argument lists[3] that can be used to pass more than one backend. This will reduce mandatory modifications without invalidating old ones (see lst. 4.2). In addition, capacity, latency, and bandwidth can be configured for each tier. These parameters are optional and can be omitted to avoid incompatibility. The chosen encoding is a suffix representation[4] to keep the configuration files human-readable.

Listing 4.2 Current JULEA configuration for object backend (left), vs definition of multiple object backends in new version (right).

```
[object]
backend=posix     | backend=posix;posix
component=server  | component=server
path=/tmp/object/ | path=/tmp/object;/mnt/fast
                  | capacity=100MB;500MB
                  | latency=50us;20ms
                  | bandwidth=1GB;200MB
```

In addition, a new configuration section for the policy is also needed to extend the existing object backend configuration. The name of this section is `object.hsm-policy` to highlight that this policy is a HSM policy for the object backend (e.g. lst. 4.3). It contains the policy module identifier (`policy`), similar to `[object].backend`. The configuration for the utilized local KV-backend is based on the `[kv]` configuration section. A list of arguments (`args`) is used to initialize the policy as described in section 4.1.2.

[3] `g_key_file_get_boolean\double\integer\string_list` to parse a `;` separated list("GLib.KeyFile.Get_string_list", 2022).

[4] Like 1GB for 1 Gigabyte or 1µs for one micro second

Listing 4.3 New configuration group for policy.

```
[object.hsm-policy]
kv_backend=leveldb
kv_path=/tmp/julea-1000/obkv
policy=lru
args=200;5
```

`JConfiguration` needs to be extended to these new fields. For the arguments list and list of backends/paths, the GLib build in `g_key_file_get_string_list` is used. This returns an array of strings, which will be converted to a `JList` of strings to avoid having a new list type in the server code section. The tier attributes will be stored as `uint64_t` for easier handling later on. The conversion between the "human-readable format" and absolute format will happen in a new function in `j_helper`. A new anonymous substructure in `JConfiguration` is also created, storing the different parameters as strings to reflect the new configuration group.

Next is the integration into the server code. Currently, the object backend is stored as a pointer in `jd_object_backend` and loaded with `j_backend_load_server`. To keep changes minimal `jd_object_backend` is changed to `jd_object_backends` an accepts a `JList` of `JBackend`. This list will then be populated with `j_backedn_load_servers`, which will replace its former singular version. With this, the initialization code from the server is nearly untouched but now able to work with multiple backends.

The next decision is the placement of `JBackendStack`. It could be declared in `server.c` as the previous backend but this would violates the object orientation paradigm. Since it is only needed in `loop.c`, to handle incoming messages, `JBackendStack` should be managed in `loop.c`. However, since there are currently no other resources in `loop.c` that need to be constructed/destructed, a `init/fini` function pair is needed. The only function currently in `loop.c` is `jd_handle_message`, the added initialize and finalize are called `jd_handle_init` and `jd_handle_fini`.

The last step is to replace each usage of `jd_object_backend` with a scoped access to the backend corresponding to the current object as seen in figure 4.2. Which is as simple as putting a `j_backend_stack_begin` or `j_backedn_stack_create` before each `j_backend_open/create` and a `j_backend_stack_end` after the last operation.

To support iterating over all objects the object backend iteration functions is also implemented in `JBackendStack` and named `j_backend_stack_get_all`, `j_backend_stack_get_prefix` and `j_backend_stack_iterate`. Their implementation then uses the KV-backend iteration mechanism to iterate over the entries in the `JBackendStack` internal KV-store as described in section 4.1.3.

4.2.2. `JBackendStack`

C is a language with a long history but without convenient features. Therefore `JScopedObjectBackend` can not inherit from `Backend`. Nevertheless, *C* defines that the order of member declarations is reflected in their memory layout. Furthermore, no unnamed padding is

allowed before the first member (Secretary, 1999)[5]. If `JScopedObjectBackend` has a `JBackend` as the first member, then the memory layout of the first `sizeof( JBackend )` bytes would be the same as for `JBackend`. As result can the address of a `JScopedObjectBackend` be interpreted and used as a pointer to `JBackend`.

To redirect the object backend access calls, the function pointer of the `JBackend` part will change to call a function that executes the backend access and sends a signal to the policy. A copy of the original `JBackend` is kept to store the original functions. This way, all `j_backend_object` calls will be redirected to the corresponding policy functions, which can call the initially intended function because they reference an unchanged `JBackend`. All object access calls cascade as shown in figure 4.4.

The implementation of `JBackendStack` requires a slight change in the existing object backend interface. All access functions of the object backend take a pointer to a slice of memory allocated by their `init` function.Therefore it is impossible to append their data without extensive code changes. A less *invasive* solution is to alter the signature of the object backend functions to take a `JBackend` pointer instead. This way the backend function can access their data via `JBackend.data`, while the remaining code can work on `JBackend` and its extensions `JScopedObjectBackend`.

Because the `JScopedObjectBackend` contains object-specific data, each scope must create a new one and clean it after `end`. This is not ideal, but the growing efficiency of `malloc` (Durner et al., 2019) is not a concern at this early stage. Since each client can only have one access at the time, it would be sensible to allocate space once a new client connects to the server and reuses this space. Each time, the space would still be filled with the backend meta data for the corresponding tier, but copying a few hundred bytes has a smaller impact allocating additional memory. Nevertheless, to implement this, the client connection process must be reconstructed by either giving each client a unique ID or adding the possibility for client scoped memory, or alternative solution. However, solving this issue is out of scope for this work.

4.2.3. Policy Communication and Execution Loop

Processing As discussed earlier, in section 3, the execution of the HSM should run in a separate thread to allow asynchronous data movement. The function `process` is used to hand these resources to the policy. This function will be called from a dedicated thread and minimize interference with client handling. As shown in figure 3.1, continuously running the data-mover dose not create optimal results. Instead, using a separate function can provide different waiting schemes for the policy data-mover. The gain of directly implementing these schemes in JULEA would be that the policy code would get simpler and, therefore, easier and faster to write. In addition, it is much easier to terminating the program in a defined state.

Possible simple policies for executing the data-mover are a flat delay between data-mover runs, or running it after a fixed amount of hints/object accesses (Dong et al., 2016). Currently, neither of them is implemented. However, to allow an easy implementation in the future, resource handling is already constructed that way.

[5]See section 6.7.2.1 paragraph 13

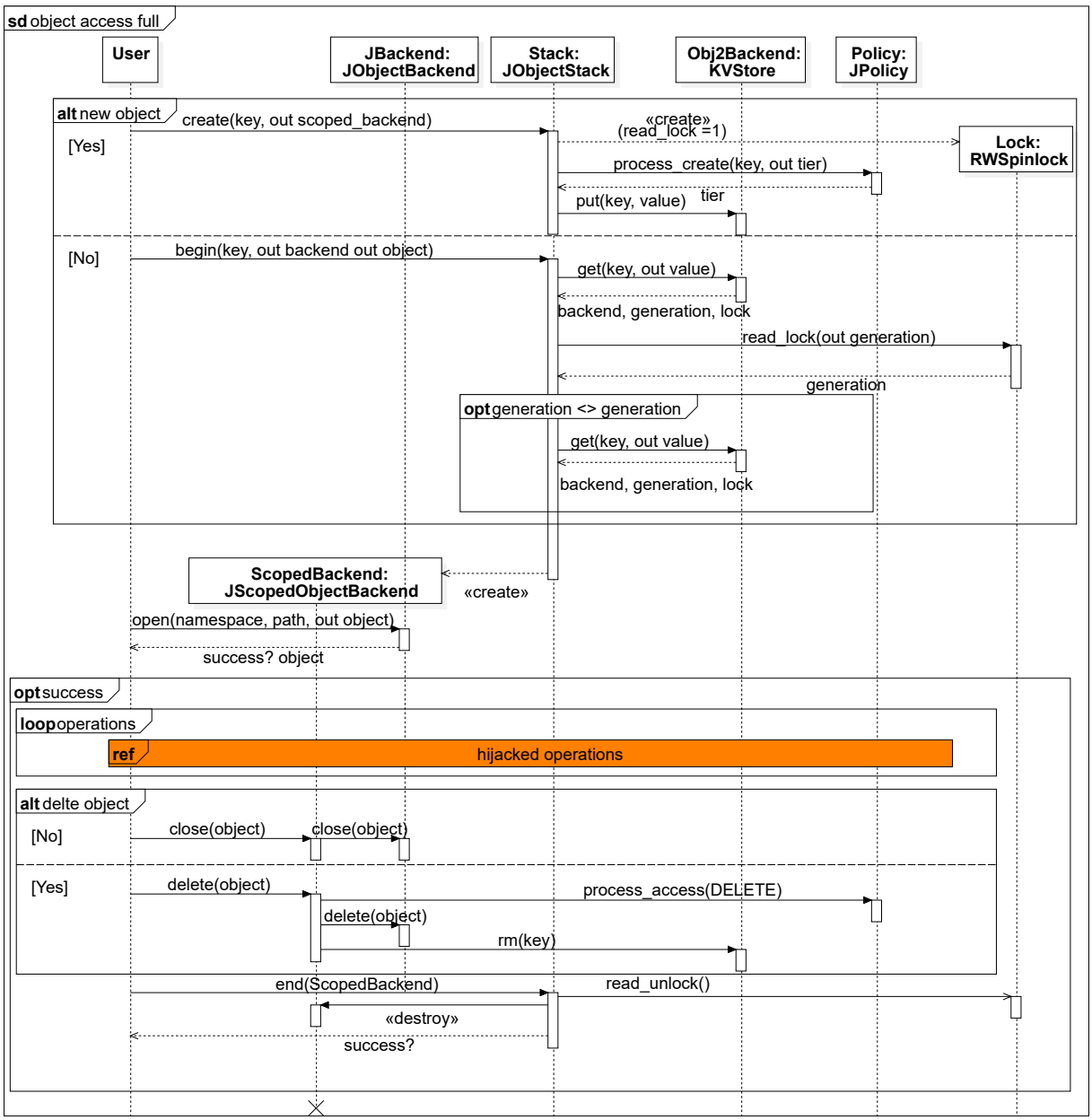


Figure 4.4.: Detailed sequence diagram for accessing a JULEA object backends via `JBackend-Stack`, for referenced diagram see figure 4.3.

The `process` function will be called in a loop until JULEA receives a `SIGKILL` or shuts down for other reasons. As results it is a requirement of `process` to return regularly to allow termination.

Object access handling Besides processing resources, a policy also needs information to make decisions. The first data fragment can be received via the optional `[object.hsm-policy].args` configuration as a list of strings. This is an easy way to pass parameters to the policy without concerns that the configuration name used by the policy may also be used by JULEA.

The next and most important type of data is the access notification. If object access is issued, the time to process this access should be minimal to reduce latency. Because JULEA has one thread per client, processing in that thread will potentially delay a follow-up request. The conclusion is that the functions `process_access` and `process_create`, which interfere with regular data access, should be fast as possible. Especially `process_access` is only intended to produce light-weight hints.

`process_create` has a unique role, because in contrast to `process_access`, its results are used to determine on which tier to store the new object. Also, it will properly set up data for later tracking inside the policy.

Hints managing A structure that allows collecting hints asynchronously and pressing them in a separate thread would be an async-queue, for example, `GAsyncQueue`. Then each `process_access` could push a message to that queue, and in `process`, this queue would be read. Example usage of that can be found in the LRU-policy described in section 5.3.

As seen in figure 4.3 for each object access, the policy hinting function is first called before executing the corresponding backend function. This is done to maximize the processing time of the data mover with the new knowledge. Because `process` runs on a separate thread, this order should have no impact on the duration of the object access. Since data access is a relatively slow process, the data mover can perform in the meantime tasks like clearing space for the object, which should be promoted.

Messages The final communication channel to the policy is by sending a direct message. A message consists of a string-type descriptor and a blob. The string type descriptor is chosen to allow easier debugging and motivate a more human-readable data exchange. This direct channel can be used, for example, to manually promote objects or, like proposed from Ghoshal & Ramakrishnan, to execute a pre-calculated migration or prefetching issued from the workflow manager (Ghoshal & Ramakrishnan, 2021).

It should be mentioned that the policy also has access to `JBackendStack` and, therefore can create/read/write/status/delete objects. Besides the status fetch to check the size of objects, data modification is not in scope for a typical HSM. However, this option is provided to offer more *flexibility* and an easy way to store policy data.

Lastly there exists no mechanism to inform the policy that a shutdown will happen. For storing data on shutdown, which is often necessary to properly function after a restart, the only place is the `fini` function. At this point, only the functionality of the private KV-store is available.

4.2.4. Handling Migration

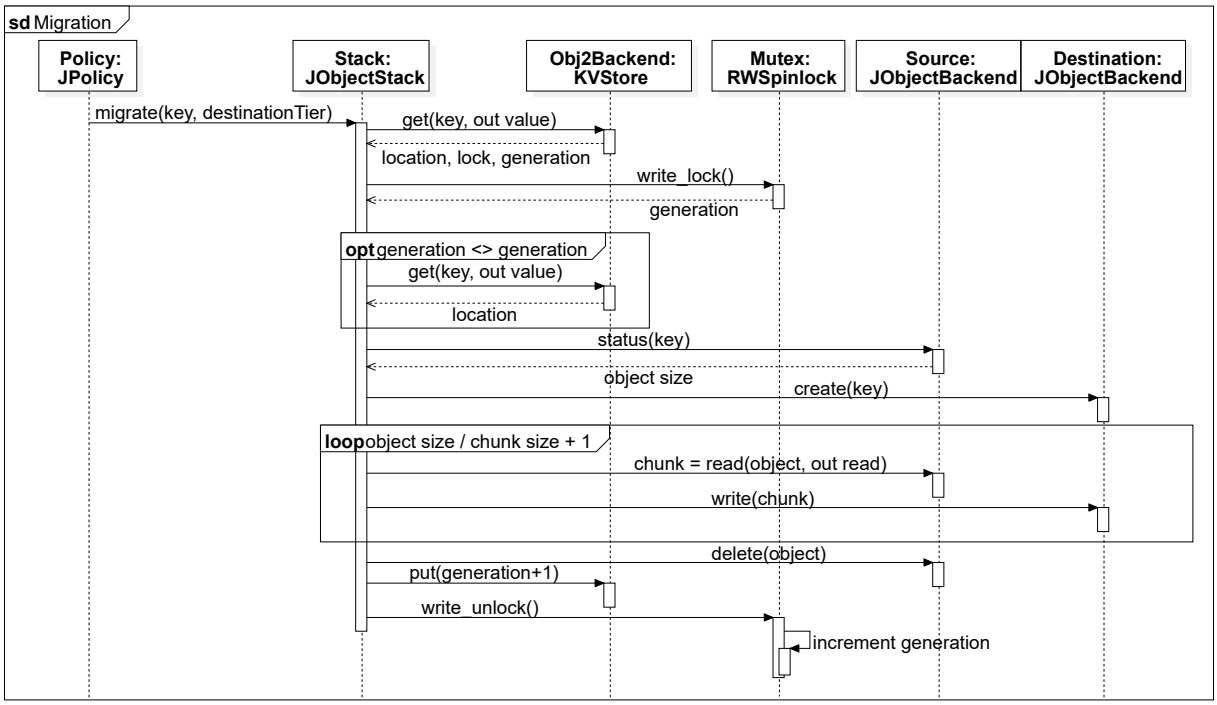


Figure 4.5.: Migration process between two object backends via the `JObjectStack`

Mapping objects to tiers is the main purpose of a HSM. Besides setting the initial storage tier with `process_create`, a mechanism is needed to move data between tiers. This mechanism is used with `migrate`, figure 4.5 gives an outline of how it works. Important to note is that if a migration is running, this process has exclusive usage to avoid invalid access. It would be possible to still allow state and read access, but it would introduce more complexity since the linked backend would then be changed in a scoped operation. If migration is running, there is no data access to keep things simple and functional.

Access control A read-write lock is used to guard the objects to ensure exclusive usage on migration while still allowing shared usage for read and write operations. It would be handy to use the `GReadWriteMutex`, but GLib does not specify in which order queued read and write locks are served ("GLib.RWLock", 2022). The priority of migration access is important since the object accesses may overlap for data with high demand, and therefore the object is never free to migrate. However, if the migration access has priority, it may briefly delay access while the migration is running, but all following access benefits from the faster tier. The simple read-write spinlock presented by Fuerst is implemented to have a well-defined behavior (Fuerst, 2018).

First, this read-write spinlock implementation grants that no more shared access is permitted if exclusive access is queued. This could potentially lead to significant latencies but prevents starvation of the migration. This design was chosen, because scenarios with many shared and few exclusive migration accesses are common where hot data is migrated. For these kinds of scenarios the performance of this simple lock is similar or better to more complex locks (Fuerst, 2018). Secondly a simple implementation leaves less room for error, and finally, implementing

a read-write lock with mutex would use semaphores not provided by GLib and would lead to a even more complex implementation.

Migration The process of migration (see fig. 4.5) contains acquiring the exclusive access, creating a new object at the destination, and then copying the data in chunks from the source tier. After the transfer is finished, the original object will be deleted, the generation of the object increased, and the new metadata is stored in the local KV-store. The last step then is to release the lock.

A generation count is used to recognize outdated data. An object entry contains a generation count and a reference to a read-write spinlock. Acquiring a lock yields a generation. If the generation matches the entry currently stored, the entry is still up to date. If the lock yields a different generation, the new entry must be fetched from the KV-store. This mechanism allows reducing KV-accesses. Otherwise the KV-store would have to be accessed before requesting the lock, in order to find the matching lock, and again after the lock is granted to ensure that the data is up-to-date. Since an entry only stores the data placement, not the data itself, only migrations can change the validity of an entry. Therefore, if no exclusive lock was in place when a read lock request was processed, there is no need to fetch the entry from the KV-store again.

At its core, migration is a straightforward process, involving guarding shared storage. The problem is that migration will change the object's `last accessed` information. It is generally undesirable to modify objects on migration. However, since JULEA's object backend interface does not provide a way to modify object metadata, this can only be solved by storing access data while migrating in the KV-store and hijacking the status request to inject the correct data.

Chapter 5.

Evaluation

In this chapter the new HSM mechanism is evaluated. It will be analyzed how high the system's overhead is and if simple policies can result in plausible timings. Based on this functionality of the interface and the effort to write the policies is evaluated. First, different use cases and tiering scenarios will be described. Then, the hardware is used to execute the benchmark and the policies. Finally, the benchmark results will be presented and interpreted.

Different policies will be tested under various scenarios to confirm the general functionality of the HSM module. Three dimensions are chosen to construct these scenarios for a comprehensive functionality test – the use case, the available tiers, and the policy used.

5.1. Benchmark Scenarios

The selected uses cases are first, a partial equation solver (`partdiff`) which writes the current state as checkpoint and second, an Enzo test case as a scientific application example. "Enzo is a community-developed adaptive mesh refinement simulation code, designed for rich, multi-physics hydrodynamic astrophysical calculations" ("The Enzo Project", 2022). Moreover, it uses JULEA via the HDF5 interface. In addition, a constructed read script is used, which will read checkpoints from `partdiff` since no out-of-the-box JULEA compatible application to test read properties was found. This script will read checkpoints and calculate the sum of the main diagonal to simulate a data-heavy workload. The data is fetched in two phases: the metadata (e.g. matrix size), and the matrix data. Furthermore, to provide a behavior that allows caching, these two phases are executed in batches of ten, including ten metadata reads, ten matrix reads with calculations, etc.

For further insight into performance behavior, the `julea_benchmarks` are also used. This series of micro-benchmarks is set around creating and deleting empty objects to measure management performance. The other set is to check reading or writing of many small objects. A detailed list of this benchmark can be found in the project's source code[1].

The scenarios are constructed out of 2 tiers to keep the policies simple. At first, the two tiers are the local SSD and the network PFS (CephFS) to simulate a local BB setup. The local SSD is also the fast tier in the second scenario. The slow tier is an artificial slow drive with latencies of 50ms created with SlowPokeFS (SPFS). This slow drive constructs a scenario where the second tier is strictly worse. In contrast, PFS and SSD each have their advantages, especially with high network performance.

[1] https://github.com/julea-io/julea/blob/master/benchmark/object/object.c

There is also a scenario with only one tier for each type of memory used to classify these values. This will allow determining the performance gain of the different policies used.

5.2. Hardware/Cluster

The tests and benchmarks were run on two nodes of a computer cluster, one as the server the other as the client. This separation was done to ensure that other I/O-output from the client applications would influence the server object backend as little as possible. The nodes are identically constructed as follow:

CPU: 1x AMD Epyc 7443 2.85GHz 24 Cores
RAM: 128GB
Network: 100GB-Ethernet
Local NVMeSSD: 256GB
SlowPokeFS: FUSE on local NVMeSSD which introduces a delay to each I/O-operation (Schoenmakers, 2013, April 8/2021).
Network storage: CephFS on 10 storage nodes. Each equipped with multiple NVMeSSDs, typically four `INTEL SSDPE2KX040T8 (P4510) 4TB`.

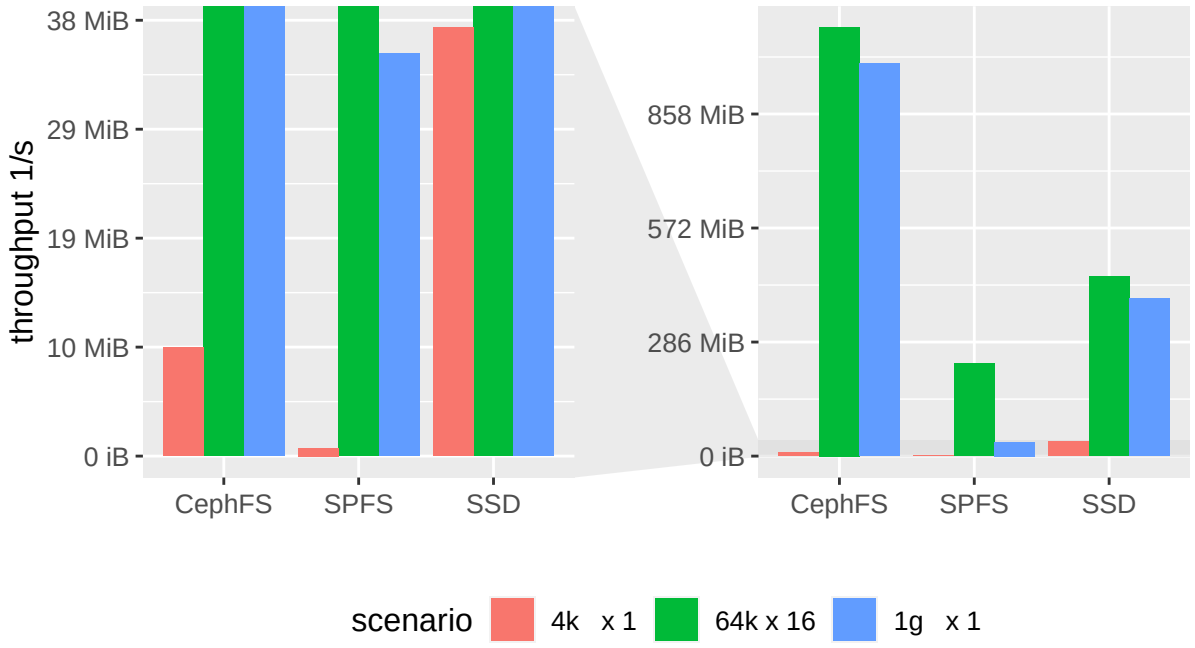


Figure 5.1.: Performance tests on storage media used as a reference in further evaluation tested with random read-write access in different scenarios. There are sixteen streams with 64kB accesses, one stream with 4kB accesses, and one stream with 1GB accesses. As intended, has SPFS (SlowPokeFS) performed the worst, especially for small accesses.

All applications and servers ran exclusively on an individual node to provide results with low variance. To set expectations, first, the I/O-capabilities were tested. Therefore the latency and the throughput for different access patterns on each storage were measured in addition to the network bandwidth between these two nodes.

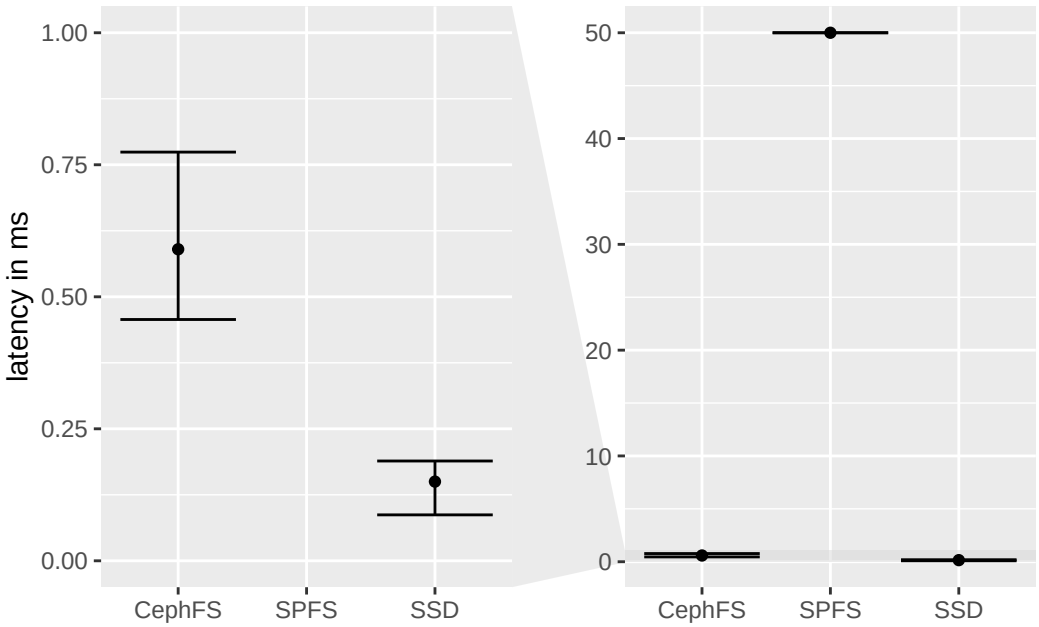


Figure 5.2.: Latency measured for different storage media: the latency for access on the network storage is significantly higher and with a higher variance than the local SSD. For SlowPokeFS, the latency is 50ms as configured with an absolute variance comparable to the SSD.

Storage media used are network storage, a built-in SSD, and an SSD storage artificially slowed with SlowPokeFS, which introduce a flat latency to each I/O-operation of 50ms. Three random read-write configurations were executed to measure the throughput of this media.

- 1 I/O-stream, which writes and reads random blocks of size 4kB to test for small data access
- 16 I/O-streams which each write and read a random block of the size of 64k to test parallel scaling with larger packages
- 1 I/O-stream, which writes and reads random blocks of size 1GB to test sequential file access performance.

The `fio` (Axboe, 2012, October 22/2022) tests results are shown in figure 5.1. As expected, the SSD's bandwidth for random small accesses is significantly higher than for PFS or SlowPokeFS. For high parallel access and large access, the local storage medium is outperformed by the PFS since the network bandwidth is larger than the bandwidth of the SSD (see tbl. 5.1), and CephFS seems to have a good load balancing. The SlowPokeFS is, as intended, by far the worst in each category, which will give insight in cases where the performance of SSD and PFS is indistinguishable.

To test the latency, `ioping` was used, the results for different media are plotted in figure 5.2 (Khlebnikov, 2015, March 29/2022). As seen, the latency variance of the PFS is much larger than of the SSD, which likely results from the network and metadata handling. The SSD has significantly lower latency, and as intended SlowPokeFS shows the highest latency for more detailed benchmark results.

Table 5.1.: Shows connectivity between the benchmarks nodes in GBits/s. The maximal bandwidth per channel is 40Gbits/s, and the total bandwidth per connection is around 100Gbits/s. These values were measured using `iperf`.

#connections	bandwidth total	channel 1	2	3	4	5
1	40.6 =	40.6				
3	92.6 =	28.1 +	31.6 +	32.9		
5	93.9 =	18.6 +	18.7 +	19.0 +	18.5 +	19.2

Finally, the connectivity between the two nodes was tested with `iperf` ("Iperf3", 2022). A one-channel transfer was made to test the maximal available throughput per channel, a transfer with three channels on three cores, and another transfer with five channels on five cores were performed to measure the connection's maximal bandwidth. As seen in table 5.1, the maximal bandwidth per core is 40Gbits/s, and the maximal total bandwidth is 100GBits/s, which means that all access to the PFS is capped at 5GB/s per core or 12GB/s totally. These bandwidth caps were sufficiently high to not impact the test results, since they are above the measured performance values (see fig. 5.1).

5.3. Policies

Three policies were implemented to evaluate the constructed interface and check if the mechanism provides the required calculation times and performs migrations as intended. This way the capabilities of the designed interface and whether the locking leads to starvations or deadlocks can be tested.

The first policy[2] is a dummy policy. This policy will put everything on the same tier and never migrate. This allows recognizing the overhead for the new redirect layer and other additional calls compared to JULEA without policies. It also allows straight support for only one object backend.

The second policy is a default LRU[3]. Because requesting used memory is not a simple task, the fast tier size is defined by object count instead of size. The impact for the selected use cases is small since `partdiff` writes equally sized checkpoints, and the output of the `Enzo` test case writes objects of 33 to 50kB. This also allows for a more straightforward implementation and, consequently, less latency until the migrations starts. The number of objects allowed for the fast tier can be passed as an argument. It will be tailored to the task, e.g., if the `partdiff` object is twice as large as an `Enzo` object, it will only have half the number of slots.

The third policy is a write buffer policy[4], which places new objects on the fasts tier, if possible, and keeps them there for a fixed amount of time. After this time interval, the object is moved to a lower tier. This migration timer is implemented to account for the fact that creating and writing an object are two sub-sequential commands, as well as that large objects may be split into multiple write commands. The time interval will be different for each use case and

[2] backends/object/policies/test.c
[3] backends/object/policies/lru.c
[4] backends/object/policies/write buffer.c

depends on the time between data bursts. The timeout was determined based on access charts as shown in figure 5.7.

Aside from the different behavior, depending on the complexity, the calculation times for each of the three policies are also different. While the dummy policy only provides empty functions, the write-buffer-policy needs the object creation to queue the migration of an object. In contrast, the LRU-policy requires a logic for placing the object on its creation and for processing when to migrate which objects.

5.4. Tests and Evaluation

The test cases were run separately on two exclusive used nodes to minimize variance. Despite this, there was high variance in the measurements, especially for the micro-benchmarks. All data should therefore only be interpreted qualitatively and not quantitatively. If more data is moved, the performance difference between SSD and CephFS shrinks, as seen in figures 5.6b and 5.3b. The implementation of the different policies is evaluated based on the analysis of benchmark results, test applications, and access patterns.

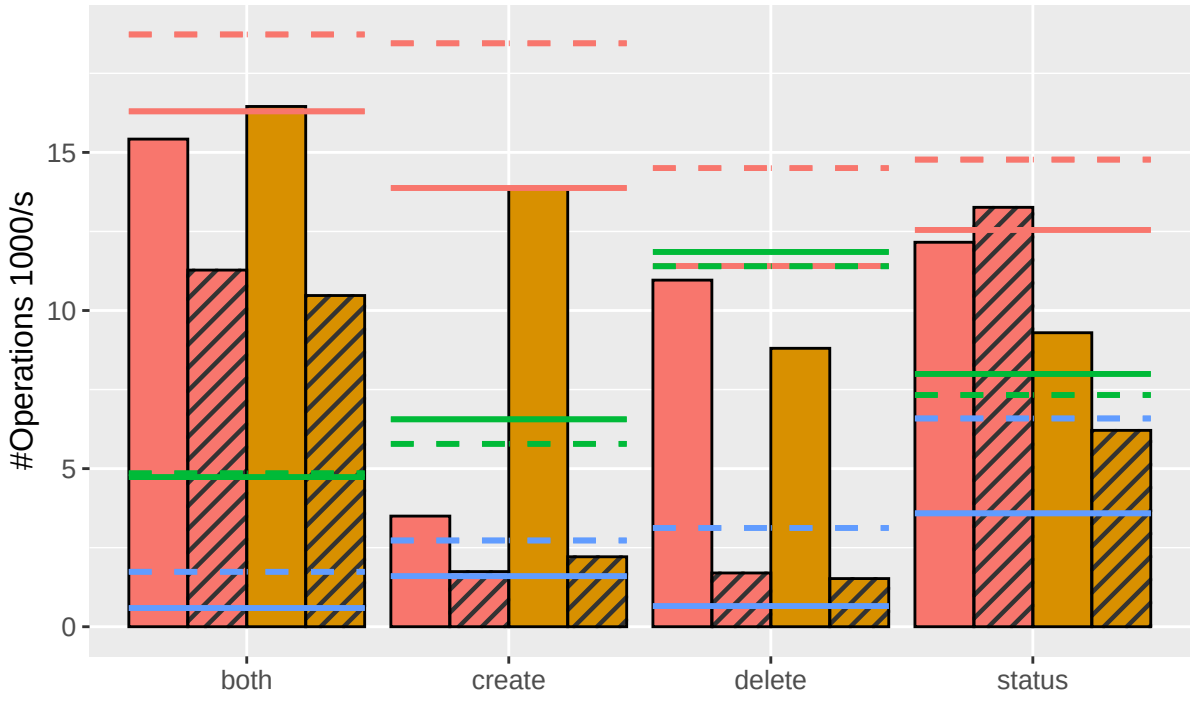
The KV-store in each scenario was placed on the local SSD based on the typically KV-store storage requirements described in chapter 1. Therefore, all available server-sided KV-stores supported by JULEA were tested with the JULEA-KV-store micro-benchmarks. Based on the results shown in figure 5.4, levelDB was selected as backend for the KV-store. LevelDB offers the best `put` and `delete` performance while still being close to the best `get` performance of lmdb.

JULEA micro-benchmarks

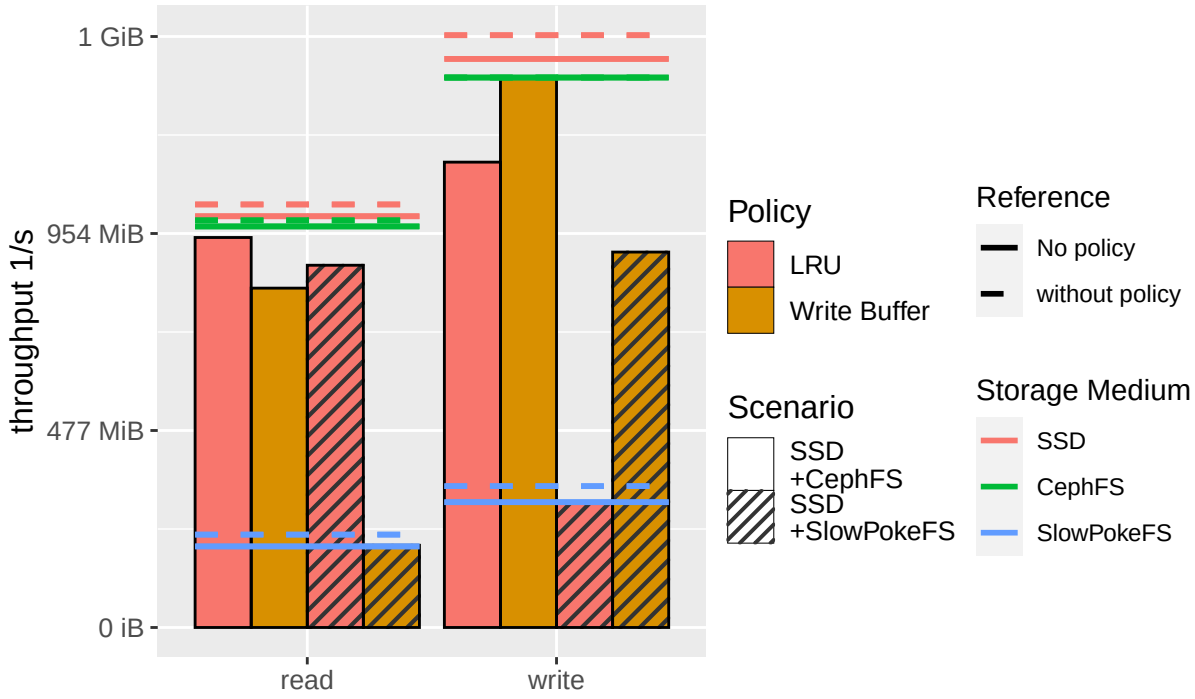
Figure 5.3 shows the results of different micro-benchmarks. Each benchmark was executed with the dummy policy and JULEA without policies on each storage media, represented as solid and dashed lines. In comparison, the performance of the remaining combinations is visualized as bars. The difference between solid lines and their corresponding dotted lines shows that the added abstraction layer with the policy system significantly impacts the performance of simple create/delete/status operations. This may result from the additional KV-store access for each operation and the overhead for acquiring the lock. Based on figure 5.4, the mean duration of a levelDB access is 13µs. For the local SSD, the time difference between the dummy policy and without policy varies between 18 and 20µs[5]. LevelDB, which was used as KV-store, is probably accountable for ¾ of the overhead in the fastest scenario.

The object backend micro-benchmarks also show that migration can harm the performance, as seen for the LRU policy, where the *create* performance is halved compared to the reference without migration. Only one object is created for the *both* and the *status* benchmark and then immediately deleted or repeatedly requested. The LRU *status* benchmark shows that the slow tier does not influence the performance because the object is kept in the fast tier. For the status benchmark with the `writebuffer`-policy it can be observed that the data is moved to the slow tier and impacts the remaining benchmark negatively after the timeout.

[5] $t = \frac{1}{\#\text{Operations/s}}$



(a) Micro-benchmarks for object managment without data movement.



(b) Micro-benchmarks for writing/reading many 256KB objects

Figure 5.3.: JULEA micro-benchmarks on different policies, for different scenarios, where horizontal lines represent the performance of unused policies (solid lines) and not implemented polices (dotted lines). As neither of these policies is fitting for continuous interaction, they perform worse than a policy that does not migrate.

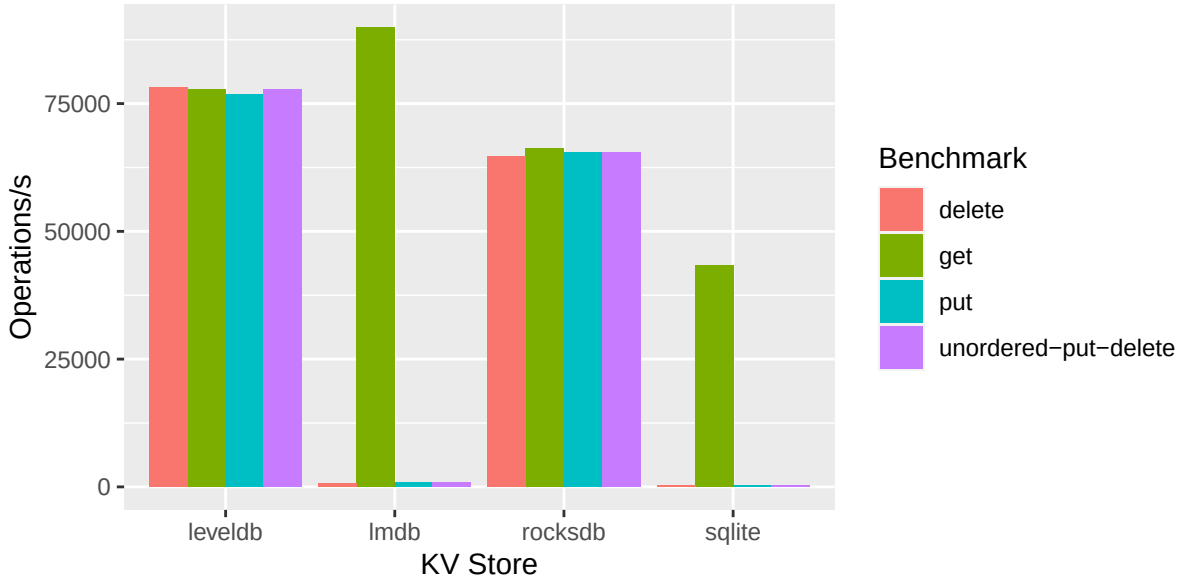


Figure 5.4.: Micro-benchmarks for KV-store performance

The write and read performance are only slightly different for SSD and CephFS. Additionally, the overhead of the policy management is also less noticeable, because write and read operation have longer execution times. For the combination SSD and CephFS, both policies reduce the performance since the migration will only lead to more load on the I/O-devices. Furthermore, it is noticeable that the `writebuffer`-policy performance is better for write access for SlowPokeFS, which is plausible.

JULEA supports batching commands. Batched commands are sent at once instead of separately, which reduces the summed network latencies and, therefore, the total execution time. All micro-benchmarks were executed with separated and batched commands. The batched execution increases the performance significantly, but the relationship between the policies remains the same. Since it is only a qualitative analysis, the data was removed from the figures for clarity.

In conclusion, the high-frequent management of small objects negatively impacts the additional abstraction overhead and locking. Also, if smaller objects are written and read, selecting a suitable and fast policy is essential to maximize the gain.

Application Performance

When analyzing the application performance, at first it was noticed that Enzo's performance is similar on all storage media, which is probably a result of their asynchronous write-out, and small object sizes (see fig. 5.5).

`partdiff` does not provide asynchronous checkpoint writing and therefore depends more on the storage performance. The `writebuffer`-policy performs independently of the slow tier, because the objects are entirely written to the fast tier and then migrated to the slow tier, which finishes before the next checkpoint is written as seen in figure 5.7a. In one scenario the LRU provided no speed up and in another decreased the performance drastically for SSD and

SlowPokeFS. These results are caused by the limited performance and high latency (figs. 5.2, 5.1) in combination with many unnecessary migrations (fig. 5.7c).

For the reading application, it can be seen that CephFS has a good prediction of which data are of interest and high throughput since the performance of SSD and CephFS are the same. As aspected does the `writebuffer`-policy not impact the read performance. LRU on the other hand provides a small speedup, through prefetching.

For more realistic use cases, the selected policy can significantly influence the results, positively as well as negatively. The costs for the additional abstraction are not as significant as they were for the micro- benchmarks. Also, asynchronous storage management for non-data-heavy applications can reduce the impact on the selected storage solution.

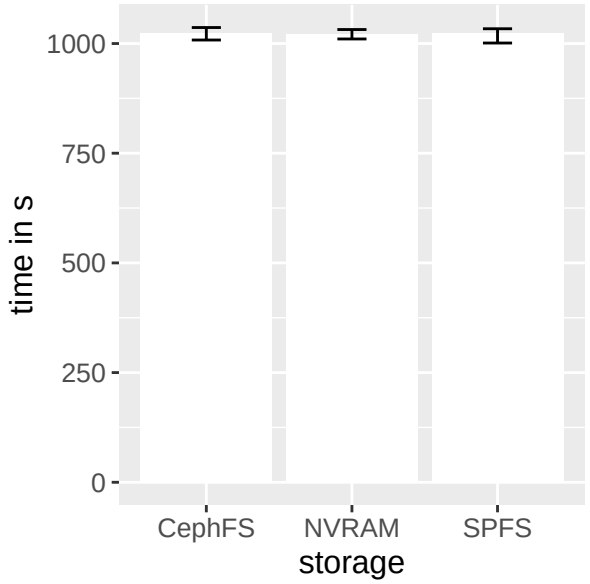


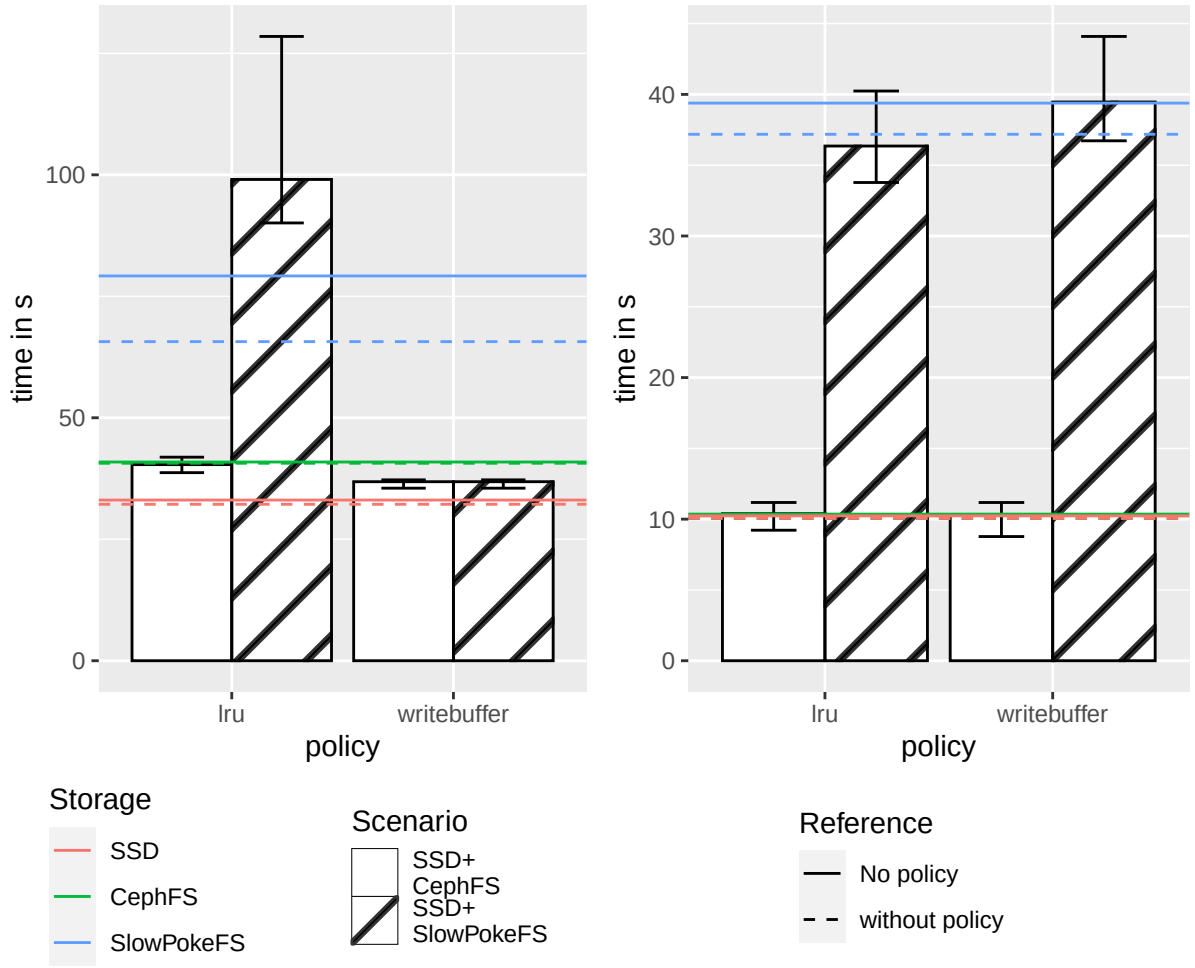
Figure 5.5.: Total execution time of `enzo` scenario `Hydro-3D/CollapseTestNonCosmological` on different storage media.

Policy Access Pattern

In figure 5.7 the access to the object storage is visualized for a pair of `partdiff partdiff_read` execution with the LRU or `writebuffer`-policy. The beginning of different operations is denoted with a corresponding mark, where the color indicates on which tier the accessed object is located. The tier where the object is migrated to is indicated for migrations. Each object's first and last access is connected with a line to visualize its lifetime.

Figure 5.7a shows that for the `writebuffer`-policy in most cases, only one object is on the fast tier at a time. This tailored timeout improves the performance, as seen in figure 5.6a, while only requiring a small part of the BB. By increasing the timeout, the space needed on the BB increases too. Figure 5.7b shows that five other objects were created and stored in the BB between creating the object and demoting it when the timeout is increased from 100 to 15000ms, which results in five times the memory consumption.

Figure 5.7c shows why the `partdiff` performance of the LRU-policy is worse than the reference case. The fast tier is fully populated; therefore, all new objects are moved to the slow



(a) `partdiff` calculation with high amount of blocking storage writes. Writebuffer hides slower tier completely, while the LRU policy even decreses performance for SlowPokeFS.

(b) `partdiff_reader` which reads data in chunks. Performance between SSD and CephFS is non existing, but LRU increases the SlowPokeFS performance.

Figure 5.6.: Time plots for different uses cases over different policies, with horizontal lines marking the performance without policies. As a reference, the dotted line represents the performance without policy management overhead. In general, the policy introduces overhead, but when used right it can increase performance.

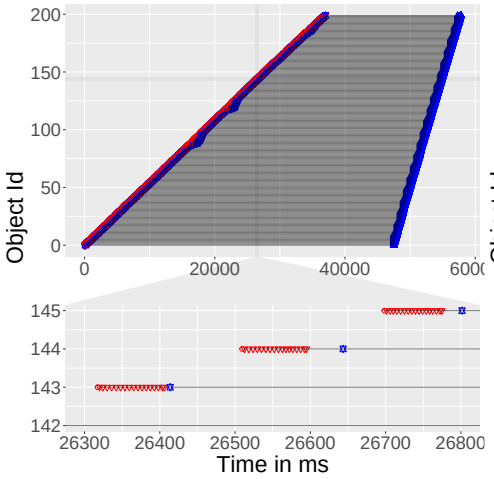
tier. Then, on each mover invocation, many objects are evacuated from the fast tier, and a new batch is loaded, even though this batch is not needed anymore. This increases the load on the storage device and locks the object for access during the migration, which delays access further. A similar problem can be seen during reading access.

The wrap size in the LRU-policy-access of 12 equals the object size set as the limit for the fast tier. This shows that after completing the migration operations, enough new requests were generated to completely swap out the high tier again. First, it would probably be better to clear the high tier and then reevaluate before promoting objects. Then, new objects can be created at the high tier instead of the slow.

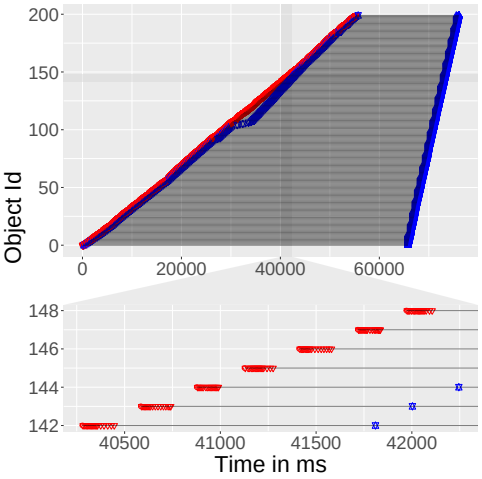
Concluding Thoughts

The implementation of the LRU-policy is simple since each access pushes a message in a `GAsyncQueue` and `process`. This information from this queue is used to populate a ring buffer. This implementation and the implementation for the `writebuffer`-policy were easily achieved in only 250 and 120 lines of code, respectively. Conveniently, objects have an ID that allows for easier and faster hint generation for the LRU policy. However, since migration is not possible with that ID, a list of all objects must be handled in each policy. The additional problem is that a migration once started waits until the object is able to migrate. This prohibits the policy from starting another migration if one object is still in use or initiating multiple migrations in parallel.

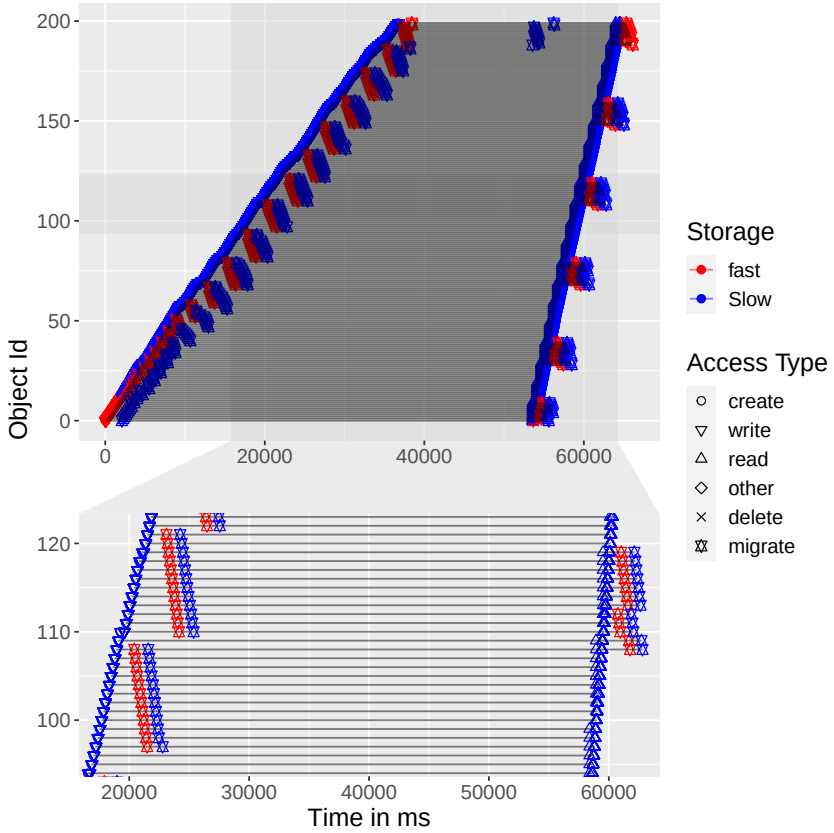
This can results in a scenarios where the HSM performance is equal to or worse than only using the slow tier. In combination with the observation that PFS can outperform local BB this leads to the question whether a HSM is useful or if the performance provided from the PFS is sufficient. The read and write bursts from `partdiff` and `pardiff_read` have shown in combination with LRU that through the higher calculation times and additional management overhead mechanism working for CPU-Caches can not be transferred straight to HSM. Also, as seen in figure 5.7b, the wrong configuration can lead to using too many resources, ultimately limiting concurrently used resources.



(a) `writebuffer` policy access with a fitting migration time. Each object is created on fast memory and moved away before the next is created.



(b) `writebuffer` policy access with a too high migration time: Objects are held on the fast tier even if they are no longer needed, which leads to unnecessarily allocated space on the fast tier.



(c) `LRU` policy access. As seen left, the policy migrates data from the slow to the fast tier (red star), but does so too late, since the burst is already over, and migrates them back to the slow tier (blue star), which results in additional I/O-Operation and therefore reduced performance.

Figure 5.7.: Object access visualization for different policies for execution of `partdiff` and `partdiff-read` sequenced: Each object is represented by a horizontal life line. Each access is denoted with a mark, which represents different access types. The mark's color denotes on which tier the object is after that operation. A special case is the migration mark, and a red migration mark stands for migration from blue to red. The total execution time was 3min.

Chapter 6.

Conclusion

This chapter starts with a summary of the thesis then suggests improving the implemented mechanism further based on insights from working with it.

6.1. Summary

A mechanism was written to integrate module-based HSM policies in JULEA for object storage to provide a fast and easy option to prototype and test these policies for research purposes. Multiple currently researched HSM policy approaches were evaluated and adopted for the interface design. General concerns that also influenced the design were listed in section 4.1.1. To manage the policy modules a new abstraction layer was introduced on top of the object backend (`JBackendStack`).

The implementation was tested on a client-server JULEA setup, with different storage setups, each containing two tiers for simplicity. In addition to the execution times of micro-benchmarks and applications, the access pattern was also analyzed for deeper insight into the object migration.

The tests show that a tailored policy can improve object storage performance within the system. However, they also show that the opposite can be the case. The policy should therefore be adapted to the use case and parameterized.

A new abstraction layer and additional data needed to be managed with the new module, which led to slower performance even if only a dummy policy was used. Also, especially for workflows with small or empty objects, the hint function can significantly impact the performance, which should also be considered when writing a policy. The interface allows for a simple implementation of these test case policies without modifying them.

In conclusion to maximize the performance gain and avoid negative impacts HSM-policies should be designed with the application in mind (Ghoshal & Ramakrishnan, 2021; Lehner, 2017).

6.2. Future Work

While working on this thesis, different areas for further improvements and extensions were identified, namely improving the interface, performance, tooling, and integration.

The interface can be further improved by merging more with the `JBackend` interface. Currently both `j_backend_stack_begin` and `j_backend_begin` being are called, and again for closing, which feels redundant and could be avoided. Furthermore, a more flexible approach for migration calls would allow more straightforward implementation and more variety for policies. For example, could migration calls take an array of commands and parallelize the migration or allow for out-of-order execution. Also, implementing non-blocking migration calls or migration calls that immediately return if the resource is busy would allow for faster and more performant implementations. A configurable execution interval for the `process` function, as described in section 4.2.3, could streamline the policy creation and testing further.

As demonstrated, the performance impact of the abstraction can be highly dependent on the use case. To better support a variety of use cases the following enhancements are suggested. Currently, each object gets a new lock. These locks could be reused and reconstructed with a mutex to allow for more control by the OS and, therefore, reduced platform dependency. Also, it may be possible to still allow read access while migrating an object to hide migration times further. The introduction of a `cache` semantic which would create a copy of an object on a faster tier and will only migrate it back if changes are made. Furthermore, if only one object backend is used, the policy mechanism should be opt-in to avoid unnecessary performance loss. Furthermore, it could be researched if the order of calling the hinting functions and execution of the object access impacts the performance.

Another issue is that currently it cannot be checked how long different parts are waiting for an event to occur. It would give a lot more insight if it were possible to see when actions are issued and when they are executed, and how long they queued before and why. With this information it would be possible to identify further factors for performance optimization within the policy module.

Experiments with example implementations of simple policies have shown how essential tailored parameters are as they can, reducing the memory footprint as shown for the `writebuffer`-policy. It would be further possible for the `writebuffer`-policy to deduce that parameter automatically, based on the duration between object creation and last write-access. Semi and full automatic policy parameter deduction can improve the efficiency of policies and should be further pursued.

The last area for further improvements is the integration into JULEA. Currently, it is only possible to use the policy at a JULEA server. However, the framework also supports accessing the backend directly without a server and network in between. Supporting direct access would also give better insights into performances of different HSMs, since it would eliminate network latency and limitations.

Bibliography

Axboe, J. (2022, March 7). *Axboe/fio*. Retrieved March 7, 2022, from https://github.com/axboe/fio. (Cit. on p. 39)

Borba, E., Tavares, E., Maciel, P., Lira, V., & Araújo, C. G. (2020). Performance and Energy Consumption Evaluation of Hybrid Storage Systems. *2020 IEEE International Systems Conference (SysCon)*, 1–6. https://doi.org/10.1109/SysCon47679.2020.9275913 (cit. on p. 7)

Cache replacement policies. (2022, February 22). In *Wikipedia*. Retrieved March 7, 2022, from https://en.wikipedia.org/w/index.php?title=Cache_replacement_policies&oldid=1073465080. (Cit. on p. 55)
Page Version ID: 1073465080

Carzaniga, A., Mattavelli, A., & Pezzè, M. (2015). Measuring Software Redundancy. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, *1*, 156–166. https://doi.org/10.1109/ICSE.2015.37 (cit. on p. 22)

Ceph File System — Ceph Documentation. (2022). Retrieved March 7, 2022, from https://docs.ceph.com/en/pacific/cephfs/index.html. (Cit. on p. 55)

Chen, H.-Y., Lee, P.-Y., & Chang, H.-P. (2016). A Multi-Tiered Storage Structure for Cloud Computing. *2016 International Computer Symposium (ICS)*, 636–639. https://doi.org/10.1109/ICS.2016.0130 (cit. on p. 22)

Devarajan, H., Kougkas, A., & Sun, X.-H. (2020). HFetch: Hierarchical Data Prefetching for Scientific Workflows in Multi-Tiered Storage Environments. *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 62–72. https://doi.org/10.1109/IPDPS47924.2020.00017 (cit. on pp. 13, 17, 18)

Dong, B., Byna, S., Wu, K., Prabhat, Johansen, H., Johnson, J. N., & Keen, N. (2016). Data Elevator: Low-Contention Data Movement in Hierarchical Storage System. *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, 152–161. https://doi.org/10.1109/HiPC.2016.026 (cit. on pp. 8, 11, 18, 31)

Durner, D., Leis, V., & Neumann, T. (2019). On the Impact of Memory Allocation on High-Performance Query Processing. *Proceedings of the 15th International Workshop on Data Management on New Hardware*, 1–3. https://doi.org/10.1145/3329785.3329918 (cit. on p. 31)

Duwe, K., & Kuhn, M. (2021). Coupled Storage System for Efficient Management of Self-Describing Data Formats (CoSEMoS), 14. Retrieved February 15, 2022, from https://www.parcio.ovgu.de/parcio_media/Research/CoSEMoS/Deliverable_D1_Report-p-166.pdf (cit. on pp. 9, 14)

The Enzo Project. (2022). Retrieved March 2, 2022, from https://enzo-project.org/. (Cit. on p. 37)

Fuerst. (2018). *Spinlocks and Read-Write Locks*. Retrieved February 22, 2022, from https://www.cs.utexas.edu/~pingali/CS378/2015sp/lectures/Spinlocks%20and%20Read-Write%20Locks.htm. (Cit. on p. 34)

Ghoshal, D., & Ramakrishnan, L. (2021). Programming Abstractions for Managing Workflows on Tiered Storage Systems. *ACM Transactions on Storage*, *17*(4), 29:1–29:21. https://doi.org/10.1145/3457119 (cit. on pp. 17, 19, 20, 22, 33, 49)

GLib-2.0. (2022). Retrieved February 23, 2022, from https://docs.gtk.org/glib/. (Cit. on p. 21)
GLib.KeyFile.get_string_list. (2022). Retrieved February 23, 2022, from https://docs.gtk.org/glib/method.KeyFile.get_string_list.html. (Cit. on p. 29)
GLib.RWLock. (2022). Retrieved February 15, 2022, from https://docs.gtk.org/glib/struct.RWLock.html. (Cit. on p. 34)
Guerra, J., Pucha, H., Glider, J., Belluomini, W., & Rangaswami, R. (2012). Cost Effective Storage using Extent Based Dynamic Tiering, 14 (cit. on p. 7).
Hariri, R. H., Fredericks, E. M., & Bowers, K. M. (2019). Uncertainty in big data analytics: Survey, opportunities, and challenges. *Journal of Big Data*, *6*(1), 44. https://doi.org/10.1186/s40537-019-0206-3 (cit. on p. 7)
How L1 and L2 CPU Caches Work, and Why They're an Essential Part of Modern Chips - ExtremeTech. (2022). Retrieved February 22, 2022, from https://www.extremetech.com/extreme/188776-how-l1-and-l2-cpu-caches-work-and-why-theyre-an-essential-part-of-modern-chips. (Cit. on pp. 8, 12)
Iliadis, I., Jelitto, J., Kim, Y., Sarafijanovic, S., & Venkatesan, V. (2015). ExaPlan: Queueing-Based Data Placement and Provisioning for Large Tiered Storage Systems. *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 218–227. https://doi.org/10.1109/MASCOTS.2015.41 (cit. on pp. 9, 13)
Iperf3: A TCP, UDP, and SCTP network bandwidth measurement tool. (2022, March 10). Retrieved March 11, 2022, from https://github.com/esnet/iperf. (Cit. on p. 40)
Jiang, T., Zhang, G., Huang, Z., Ma, X., Li, Z., & Zheng, W. (2021). FusionRAID: Achieving Consistent Low Latency for Commodity SSD Arrays, 17 (cit. on p. 7).
Kang, S., Park, S., Jung, H., Shim, H., & Cha, J. (2009). Performance Trade-Offs in Using NVRAM Write Buffer for Flash Memory-Based Storage Devices. *IEEE Transactions on Computers*, *58*(6), 744–758. https://doi.org/10.1109/TC.2008.224 (cit. on p. 12)
Katal, A., Dahiya, S., & Choudhury, T. (2022). Energy efficiency in cloud computing data center: A survey on hardware technologies. *Cluster Computing*, *25*(1), 675–705. https://doi.org/10.1007/s10586-021-03431-z (cit. on p. 7)
Khlebnikov, K. (2022, March 8). *Ioping*. Retrieved March 11, 2022, from https://github.com/koct9i/ioping. (Cit. on p. 39)
Koltsidas, I., Sarafijanovic, S., Petermann, M., Haustein, N., Seipp, H., Haas, R., Jelitto, J., Weigold, T., Childers, E., Pease, D., & Eleftheriou, E. (2015). Seamlessly integrating disk and tape in a multi-tiered distributed file system. *2015 IEEE 31st International Conference on Data Engineering*, 1328–1339. https://doi.org/10.1109/ICDE.2015.7113380 (cit. on p. 12)
Koo, D., Kim, J.-S., Hwang, S., Eom, H., & Lee, J. (2017). Adaptive hybrid storage systems leveraging SSDs and HDDs in HPC cloud environments. *Cluster Computing*, *20*(3), 2119–2131. https://doi.org/10.1007/s10586-017-1002-5 (cit. on pp. 12, 18, 19)
Krish, K. R., Wadhwa, B., Iqbal, M. S., Rafique, M. M., & Butt, A. R. (2016). On Efficient Hierarchical Storage for Big Data Processing. *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 403–408. https://doi.org/10.1109/CCGrid.2016.61 (cit. on p. 7)
Kuhn, M. (2017). JULEA: A Flexible Storage Framework for HPC. In J. M. Kunkel, R. Yokota, M. Taufer, & J. Shalf (Eds.), *High Performance Computing* (pp. 712–723). Springer International Publishing. https://doi.org/10.1007/978-3-319-67630-2_51. (Cit. on p. 8)

Lehner, W. (2017). The data center under your desk: How disruptive is modern hardware for DB system design? *Proceedings of the VLDB Endowment*, *10*(12), 2018–2019. https://doi.org/10.14778/3137765.3137834 (cit. on pp. 12, 28, 49)

Meng, X., Wu, C., Li, J., Liang, X., Bin, Y., Guo, M., & Zheng, L. (2014). HFA: A Hint Frequency-based approach to enhance the I/O performance of multi-level cache storage systems. *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 376–383. https://doi.org/10.1109/PADSW.2014.7097831 (cit. on pp. 7, 14)

Oukid, I., & Lersch, L. (2018). On the Diversity of Memory and Storage Technologies. *Datenbank-Spektrum*, *18*(2), 121–127. https://doi.org/10.1007/s13222-018-0287-8 (cit. on pp. 7, 8, 12)

Perarnau, S., Zounmevo, J. A., Gerofi, B., Iskra, K., & Beckman, P. (2016). Exploring Data Migration for Future Deep-Memory Many-Core Systems. *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, 289–297. https://doi.org/10.1109/CLUSTER.2016.42 (cit. on pp. 11–13, 17)

Progressive File Layouts - Lustre Wiki. (2022). Retrieved March 7, 2022, from https://wiki.lustre.org/Progressive_File_Layouts. (Cit. on p. 55)

Rudoff, A. M. (2019, August 17). *Developers Embrace Intel® Optane™ DC Persistent Memory for Web-Scale Data-Centric Solutions*. Intel. Retrieved February 22, 2022, from https://www.intel.com/content/www/us/en/developer/articles/case-study/developers-embrace-intel-optane-dc-persistent-memory-for-web-scale-data-centric-solutions.html. (Cit. on p. 8)

Schmidt, D. C. (1995). An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events, 11 (cit. on p. 11).

Schoenmakers, T. (2021, October 28). *Slowpokefs*. Retrieved March 7, 2022, from https://github.com/schoentoon/slowpokefs. (Cit. on pp. 38, 55)

Secretary, I. C. (1999, December). *Information technology — Programming languages — C* (ISO/IEC 9899:1999). International Organization for Standardization. https://www.iso.org/standard/29237.html. (Cit. on p. 31)

Wong, H. (2013). Intel Ivy Bridge Cache Replacement Policy. Retrieved February 22, 2022, from https://blog.stuffedcow.net/2013/01/ivb-cache-replacement/ (cit. on pp. 8, 12)

Zhang, W., Tang, H., Zou, X., Harenberg, S., Liu, Q., Klasky, S., & Samatova, N. F. (2015). Exploring Memory Hierarchy to Improve Scientific Data Read Performance. *2015 IEEE International Conference on Cluster Computing*, 66–69. https://doi.org/10.1109/CLUSTER.2015.18 (cit. on pp. 8, 9, 13)

Glossary

CephFS ”CephFS, is a POSIX-compliant file system built on top of Ceph’s distributed object store, RADOS”(“Ceph File System — Ceph Documentation”, 2022). 44

LRU Least recently used is a cache policy which ”discards the least recently used items first”(“Cache Replacement Policies”, 2022). 9, 40, 41, 43, 44, 46

PFL ”Progressive file layouts are characterized by increasing the stripe count of the file in a step-wise manner as the file offset increases”(“Progressive File Layouts - Lustre Wiki”, 2022). 19

SlowPokeFS ”Simulates slow disk IO using FUSE. Generally useful for testing”(Schoenmakers, 2013, April 8/2021). 37–39, 43–45

Statement of Authorship

I herewith assure that I wrote the present thesis independently, that the thesis has not been partially or fully submitted as graded academic work and that I have used no other means than the ones indicated. I have indicated all parts of the work in which sources are used according to their wording or to their meaning.

I am aware of the fact that violations of copyright can lead to injunctive relief and claims for damages of the author as well as a penalty by the law enforcement agency.

Magdeburg, March 27, 2022

_____

Signature