Master Thesis

Data Migration Policies in a Copy-on-Write Tiered Storage Stack - Conception and Implementation

Johannes Wünsche
thesis@spacesnek.rocks

October 17, 2022

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

Second Reviewer:
Dr. David Broneske

Supervisor:
Jun.-Prof. Dr. Michael Kuhn

Abstract

We currently observe a trend of ever-increasing data demand in a range of computation domains, these include high-performance computing, databases, and many more. Satisfying these growing storage requirements, the continuous improvement and scaling-out of existing technology have been furthered immensely in the last decade, as well as the introduction of new mediums and interfaces. Varying advantages have emerged for these mediums. To utilize them, different storage mediums are combined in heterogeneous storage systems, which aim to optimize the usage of storage mediums to prevalent data flows, such as burst-write patterns. However, the organization of these systems is an NP-hard problem, which is, furthermore, extraordinarily hard to approximate. Hence, so-called "migration" policies, act on proxy values such as data temperature, access frequency, or access recency to gain improvements over the status quo. Additionally, with concepts like Copy on Write and write-optimized data structures becoming more common, held assumptions about data distribution and latency have changed. We explore policies and issues of data migration in a hierarchical heterogeneous storage stack based on write-optimized B^{ϵ}-trees by implementing migration functionality and a dynamic policy interface with two proven policies from other papers as example implementations. We find that strong differences in placement exist, based on data size and access type, contradicting usual assumptions made on non-write-optimized storage. Furthermore, when observing I/O time, we see a 30% speedup in common write patterns, with remaining potential.

Contents

Chapter 1.

Introduction

In this chapter we will shortly go into the motivation of this thesis as well as introducing a few key concerns which we want to address with the content of this thesis. Furthermore we give a rough historical context of the underlying development of storage technology, future trends and their impact on present problems.

1.1. The Need for Hierarchical Storage

In the last ten years great progress has been made in the improvement of storage technology and the introduction of new mediums into systems. For example, the Non-volatile Memory Express (NVMe) interface has in combination with Solid State Disk (SSD) storage brought an up to eight times greater performance in databases as shown by Xu et al. (2015).

While modern storage technologies with low latency and high throughput like NVMe-SSDs offer great non-volatile access latency and are widely available from many manufactures, the cost per GB[1] makes this technology prohibitively expensive for large scale storage. For example, the new cluster of the German Climate Research Center (DKRZ) called *Levante* uses two Lustre[2] filesystems with a total capacity of 130 PB[3]. Using only NVMe-SSDs as backing storage the cost for the storage devices alone would be approximately 26 million dollars, excluding the costs for additional connectors or anything else needed to actually use them. Considering the funding of 45 million dollars[4] which the DKRZ received for this project, this would have already required more than half of the available budget. Since storing all data on NVMe-SSDs seems unfeasible to finance, a hybrid approach of multiple technologies can offer a combination of low latency and economic viability.

While this is a singular example, we can easily deduct that other data centers will encounter similar problems when estimating the feasibility of a system configuration. Not only does this problem limit feasibility, it is also exacerbated by another problem which is the continuous rate of improvement of computational power compared to storage capacity and throughput.

[1] Lüttgau et al. (2018) denotes the cost at around 20 dollar cents per GB.
[2] A high-performance distributed parallel filesystem. `https://www.lustre.org/`
[3] `https://docs.dkrz.de/doc/levante/file-systems.html`
[4] `https://www.dkrz.de/en/communication/pressemitteilungen/new-high-performance-supercomputer-starts-its-operation-at-dkrz`

1.2. The Overproportional Relation of Computational Power to Storage Trends

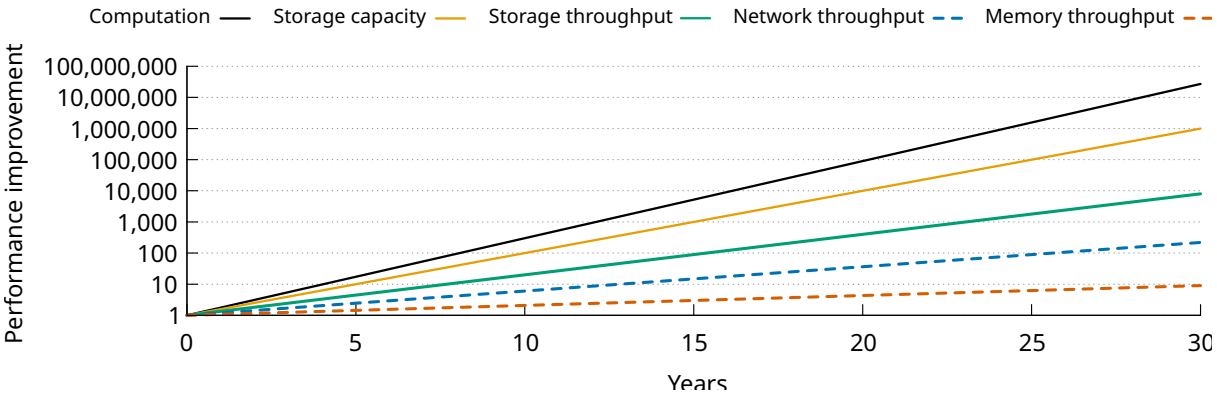


Figure 1.1.: Trends of capacity and power in HPC. Figure from lecture by Kuhn (2021).

As a rough orientation, we present in Figure 1.1 a plot portraying a trend line for important characteristics in computation systems. Interesting for us are three trend lines from this figure. First, the computation capabilities, which are majorly determining how much additional data a Central Processing Unit (CPU) may produce in a given time. Second, the storage capacity, as this denotes how much more data we will be able to store while retaining a similar amount of disks and storage nodes in a system. Third, the storage throughput, as with an increased data storage capacity the rate of data throughput to these devices heavily influences how fast we can fill this capacity.

The graph shows an exponential increase of all three although at different rates. Taking the most recent discernible point at 5 years, we see a projected increase in computation of 20x the current level, compared to 10x of the current capacity and only 5x times the current throughput. Though performance is unlikely to actually scale in this strong manner as with plateauing improvements in single core performance focus lies on multi core parallel workflows. The same can be said about storage with iterative improvement of existing technologies we likely see only a part of the projected speedup. Seeing these numbers as rough estimations, they reveal a strong mismatch between throughput and computation and a weaker mismatch between computation and capacity. Meaning we are likely to increase the number of disks and storage nodes in the future, but definitely need to optimize their usage to keep up with the rate of produced data. Therefore, the I/O bottleneck will be a stronger limitation if adjustments in architecture concepts are not considered. But we need not necessarily only consider future trends, Rybintsev (2020) notes that on the example of geographic analysis (a group of data intensive scientific applications), I/O is already the most limiting factor in modern High Performance Computing (HPC) systems. Similar observations in a general HPC setting have also been acknowledged by Thomas et al. (2021).

Due to the severity, problem solutions are already being conceptualized and designs tested. For example, new clusters like the new super computer of the Oak Ridge Leadership Computing Facility (OLCF) called *Frontier* employs burst buffers (storage of limited size with high throughput) with multiple storage tiers to deal with this bottleneck.[5] The burst buffers of *Frontier* are not managed, users have to migrate data manually between the local burst buffers and

[5] `https://docs.olcf.ornl.gov/systems/frontier_user_guide.html`

the global distributed file system. Attempts to manage these already exist with Yildiz et al. (2018) creating a burst buffer manager which improves both write performance as well as read performance via data pre-fetching. Similar focus in a two-tier storage system was made by Thomas et al. (2021), they propose a classification of file lifetimes based on neural networks to place files either in high-performance storage or long-term storage.

1.3. Summary

We see that there are multiple factors increasing the demand for an advanced storage model to be created, ranging from limitations in costs and speed, to future trends in computing and the conception of larger, more powerful systems.

This requires an extensive research process to determine effective measures to deal with this increase in system complexity. Luckily, with the increase in computation power we receive some resources to actually dedicate to storage optimization. Additionally, modern storage technologies are expensive and/or energy intensive. With the goal in mind to reduce energy and monetary costs we have to rely on a mixed storage technology stack to optimize the use of each kind. Lastly, the introduction of new storage types like Non-volatile Random Access Memory (NVRAM)[6] with unique characteristics offer new advantages which require experimentation and conception of novel data management to fully utilize them.

While some approaches already exist, the I/O bottleneck and optimal storage utilization is still a major limitation in modern system especially in the domain of HPC.

1.4. Goals of this Thesis

In this thesis, we aim to explore the combination of write-optimized hierarchical Copy on Write storage for the prospect of automated data migration to approximate the problem of data placement. Specifically, we want to know:

1. Does storage based on write-optimized data structures, provide any advantage, implication or adverse effects on migration policies?

2. Can we incorporate the semantics of Copy on Write (CoW) in the migration process?And do we gain advantages from it?

3. And lastly, how is the specific architecture of a multi-tier B^{ε}-tree effecting the migration design?

[6] Even if the first large scale production has been discontinued by Intel with the stop in production of the Optane series. `https://www.tomshardware.com/news/intel-kills-optane-memory-business-for-good`

1.5. Structure of this Thesis

We begin this thesis with an overview and introduction of related domains in Chapter 2. Specifically, we explain the underlying data structure of the B^{ε}-tree, hierarchical storage, the used storage stack called *Haura*, the problems of data placement and migration, as well as cache and hierarchical storage management methods. Then, we give a brief overview, in Chapter 3, of specific papers which are related to the topic of this thesis and which influenced the design process. Next, we develop, in Chapter 4, a concept to integrate data migration and migration policies in *Haura*. There we also describe challenges, limitations, and additions we have made in the course of this thesis to enable the integration of a migration model. Following, we describe in Chapter 5, miscellaneous modifications we have made in *Haura*, which are only tangentially related to the topic of this thesis. Afterwards, in Chapter 6, we design and perform an evaluation of the implement migration policies compared to each other and the default behavior. We formulate our findings and summarize them at the end of the chapter. Lastly, we summarize the thesis, concept and findings briefly in Chapter 7. Additionally, we list potential topics for future work.

Chapter 2.

Background

In this chapter we explain and explore topics which serve as the base of this thesis or are helpful in understanding the context and implications of concepts presented and developed. We start by explaining the B$^\varepsilon$-tree (Section 2.1) the underlying data structure, which is a write-optimized B-tree. Next we give an overview of hierarchical storage systems (Section 2.2), why they exists and where they are used. Following, we talk about *Haura* (Section 2.4) a hierarchical storage engine using B$^\varepsilon$-trees to expose a write-optimized key-value and object store. Later, we explain the concept of Copy on Write (CoW) (Section 2.5) a common safety technique in modern filesystems. Afterwards, we describe the problem of data placement (Section 2.6) and create a theoretical base which we use to understand challenges in data migration decisions. Lastly, we explore common cache (Section 2.7) and hierarchical storage management methods (Section 2.8) and categorize them based on requirements of our thesis.

2.1. B$^\varepsilon$-Tree

The B$^\varepsilon$-tree was originally conceptualized by Brodal and Fagerberg (2003) as a measure to create a trade-off between high insertion costs of (a,b) trees (most commonly B-tree) and truncated buffer trees.[1]

The base idea of the B$^\varepsilon$-tree is to reduce the number of small leaf writes which are required in a normal B-tree upon insertion. The problem in B-trees, as shown by Bender et al. (2015), is that with many small write operations performance begins to suffer as the complete leaf has to be written on each update. If we optimize a B-tree for small write operations (by decreasing the node size to below 4 MB) problems arise as read operations which normally use high-bandwidth large sequential read operation (of over 4 MB) cannot be used anymore as node boundaries would be crossed.

Instead the B$^\varepsilon$-tree uses buffers of size $B - B^\varepsilon$ with $0 \leq \varepsilon < 1$ in each internal node as shown in Figure 2.1, which allows us to use large node sizes without writing each large leaf on every insertion, therefore avoiding the diminishment of insertion performance.

[1] B-trees which are buffering a version of each leaf in all parents, which offer great insertion cost but much worse query performance

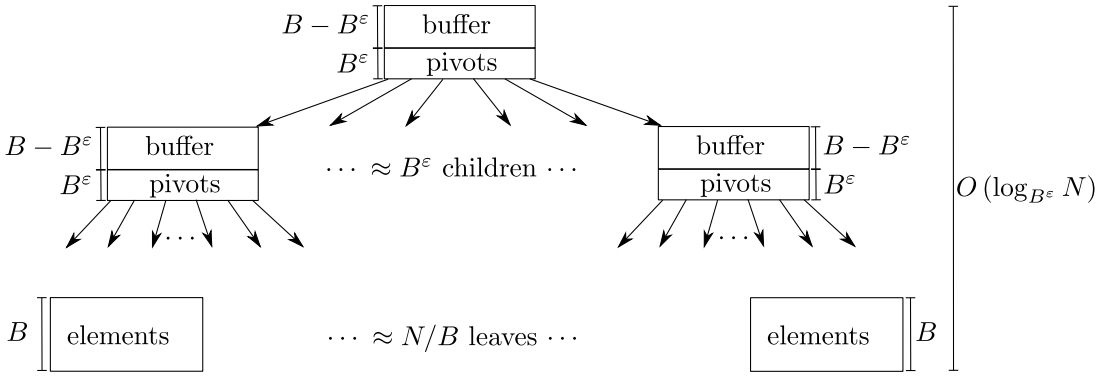


Figure 2.1.: The structure of a B$^\varepsilon$-tree. Graphic originally by Bender et al. (2015).

2.1.1. Operations

We will shortly explain the base operations and then talk about the complexity of this data structure.

Insertion

Compared to a normal B-tree insertion where we access or create a leaf directly on insertion, new data will only be inserted into the root of the tree initially in the case of the B$^\varepsilon$-tree. The tree root stores, just like all other internal nodes, updates to each key in an internal buffer until this buffer is filled. Once the buffer has been filled, the node tries to flush messages downwards, for example, to the child node with the most updates stored for. This "batching" of updates allows for a greater leaf size while retaining performance compared to flushing updates individually (Bender et al., 2015).

Deletion

A deletion works quite similar but instead of an updated value a deletion marker is inserted. Importantly, the stored value is not necessarily removed instantaneously from disk as only when the marker reaches the target leaf it may perform the removal (Bender et al., 2015).

Querying

Storing data in this manner naturally impacts the querying approach as data does not necessarily have to be stored in a leaf but may be distributed over all levels of the tree. Therefore a query reads, additionally to the leaf, all buffered data for the requested key and constructs out of them the complete result. It should be mentioned, that the order of application is vital, with data from higher levels overwriting data from lower levels since due to the insertion construction the upper data is newer and represents modifications to the already stored lower data (Bender et al., 2015).

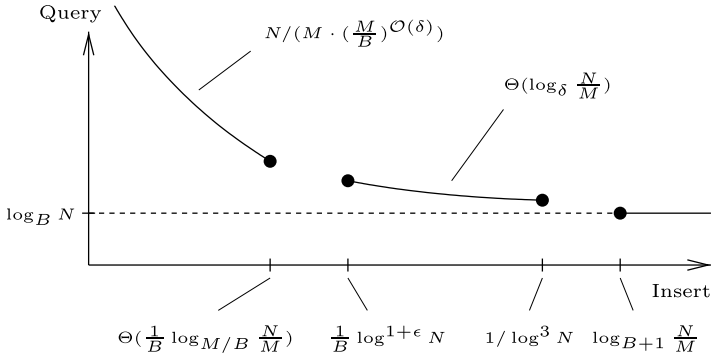


Figure 2.2.: Cost Space (in I/O operations) of (left to right) truncated buffer trees, B$^\varepsilon$ trees, and B trees. Graphic originally by Brodal and Fagerberg (2003).

Data Structure	Insert	Point Query	Range Query
B$^\varepsilon$-tree	$\frac{\log_B N}{\varepsilon B^{1-\varepsilon}}$	$\frac{\log_B N}{\varepsilon}$	$\frac{\log_B N}{\varepsilon} + \frac{k}{B}$
B$^\varepsilon$-tree ($\varepsilon = 0.5$)	$\frac{\log_B N}{\sqrt{B}}$	$\log_B N$	$\log_B N + \frac{k}{B}$
B-tree	$\log_B N$	$\log_B N$	$\log_B N + \frac{k}{B}$

Table 2.1.: Asymptotic I/O cost of relevant trees. Selection from Bender et al. (2015)

2.1.2. Costs

Brodal and Fagerberg (2003) have visualized the costs in number of required I/O operations in Figure 2.2, where truncated buffer trees occupy the left-most line followed by B$^\varepsilon$-trees and B-trees. We can see that generally the query costs of the B$^\varepsilon$-tree is not substantially higher than the B-tree[2] but insertion costs are greatly reduced.

Bender et al. (2015) provided a good overview of asymptotic I/O costs for the B$^\varepsilon$-tree with any ε and $\varepsilon = 0.5$, which we have provided in Table 2.1. In Table 2.1, we observe that, when choosing $\varepsilon = 0.5$, we do not lose much in performance on insertion but with the improved buffer write size, the costs for insertions as shown in Figure 2.2 can be expected to decrease. The remaining operations incur the same I/O costs on B-trees and B$^\varepsilon$-trees with $\varepsilon = 0.5$.

2.2. Hierarchical Storage

The design of tiered storage is traditionally a trade-off between speed and capacity. As already argued in Section 1.1 using exclusively NVMe-SSD storage is prohibitively expensive for most applications. As a result, we need to incorporate different storage technologies fulfilling specific functions, be it mass storage, low latency access, or high bandwidth throughput. Although to keep performance we need to ensure near optimal distribution of data, such that performance critical input is available from the fastest technologies and long-term storage on cheaper alternatives such as Hard Disk Drives (HDD).

[2]Which Brodal and Fagerberg (2003) actually also shows that this is a tight bound so independent of the chosen ε.

Technology	Latency	Throughput (Read/Write)	Cost ($ / GB)	Power (W / GB)
DRAM	~80ns	17/17 GB/s	5.0	0.150
NVRAM	~5μs	2.5/2.5 GB/s	1.20	0.03
SSD(NVMe)	~20μs	8.0/5.0 GB/s	0.2	0.0007
SSD	~100μs	2.1/2.0 GB/s	0.1	0.0018
HDD	~10ms	250/240 MB/s	0.03	0.0004
Tape	>20s	315/315 MB/s	0.001	-

Table 2.2.: Comparison of common storage devices. Selection from Lüttgau et al. (2018)

In the context of storage development it is important to note that new technologies do not necessarily displace older ones. For example, before the now common HDD and SSD hybrid storage was common, a combination of HDD and tape storage was used with similar arguments made about access latency (Yang et al., 2017). However, since then, HDDs have, for a large part, taken over as the mass storage device for inactive data, but tape is still widely used due to lower costs for the archival of data and the aforementioned stark expected increase in produced data (Davis, Michael C. et al., 2019). As a recent example, Davis, Michael C. et al. (2019) deployed a large scale disk and tape hierarchical storage fit for storage demands succeeding the exabyte mark.

Also, even within new concepts of storage mediums new generations are developed increasing bandwidth[3], increasing capacity[4] or accessing physical interfaces with limited availability. For example, data centers fully relying on SSDs still benefit from a tiered storage approach for different classes of SSDs as can be seen in Yang et al. (2017).

An overview of common storage technologies with their associated costs, throughput and latency is available in Table 2.2. We show in Table 2.2, the most important aspects of each technology for our context. Starting from the latency, depending on the chosen medium major differences are present ranging from 80ns for Dynamic Random Access Memory (DRAM) to 20s for tape storage. The latency is of interest to us as we have to consider fixed access costs when estimating performance, for example when deploying a system based on NVMe-SSDs it might throughput, costs and power-wise outperform a storage system build upon NVRAM but if our system performs mainly byte-wise updates on random locations in storage, the system consisting of NVRAM might outperform the other due to the significantly lower access latency. Observing the cost and power requirements, we have already made the case why monetary costs have a large influence on modern large scale systems, a similar case can be made for power requirements. First of all we can include power to a certain degree in the running costs of the system[5], furthermore with increased electricity consumption more powerful power infrastructure is required at singular nodes as well as in complete buildings for large data centers increasing the initial costs even further.

As a general overview of all these named approaches we have portrayed the most common and researched case of hierarchical storage in Figure 2.3, where a small fast storage is at the top of the hierarchy followed by multiple layers of increasing capacity with decreasing latency or throughput.

[3]The comparison can be made between SLC, MLC, TLC and QLC SSDs
[4]High-capacity Shingled Magnetic Recording (SMR) HDDs
[5]A cluster of the DKRZ had a yearly electricity costs of about 1.85 million € in 2011. `https://www.dkrz.de/en/communication/news-archive/increase-in-energy-efficiency-at-the-dkrz`

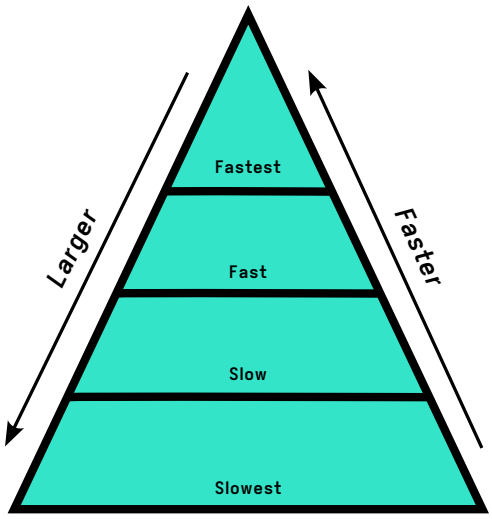


Figure 2.3.: Schema of a tiered storage architecture. Classically done, in a pyramid form with direction of increasing size and increasing speed.

Next to the term hierarchical storage there exists also tiered storage although they can be used interchangeably which we will also do in this thesis.

2.3. Hierarchical Storage Management (HSM)

The *optimized* usage of tiered storage requires more computational resources and code complexity than a brute-forced all-cache setup would require. Still, extensive research has been done to enable exactly this stack as substantial monetary savings with the shown cost-to-storage-space ratio can be made when relying on heterogeneous storage setups. Historically this has been done since at least 1985, with comparisons of multiple commercial system presented in Woodrow (1993). Compared to these implementations, the fundamental idea of hierarchical storage management has been preserved although the available space and speed has been dramatically increased. Furthermore, it is argued that in the context of HPC Hierarchical Storage Management (HSM) is one of the most important factors as I/O bottlenecks are considered to be the most prevalent source of performance loss (Lüttgau et al., 2018; Yang et al., 2017). We discuss existing methods of HSM and categorize them in Section 2.8.

2.4. Haura

Haura was initially created by Wiedemann (2018) as an experiment on how well a B^{ε}-tree is performing compared to more traditional ZFS and ext4 filesystems. The write optimization was analyzed in their thesis and noticable improvements in write performance were found to exist when compared to existing implementations, especially in small random writes but also with sequential throughput.

The storage stack has been then adapted by Höppner (2021) to support multiple storage levels to act as a hierarchical storage engine. Multiple minor additions have been done in the process of these two works and the storage stack is almost reaching a level ready to be used for further

storage experimentation. Of advantage is the construction of the storage stack as a singular codebase, aware of all levels relevant for many optimizations. This way, implementations can freely combine information from multiple levels, be it the client interface, semantic grouping of data in data sets, or the low level disk distribution.

In this section, we will describe these levels and explain shortly how they interact with the remaining stack elements.

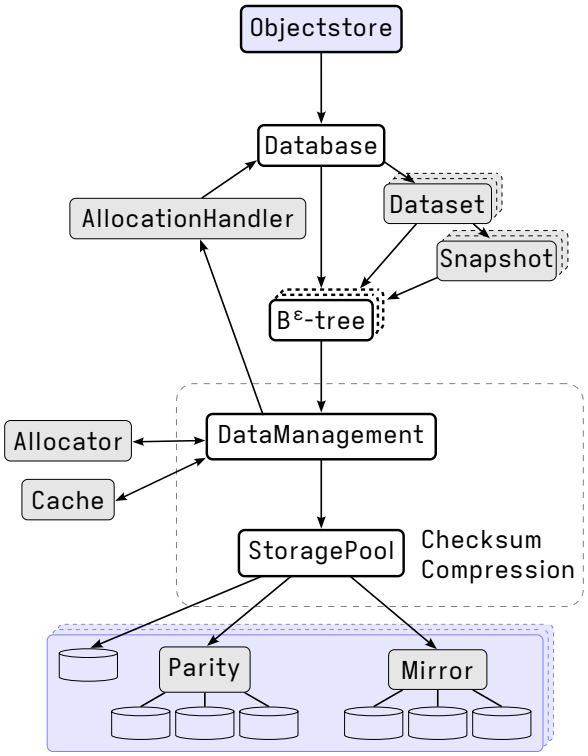


Figure 2.4.: An overview of the architecture of *Haura*

2.4.1. Database, Dataset and Object store

In Figure 2.4 we show an overview of the architecture of *Haura*. At the top of the overview we see the three most important structures of the user interface namely the *Database*, *Object Store*, and *Dataset*.

The *Database* provides the main entry point when using *Haura*. From an active *Database* we can then initiate a *Dataset* or *Object Store*. Also notable from the graphic, the *Database* creates the *AllocationHandler* which is subsequently used in the DML to organize the CoW semantics and *Segment* bitmaps. The *Database* also contains the storage configuration which is then passed to the Storage Pool Layer (SPL). The implementation of the *Object Store* is a wrapper around two *Datasets* where the keys are chunk ids and the value is the chunk content.

2.4.2. B$^\varepsilon$-tree in Haura

The *Dataset* interacts mainly with an underlying B$^\varepsilon$-tree, which receives, through its root node, messages from the *Dataset*. By default these implement *insert*, *remove* and *upsert*, although this

can be exchanged if required. Once passed, the insertion, as discussed in Section 2.1, propagates the message to the tree nodes.

Adjacent to the internals and construction of B^{ϵ}-trees are the commonalities between existing trees in an open database. Mainly non-visible to the user another tree is opened to store internal information concerning the created datasets and *Segments* information.

2.4.3. Data Management Layer (DML)

Segments are considered to be part of the DML. They are containers organizing the allocation bitmap for a range of blocks representing 1 GiB of storage. Additionally to avoid conflicts with another, all trees share the same Data Management Unit (DMU) to assure that no irregular state is reached in handling critical on-disk management, such as the allocation of blocks and updating of bitmaps.

On-disk operations and storage allocation are handled by the DML. This layer also implements the CoW semantics required for snapshots, done in delayed deallocation and accounting of a dead-list of blocks.

The *AllocationHandler* manages the *Segments* for all allocations and deallocations, adhering to the CoW semantics, and it is responsible for tracking the number of blocks distributed, which we have implemented in this thesis, see Section 4.7.

2.4.4. Storage Pool Layer (SPL)

As the abstraction over specific hardware types and raid configurations, the DMU interacts for all I/O operations with the SPL. Notable here, is the division of the layer into storage tiers of which we can use four at maximum in *Haura*. In order, they are named *Fastest*, *Fast*, *Slow*, and *Slowest*.

While named in this manner, nothing is preventing the user to initialize a storage system where the *SLOWEST* tier is fastest and the *FASTEST* tier is actually the slowest. It is by convention advised to comply to the given division, and we will assume in this thesis that this convention is followed.

The previously mentioned storage configuration can be divided into two categories, *devices* and *device groups*. A *device* can be anything from a file, block device to memory allocation (non-persistent). For files and block devices it is possible to open them with the *Direct* open flag set to bypass existing caches like the underlying file system or linux page cache.

A *device group* is any combination of devices. There are multiple supported modes including a RAID-1 like mirrored configuration, a RAID-5 like striping and parity based setup with multiple disks or a RAID-0 setup to increase throughput without safety guarantees.

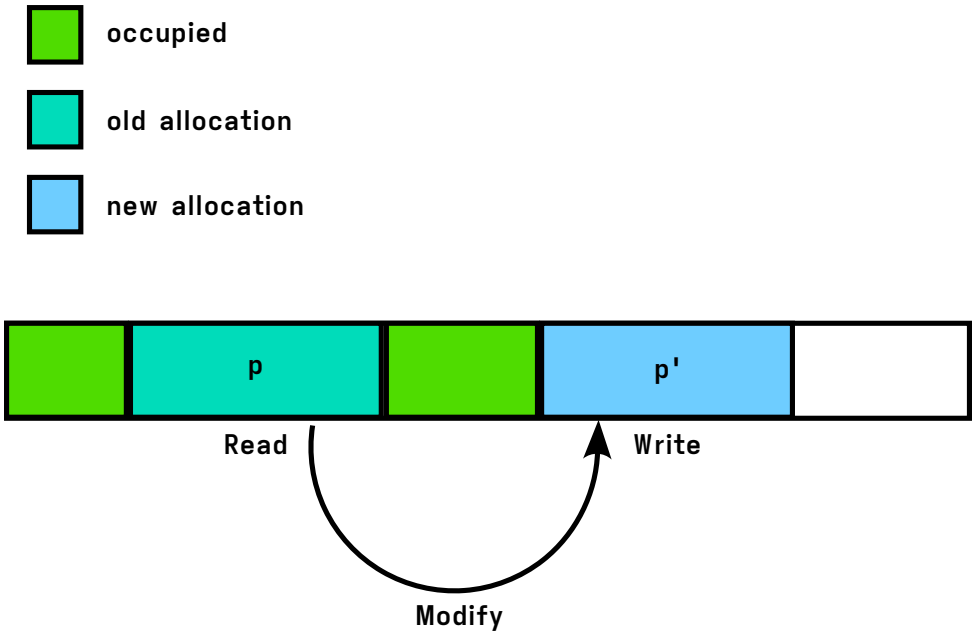


Figure 2.5.: Simplified view of the disk space and the movement of a node on storage after modification.

2.5. Copy on Write (CoW)

CoW describes a file system optimization technique employed in a number of implementations to provide read-only in-place checkpoints of previous states while also being able to concurrently provide a writable storage interface without obstructing said checkpoints. These checkpoints are also known as Point-in-Time snapshots, or short snapshots. (Peterson, 2002)

In practice, this technique can be thought of as a modification of the read-modify-write cycle. Whereas the read and modify phase of the cycle remain unchanged, first we read data from disk and then modify it as an in-memory copy. To persist the modifications then, we enter the write phase, where we find that instead of using the previously, already allocated, range on disk again, we write the modified data to a newly allocated location and discard the old one. We show this process in Figure 2.5, where some data p is read by an operation, modified to p' and subsequently written to a new location on the device. An implementation would now be able to reference the old value p which is still present at the original location. If this is not required, because no snapshot exists, we can free the occupied storage again by deallocating the original location.

Based on this example we can also observe one of the major concerns with CoW. While creating a safe workflow to preserve old data, we cause *fragmentation* of the data with continued use, especially when partially modifying files. Performance may suffer immensely on technologies like HDDs[6] when trying to sequentially read complete objects. The impact of fragmentation is lessened on SSDs, but adverse effects on performance are still not entirely circumvented as optimization when accessing close storage cells may not be utilized. When considering tape storage it is self-evident that this technology is incompatible with CoW due to high access latency.

[6]Which as we discussed in Section 2.2 still provide a valuable backbone mass storage in modern systems.

The usefulness of CoW can be seen in applications where regular short-lived backups are required, for example system stability can be enhanced when creating snapshots before modifications to kernel or system packages, as is commonly done when using btrfs[7]. Other cases can be the preservation of data in databases, to store daily backups space efficiently, in case of irrecoverable data corruption.

2.6. Data Placement and Migration

Before going into depth about specific policies managing *data placement* and *migration* we define both terms and their relation to one another.

Data Placement

The topic of *data placement* is an optimization problem which is known to be NP-hard. Furthermore, it even exists as part of the group of "extremely inapproximable optimization problems". Petrank and Rawitz (2002) show that given P $\neq$ NP it is not possible to "efficiently approximate the optimal solution even up to a very liberal approximation ratio" (Petrank and Rawitz, 2002).

Additionally, this is not only the case for the limited selection of problems which we encounter this thesis, that being the optimal *data placement* with *no* information about future accesses, but also if we could obtain this information it would not be possible, in reasonable time, to attain a nearly optimal solution. Given this we can conclude that any solution proposed in this thesis merely is a selection to organize a better-than-random distribution of data but not nearly to a degree of an optimal solution. We call this a *heuristic*.

Data Migration

Data migration is in the scope of *data placement* the movement of data from one defined position to another. Important characteristics are that a *migration* always incurs a cost on the present state of the system which occupies part of the available resources for a certain time relative to the position and properties of the element.

In the context of data movement, we tend to look at the present storage medium and size of the data transferred, though other aspects such as fragmentation are components of it.

Migration may happen in two possible direction, one is the upgrading to a higher (faster) tier, which we call *promotion*, and the other is the downgrading to a lower (slower) tier, subsequently called *demotion*.

[7] A CoW filesystem for the Linux kernel `https://btrfs.wiki.kernel.org/index.php/Main_Page`

2.7. Cache Management

The topic of data migration in HSM is closely related to cache management, also called caching policies. These have been initially developed to avoid using slow external memory in favor of volatile byte addressable DRAM or CPU caches. In general, data is replicated there and kept until a later point in time where some policy, for example Least Recently Used (LRU), decides to flush the data back onto disk to ensure persistence.

We discuss in the following section four commonly used approaches.

Least Recently Used (LRU)

One of the most effective while simple approaches is LRU, this policy simply tracks accesses and evicts if necessary the object which has not been accessed for the longest time. It has been found to be in most cases a well fitting policy and outperforms other simple policies such as First-In First-Out (FIFO). (Chrobak and Noga, 1999)

Technically LRU is an approximation of Least Frequently Used (LFU), but in most cases LRU is accurately enough to justify the simplification and therefore reduction of complexity and overhead.

The simplest implementation of LRU is a linked list where the head is the most recently used object and the tail the least recently used. On access to an object in the list the node, representing this object, is moved one position towards the head.

A slight derivation of LRU is used by the linux kernel to manage the page cache, which caches active data to avoid frequent rewrites back onto disk and is applied on block devices as well as file systems. This derivation holds an additional list which temporarily keeps already evicted pages for eventual cache misses. (Corbet, 2021)

Adaptive Replacement Cache (ARC)

A more sophisticated derivative of LRU is the Adaptive Replacement Cache (ARC) originally introduced by Megiddo and Modha (2004). The base idea is that two lists of size c are held which both are keeping recently accessed pages, in the first L_1 all pages which have been accessed once are held and in the second L_2 all pages which have been accessed at least twice are held. Therefore, we can interpret the second list as an indicator of the actual frequency of the contained pages. The cache consists of c pages in total, so that not all pages contained in the lists are actually in cache.

The number as to how many elements of L_1 or L_2 are placed into cache is determined by the number of cache misses we have based on elements which we already know. This is intended to balance between situations where a frequency based approach or a basic LRU approach works better.

Least Frequently Used (LFU)

The LFU policy can be constructed as an expansion of the LRU. To each stored element, a counter is added, which stores the number of accesses to this element. In a simple linked list implementation this can be constructed with the head as the most frequently accessed element, the tail as the least frequently accessed, and for each element we store a counter which determines on access if the element will be moved one step towards the head. We move the element if the counter of a node is after incrementing at least as large as the element in head direction.

CLOCK

CLOCK was originally developed by Corbato (1968) for Multics[8] to be used as page cache retention algorithm with a lower overhead (due to less copying or modification required) than LRU. The original definition includes a configurable k which allows the policy to be tuned to retain pages multiple times in favor of new pages, ranging from $k = 0$ representing a FIFO strategy up to $k = \infty$ expressing LRU semantics. Therefore any setting $k > 0$ and $k \neq \infty$ approximates the LRU policy.

Because of this reduction in overhead, CLOCK is the chosen algorithm for the internal cache in *Haura*.

2.8. Hierarchical Storage Management Methods

Related to the cache problem, and of interest in this thesis, is the management of hierarchical storage. Hereby, we no longer have only two layers of storage (volatile or non-volatile) of varying speeds but multiple layers n where $n \geq 2$. Most often these hierarchical storage setups are also a heterogeneous storage, consisting of multiple storage technologies such as HDD, SSD, NVMe SSD, NVRAM, or DRAM.

Important to note, the technologies and methods discussed here are intended for a vertical (single machine) storage system. Related to hierarchical storage management, is the provision of horizontal data placement where factors such as access latency become even more critical as even the highest storage latency of the above selected technologies is "only" ~10ms for HDDs. Adding several milliseconds network delay makes most accesses unbearably slow and diminishes performance especially for random access patterns. This leads to a stronger focus on geo-location and data replication in the domain of distributed storage management. An interesting problem but specifically excluded here.

2.8.1. Goals

We generally aim to improve two main characteristics of any hierarchical storage with automatic migration techniques. First, the average latency, speaking the time required starting from

[8] A mainframe operating system initially developed by MIT, General Electric, and Bell Labs. `https://web.mit.edu/multics-history/`

issuing a request to receiving some data. Second, the achieved bandwidth for streaming data from storage.

Improving the average latency is based on the access patterns to the storage, frequency of access, frequency of writes, the storage technology, the distribution of data (implying the actual location of our data), and depending on the access pattern also the fragmentation of the data.

To achieve an impact on these characteristics, we can perform migrations to move certain nodes of the tree to an available storage tier. We have, as defined in Section 2.6, no future information and are therefore restricted to estimations of future access patterns. Furthermore, we may only use a restricted amount of available computational resources as occupying most resources for expensive experimentation of state changes would lead to a net loss of performance for other tasks on the system which were meant to be optimized with the improved storage access. Also, state changes also have to consider the costs of migrating data, as valuable I/O resources are required on both memory and disk throughput of both afflicted targets.

2.8.2. Static Heuristics

As already mentioned, we require a *heuristic* to approximate a result to the problem of *data placement*. They can encompass many aspects and optimizations in behavior, although the simplest of which is a basic static heuristic. We will first talk about *Static Heuristics* and explain how they approximate a solution to the problem of data placement. Then we name their common problems and challenges as well as examples for them.

Approach

Simple static heuristic are most often fixed rule based approximation to the data placement problem. They can focus on properties of data such as recency, frequency, or lifetime of access and set based on this recorded data their appropriate storage tier. They can range from sophisticated analysis of frequency in relation to their tier and surrounding tiers to gain a holistic overview of the stack or be a simplistic approximation like lifetime-bound rules with set boundaries which lifetime belongs onto which tier, as done for example in Amazon Web Service's Tiered S3 storage[9].

A main critique of these static heuristics made by Vengerov (2008) that gains in performance are hard to verify mathematically. This is true for example when heuristic are made on "educated guesses" by an administrator or best-knowledge assumptions based on the available hardware. An indicator of performance benefit can then be made based on benchmarks either in the micro or macro scale.

Examples of static heuristics can be found in the examples of cache management, which we have presented in Section 2.7. Namely common algorithms are LRU, LFU, and CLOCK.

Each of these policies can be expanded to control the migration flow in a tiered storage stack upwards and downwards. For example the LFU policy can simply be expanded with an Most Frequently Used (MFU) interface.[10] Structurally this can be constructed as shown in Figure 2.6 where to each tier a "Decision Unit" is attached which can initiate promotion or demotion to

[9] `https://aws.amazon.com/s3/storage-classes/intelligent-tiering/`
[10] Which we will do with a chosen algorithm in this thesis.

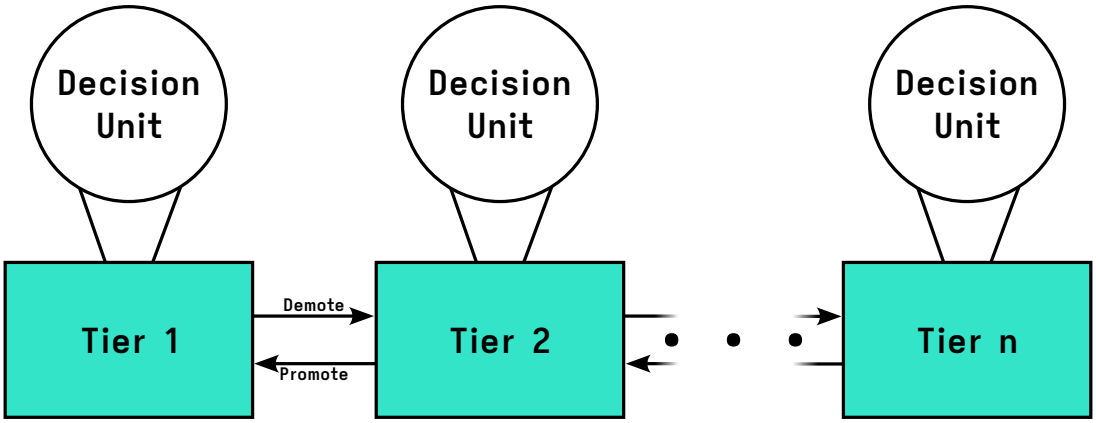


Figure 2.6.: Any policy can be expanded to function in a multi-tier storage setup by isolating multiple "Decision Units" (policy implementer) one for each tier.

the adjacent tiers. No complete view of the stack is required and the complexity remains the same. Although, while easy to construct these implementations often cause more migration load on the system than other approaches. (Zhang et al., 2022)

2.8.3. Learning Heuristics

To approach more dynamic environments, with eventually changing access patterns or unknown applications, automated adaptable solutions are under research, utilizing machine learning to solve for time dependent approximated distributions and likely future accesses.

The implementation of these approaches can vary by a large margin. As one of the simplest "self-learning" approaches we can refer back to Section 2.7 where ARC was presented, which based on cache misses, adjusted its ratio of recency to frequency based page caching. While being simple, this already adjust to a change in access patterns based on two fundamental classification of data accesses.

More complex approaches may incorporate reinforcement learning approaches like Q-Learning (Shi et al., 2020) or Temporal Difference (TD) learning (Vengerov, 2008) or even more complex setups with pre-learned neural networks (Cherubini et al., 2017; Cheng et al., 2021; Monjalet and Leibovici, 2019; Ren et al., 2019).

Although the artificial intelligence and deep learning trend of the recent decade probably already introduced the reader to these topics we will categorize the approaches to establish a common ground of understanding to allow for better matching of them to our demands.

Reinforcement Learning

Reinforcement Learning describes a class of problem solvers which act purely based on reward-driven actions on an observed system. Typically, the unit performing actions is called an agent and dynamically adjusts its output (the action to perform) based on the current input (the system state) and reward. Over time, an agent learns which actions are preferred over others by analyzing the performed actions with their returned rewards. (Kaelbling et al., 1996)

The aforementioned Q-Learning and TD learning are part of a group of reinforcement learning algorithms called *model-free* algorithms, which are algorithms which do not try to defer a model from the observed behavior but simply learn controlling them to maximize the reward. The opposite method, where at first a model is determined to which an agent can be created, is called *model-based*. (Kaelbling et al., 1996)

Temporal Difference learning or TD(λ) defines a class in which a "heuristic value" is learned instead of the reward right away. This implies the existence of some heuristic judging the state and reward output and passes on a modified value to the agent. The goal of this structure is to learn "future-proof", so to say determine how the performed action will impact not only the immediate future but which long-term implications are involved. Computation-wise it is categorized as rather expensive compared to other solutions but is shown to reach better solutions in a shorter time than other schemes. (Kaelbling et al., 1996)

Q-Learning aims to simplify the construction by incorporating the functionality of the heuristic into the agent. In contrast to a normal agent, we consider the *optimal* choice of actions after our current choice also (Q-function), if the action is still preferred we perform the action. Consequentially, we maximize this return value instead of the raw reward. It is generally consider more "effective", speak sparse resource usage, but tends to converge slower than other methods. (Kaelbling et al., 1996)

Neural Networks

Neural Networks are a class of problem solvers inspired by the basic model of neurons, which are units receiving a vector of inputs, combining them in some function to a singular value and then return the result as their output. They adjust their outputs, and learn a specific function, by tweaking the weighting assigned to individual elements of the input vector. This is usually done by some iterative error correction scheme adjusting the weights based on the perceived difference to the expected result. This process is also known as *training*. (Bishop, 1994)

Importantly, a single neuron has, due to its simple one function construction, not a large decidable solution space. If too many factors have to be considered, it will not be able to successfully identify the correct solution. Therefore, multiple neurons are connected to solve subproblems of the input and then an additional neuron is added to combine their respective outputs and use them as input to decide if the complete original problem can be classified. This results in a *network* of *neurons* with multiple layers. The training then is required to be modified as the error cannot be attributed entirely to a singular neuron, we call this modified training procedure *error backpropagation*. (Bishop, 1994)

Problematic is the resource usage with these models, the backpropagation training is rather expensive to compute especially with larger networks and this rather simple comparison to the expected or optimal result is in the problem of data placement hard to achieve, as the solution itself is hard to find out or to approximate to a reasonable degree. This makes an continuously learning approach expensive and with the restriction in system resources hard to justify, so all papers discussed, which apply neural networks, have opted to use pre-learned networks, where they use existing traces of past sessions to try to optimize future use based on extrapolated priorities of data.

2.8.4. Hinting

Last to name is the topic of *hinting* and specifically *user hinting*. Many of the mentioned models require only the basic amount of information (fetch, write, delete) about present data as these are mostly available and do not require great modification to be accumulated and analyzed. However, additional information can improve the accuracy of our assumptions by alleviating one of the fundamental problems of data placement, the prediction of future accesses.

Knowing beforehand which data will be required at which point in time, allows for a better matching of demand, although the problem in general would still be very hard to optimize.

Using a hinting interface can provide a limited and flawed image of future access patterns and is therefore a helpful optimization source which we can rely on. Although it has to be kept in mind, that an incorrect hint may also incur a performance loss as interfering optimizations may be made.

Ge et al. (2022) showed multiple low entry hints a user can use to classify files to optimize their storage location. For example they used a "streamID" to categorize files as sequential read or write and random read or write and already achieved a significantly better placement with such a simple addition. Furthermore, depending on the interface, this could also be classified automatically.

Additionally, not only user information can be valuable, Meng et al. (2014) showed that utilizing historical information of a block of data in the decision process also can significantly improve the overall performance of a storage system.

Lastly, if a defined workflow exists in which a program is run, for example with task runner, such as SLURM[11] further optimizations are feasible. Shin et al. (2019) implemented a workflow based data management tool called "Data Jockey" which provides data at high-performing tiers in HPC where firmly defined workflows are common practice. Importantly, such an approach is not required to be implemented separately but can be combined with many named in this chapter.

These papers show that hinting can offer a valuable addition to the normal decision parameters required in data placement heuristics.

2.9. Summary

The basis of this thesis is found in the fundamental problem description of *data placement*. This problem is hard to approximate with a reasonable degree of error. Early research and wide application was focused on the explicit subproblem of *cache placement*, which can be described as a two tier storage system. The movement from one such tier to another is called *data migration*.

There exist many different algorithms to decide which data should be placed where in these systems. Due to the aforementioned hardness of the problem, none of them deliver optimal results but they can substantially differentiate in the quality of their solution. We presented four common algorithms which can be used for this purpose and noted that with some modification they can be used for the larger problem of *data placement* in a multi-tier hierarchical storage.

[11] Formerly Simple Linux Utility for Resource Management (SLURM): `https://slurm.schedmd.com/`

These *hierarchical storage systems* exists as there is a fundamental trade-off we have to make when designing systems considering the cost in monetary units, energy consumption, capacity, and speed. The organization of storage in such manner, creates a dynamic system in which we potentially can optimize capacity to monetary and energy cost as well as speed to monetary and energy costs.

To leverage this potential, *hierarchical storage management* has to be used to move data between tiers in this hierarchy employing *heuristics* to approximate a solution to the fundamental problem of *data placement*. Multiple heuristics can be used, we have categorized them into the categories *static* and *learning*.

static heuristics can be extracted from the classic cache problem and can be applied to any multi-tier hierarchical storage system with some modifications, which we have shown.

learning heuristics are less common but also used in the cache problem although to a lesser degree. Also, more complex approaches are present in this class of heuristics which include *reinforcement learning* and *neural networks*.

Additionally, we have shown a write-optimized data structure called the B^{ε}-tree which has been utilized in the *Haura* storage stack. *Haura* will be the basis onto which we design our concept and implement the chosen algorithms on.

Chapter 3.

Related Work

A reinforcement learning framework for online data migration in hierarchical storage systems by Vengerov (2008)

A paper by Vengerov (2008) proposed a reinforcement learning framework and algorithm candidate based on extendability and scalability. They constructed a full-fledged algorithm and simulated, based on assumptions of access pattern histories and predictions, how well the approach treats data migrations compared to commonly used static heuristics like LRU and Size-Temperature-Replacement[1]. They found that using a reinforcement learning policy as proposed delivers a 35% improved performance relative to the compared static heuristics. Additionally, their base assumptions about the storage system structure fit nicely to our described situations and requirements.

Efficient Hierarchical Storage Management Empowered by Reinforcement Learning by Zhang et al. (2022)

A recently published paper by Zhang et al. (2022) uses the proposal of Vengerov (2008) for a hierarchical storage environment including cloud storage. Simulated results have shown a better utilization of available storage tiers when compared to heuristic approaches using file temperature indications and file size. Additionally, they found that the reinforcement learning strategy acts more stable than static heuristics, requiring only 20% of migrations compared to static heuristics.

The results of this paper reaffirm the findings by Vengerov (2008) again and indicate that the approach is viable and optimal to be applied in an actual research storage stack.

An O(1) algorithm for implementing the LFU cache eviction scheme by Matani et al. (2021)

Matani et al. (2021) conceptualize an algorithm to determine the LFU cache eviction candidate in constant time. Overall the costs are constant with $O(1)$ for insertion, querying, and deletion. They provide a data structure to implement LFU in $O(1)$ based on nested doubly linked lists, whereas the lower one acts as a bucket for all entries with the same frequency, and the upper one as the indication of frequencies. Additionally, to achieve constant time a hashmap is used as

[1] A policy trying to demote cold large files before smaller files and inversely so when promoting.

an index structure, directing access to the entries within the buckets. Considering the relatively high costs of lookup operations, the constant cost factor is relatively large but it can be assured to require a constant amount of time on each individual requests and does not require any deamortization procedure.

HintStor: A Framework to Study I/O Hints in Heterogeneous Storage by Ge et al. (2022)

A recent paper on hinting efficiency has been published by Ge et al. (2022) which focuses on creating a framework to find effective I/O hints in multi tier storage environments. In their study they modified the standard Portable Operating System Interface (POSIX) to carry some information down to the filesystem level, which allows more informed decisions on data placement and migration.

They implement a basic migration heuristics based on hints like file types, file size, metadata etc. and use a top-k selection of the most frequently used files. They also extend this heuristic with the usage of file sizes. Measurements have shown, this reduced the average latency by 50%. Valuable for our research, they found that size considerations joined with user given access pattern hints can greatly improve performance, if the access patterns are known beforehand. In examples, random read and write, as well as single stream write have been tested, each showing improvement over the status quo migration MFU implemented. Profiles suggest that this is due to the better initial placement possible due to known access patterns. (Ge et al., 2022)

Data Prefetching for Large Tiered Storage Systems by Cherubini et al. (2017)

We found in our research a number of papers successfully deploying pre-learned neural networks in multiple environments. For example, a paper by Cherubini et al. used filesystem traces recorded in a large scale scientific storage system for radio telescope data. To better predict when data is required, these traces have been analyzed to build a model which can, based on knowledge extracted from past accesses, fetch data from slow archival disk early and migrate it to more readily available fast storage. Their solution analyzed access paths and divided each segment (directories and files) to predict which directories or files are required in the near future. This makes the learned model very specialized on one specific system but may allow for better results than another more general method would have yielded. Abstraction of such an approach would yield valuable information for our case but generalization seems, due to the chosen method, unlikely to realize.

Block Popularity Prediction for Multimedia Storage Systems Using Spatial-Temporal-Sequential Neural Networks by Cheng et al. (2021)

Another paper by Cheng et al. (2021), analyzed the locality of blocks in a multimedia data stream to allow for pre-fetching of data blocks. They used a neural network which has been specifically designed for solving problems of space and time dimension called spatial-temporal-sequential

neural network. In their results they found that the overall performance gain compared to the normal fetching operations was improved by 10% to 62% (based on the variant choice) compared to common policies such as LRU. The results of this paper show the promise of further improvements when considering the practice of automated prefetching.

Summary

We have found multiple papers related to our domain of research with many tangentially touching similar topic or proposing applicable algorithms. We have not found any publication concerning themselves with multi-tier trees, as they are present in our tree structure, or with specifics about migration in CoW filesystems and possible optimizations.

Most influential to this thesis is the algorithm by Vengerov (2008) and works derived from it. Also, the surveying of influential hinting factors has influence on this thesis, with Ge et al. (2022) singled out as well applicable to our cases.

The premise of data access pattern prediction and prefetching too is closely related and is a viable enhancement of existing reacting policies as named before. Although, the application of these specific papers are limited and domain information is required beforehand. This implies that further research ,with the existing stack, is needed and extensive traces and system recordings of multiple applications have to be created before further investigating this area.

Chapter 4.

Policy Conception and Application

This chapter covers the integration of migration policies in general into the existing B^{ε}-tree storage stack, the implications and alternatives of choices made in this, and how this integration impacts other characteristics of the already implemented infrastructure and performance.

The reader should gain an overview of the structure of the problem definition of a migration policy and have insights of how additional policies may be adapted to work in the provided environment.

4.1. Scope

In this thesis, we have picked three methods of storage tier determination, which we analyze more closely and finally evaluate. In this chapter we argue for why we chose them and which steps are required for their implementation.

First, we picked heuristic based on the algorithm proposed by Matani et al. (2021), which realizes a LFU determination in constant time. Additionally, we extend the structure to incorporate a MFU interface.

Second, as this approach is rather static and some conditions, such as changing access patterns, may not be reflected optimally in its decisions, we have chosen an online learning approach for comparison. For this, we have chosen reinforcement learning to be fitting for our use-case, as it is simple enough to process next to already present resource utilization and allows us to introduce multiple factors to consider in promotion and demotion selection. Specifically, we use the approach proposed by Vengerov (2008), a file based migration policy for multi-tier storage systems, utilizing fuzzy reinforcement learning.

Another deciding factor in performance is hinting from upper layers such as the user interface (Ge et al., 2022). For this we already are at an advantage, since we know most information about the stack internally within the same project due to our monolithic base design. The application will also manage all required datasets and object stores, so that we are able to estimate which data will likely be used in the future. No additional passing of information is required at a later point, as we have also extended the user choice to include access patterns if known beforehand.

4.2. Types of Migrations

Due to the architecture of *Haura* we have devised two distinct types of migrations processes, each with their own advantages and disadvantages.

4.2.1. Key-based Migration

First, we designed the intuitive key-based migration, which is also available as a manual interface for the user. Internally the migration update overwrites whichever value of the "KeyInfo", which is the main structure representing the key metadata, was present before and forces a reevaluation of the storage preferences of all involved nodes. Importantly, the insertion logic of this update differentiates from the normal B^ϵ-tree insertion logic which would only insert at the root node, rather we perform a normal B-Tree like insertion but update all entries of internal nodes which belong to the affected key on the path with the new "KeyInfo" value.

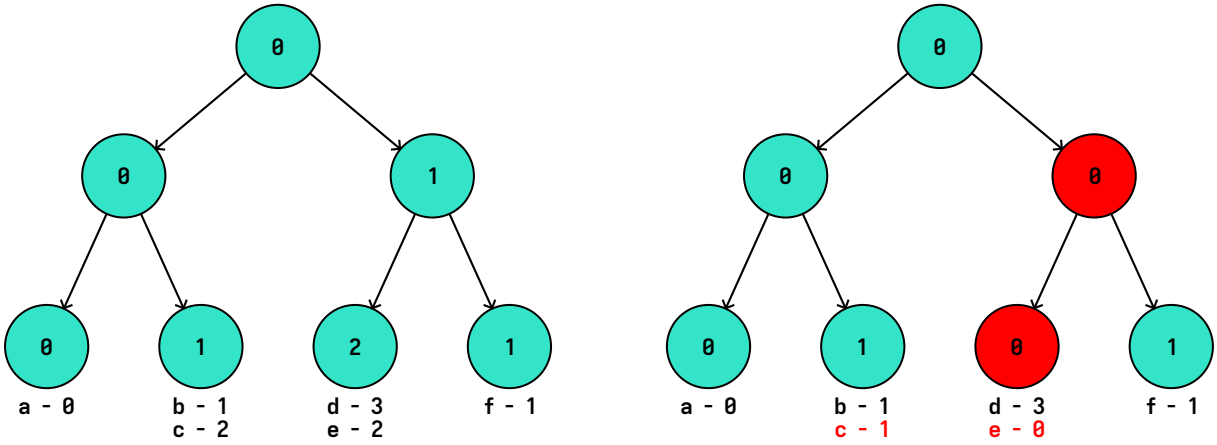


Figure 4.1.: Simplified overview showing possible key promotions in a tree. On the left to state before migration is shown, on the right the state after. Tiers are ordered with 0 being the highest and 3 being the lowest.

To visualize this process, we show in Figure 4.1 the state of the tree with changes in storage locations highlighted. Notable, is that the migration of key "c" does not affect the state of the tree except for the modification of the "KeyInfo", this is due to other keys in the leaf, which already request a higher or equal tier, therefore no migration is necessary. Of greater effect, is the migration of key "e" to tier 0. There the containing leaf and its parent are migrated to fulfill the new storage requirement. We can see from these two examples that key promotions do not violate any given constraints of the tree and the resulting state always fulfills the desired effect expressed by the promotion.

In contrast, the demotion of a key, as shown in Figure 4.2, does not always result in a desired state, as keys are grouped in leaves. On the demotion of key "c" to tier 3, nothing is changed as key "b" still requests a higher tier. When demoting key "e" to tier 3 the containing leaf is migrated, because no other key, within the same leaf, requires a higher tier at the moment.

From these two examples we can see a minor flaw in this migration approach as data which has been requested to be migrated is not migrated due to tree internals such as the key to leaf distribution. While problematic, this is not of major concern as when using objects,

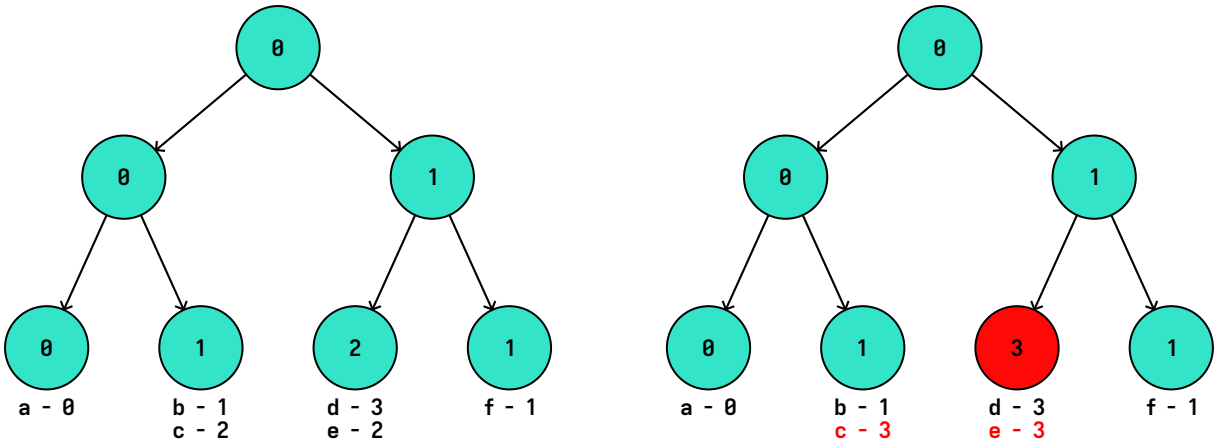


Figure 4.2.: Simplified overview showing possible key demotions in a tree. On the left to state before migration is shown, on the right the state after. Tiers are ordered with 0 being the highest and 3 being the lowest.

which contain large groups of adjacent keys, or large contents for the majority of keys as the probability that other keys on this leaf are not moved is lower, with a node size of just 4 MiB.

The provided interface requires an opened *Dataset* and known keys to which we issue a new storage preference. This preference is then considered as usual by combination with all other preferences of the containing leaf. By nature of this method, the migration is always performed eagerly, with data fetched at the time of the request, including all nodes on the path to it. In contrast, the write process back onto disk, is not performed immediately, but at any point after the call has finished. This is the result of the node insertion into the CLOCK cache (held in DRAM), which individually decides whether to keep or evict the value after it has been modified.

Additionally, performing data migrations on a *key* basis may move more data than anticipated, as all entries on the path belonging to the key value will have to be updated. For example, for an entry of size n on a path of length h containing h nodes, $h - 1$ internal nodes and 1 leaf node. Each internal node may hold up to n bytes of updates for the data entry which are not yet flushed. This requires the additional migration of these updates, to avoid updates with an old “KeyInfo” overwriting the just initiated “KeyInfo” update. This brings us to a maximum number of bytes migrated in the operation of $h \cdot n$, excluding the required bytes for the nodes themselves and any other data which they might hold. Since a leaf does not necessarily hold data of this key exclusively we must assume that other data may be moved by our operation also, increasing the upper bound of data moved to $h \cdot N$, whereas $N = 4$ MiB is the maximum size of a node in the storage stack.

4.2.2. Node-based Migration

Second, data can be moved by node affiliation also. In the context of *Haura*, node identification and access is not trivial as we explain in Section 4.5.

Again, for visualization we show in Figure 4.3 the resulting tree state after a node promotion. Visible is the migration of two nodes in each half of the tree. Most important, node migration allows us to migrate in all levels of the tree as we can see in the right half of the tree.

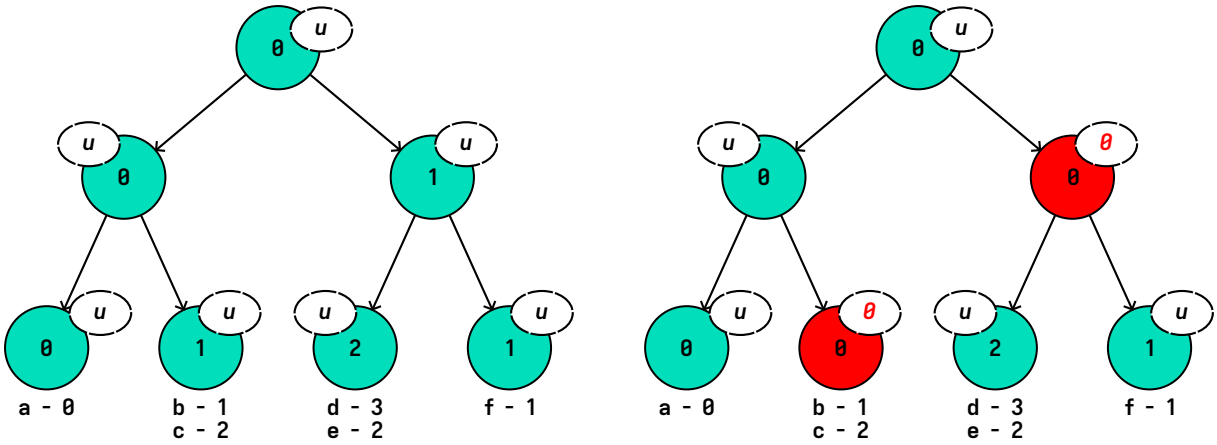


Figure 4.3.: Simplified overview showing possible node promotions in a tree. On the left to state before migration is shown, on the right the state after. Tiers are ordered with 0 being the highest and 3 being the lowest.

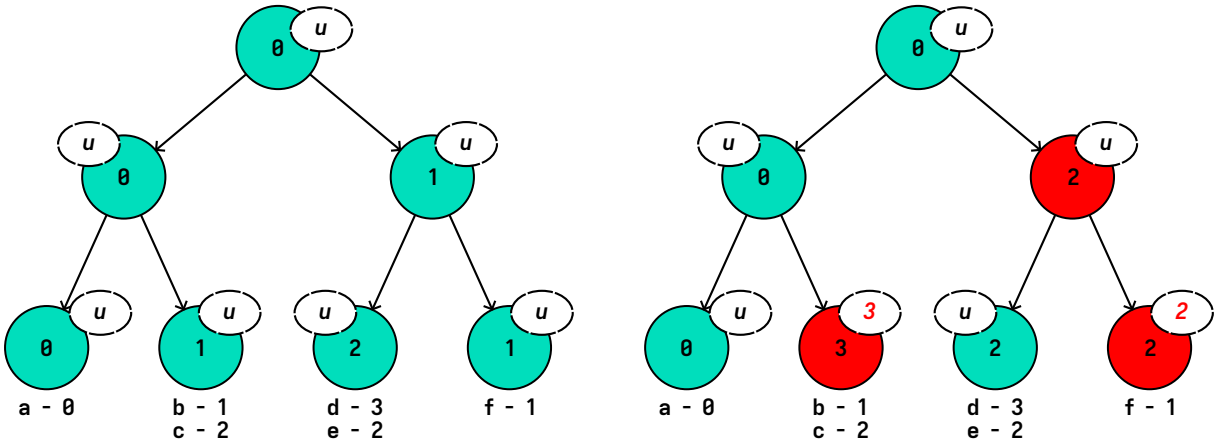


Figure 4.4.: Simplified overview showing possible node demotions in a tree. On the left to state before migration is shown, on the right the state after. Tiers are ordered with 0 being the highest and 3 being the lowest.

If we consider the node demotion, we might encounter cases such as we show in Figure 4.4. Two nodes are demoted in this constellation, the migrated node on the left simply moves the afflicted leaf to the desired location. The migrated node on the right not only changes the location of the leaf, but with the relaxed restrictions of itself and its siblings, the parent node may also be migrated. Important to note is, with this kind of migration we violate the restrictions expressed with the storage preference of keys “b”, “c”, and “f”. We talk about this problem more in Section 4.5.2.

The main limitation of this method is the passivity of the storage hints on their own, as they only move a node if the DMU accesses this node as part of any operation initiated by a present dataset. Furthermore, this implies that node based migration can only be used as a *hint* to the DMU, which we have expanded accordingly to always respect and use these hints. Importantly, this hinting mechanism yields a *lazy* “zero-cost” way to promote which is modified by the workflow of the user, due to the characteristics of CoW in the complete storage stack, as data which is modified will be written again to a different location in its entirety regardless of the presence of storage hints. Reversely, demotions are difficult to realize or when given impractical, as the probability that a node needs to be demoted is inverse to the probability

```rust
trait MigrationPolicy {
    fn update(&mut self) -> Result<()>;
    fn promote(&mut self, storage_tier: u8, desired: Block<u32>)
        ↪ -> Result<Block<u32>>;
    fn demote(&mut self, storage_tier: u8, desired: Block<u32>)
        ↪ -> Result<Block<u32>>;
    fn db(&self) -> &Database;
    fn dmu(&self) -> &Dmu;
    fn config(&self) -> &MigrationConfig;
    fn thread_loop(&mut self) -> Result<()>;
}
```

Listing 4.1: Simplified representation of the MigrationPolicy trait.

that it is accessed in the foreseeable future; invalidating their purpose and accumulating in the DMU. Therefore, the implementation of a node-only based migration structure does not seem feasible in the current state of the data structure, we discuss ways to resolve this issue in Section 4.5.

4.3. Dynamic Policies

Depending on the workflow and the storage environment, one policy might work better than another. To optimize in these situations, the design integrated in *Haura* allows for the policy to be chosen freely and even exchanged on reinitialization. Required for this is that the policy implements a *MigrationPolicy* interface, a simplified view of it is provided in Listing 4.1. In Figure 4.5, we show the overview of the *Haura* architecture updated with the "Migrator" added, which represents an instance of a migration policy.

Furthermore, for extendability each policy may have, next to the common configuration, which specifies the starting grace period, update interval, and migration threshold, a custom configuration if additional information are required. This can, for example, include a maximum number of nodes migrated on each interval or preference on file sizes for promotions.

Policies receive notifications on ongoing processes through multiple channels depending on their purpose, where, for one, the DMU produces messages (e.g. a node is written to storage). On updates, the policy then interprets these messages and modifies its own state based on messages received. This approach allows for a complete detachment of producer and consumer of messages and furthers the independence of migration policies from the remaining stack for ease of exchange.

Additionally, a policy has access to most top-level information, with access to the current *Database* instance, which due to safety concerns is behind a shared reference with the user. Although, the access cannot be guaranteed at each point in time due to the requirement of mutability of the *Database* instance. For storage information like the space statistics for individual disks or tiers, a lock-free DMU reference is available.

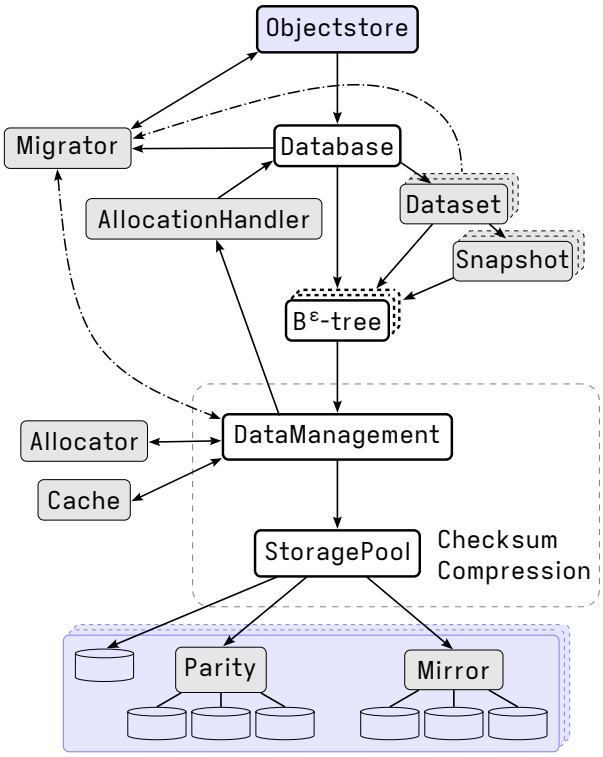


Figure 4.5.: An overview of the updated architecture of *Haura*

4.3.1. Persistence and Memory Requirements

While some additional main memory is required to be used, to track nodes and objects and decision metadata, the memory requirements should scale rather well with the present storage tier sizes and overall tree size. Observing the implemented LFU policy, we carry a "DiskOffset" (4 byte) per node, about 4 MiB, with an attached frequency (8 byte). For a storage system utilizing 100 TiB this amounts to just 300 MiB of data which will be used additionally, a moderate number which we can expect to be available on any system carrying such volume of storage space. Taking into consideration that this policy also tracks objects, which requires us to keep an identifier of the object (16 byte) with a frequency attached (8 bytes) also, the usage is not expected to grow drastically. Even if we assume again a total storage size of 100 TiB and fill this space completely with objects of size 256 KiB, the additionally required size of memory required would total about 9.3 GiB.

With the longtime collection of data, there are multiple benefits and arguments to be made to keep information from previous sessions to know which nodes in the tree might be of higher importance and might be accessed more often in the future. Although, we also may benefit from handling them individually, as patterns may change between initialization. With a blank reinitialization we avoid specializing on one specific session.

4.3.2. Restoration

The main advantage of a restorable policy would be the preservation of already learned patterns or rules from previous sessions. For example, in Section 4.8.2, the analyzed reinforcement policy would gain advantages when storing learned values into the root tree and fetch them on

restoration to avoid starting with generic starting parameters. Potentially, avoiding unnecessary migrations, reducing wear on disks and saving energy.

This can become especially advantageous, if multiple applications use the backing storage in a pipeline like modification or consumption of previously created data. When reinitialization takes place often, patterns which include sporadically opened common objects may not be found, if no information can be passed from one initialization to the next. For example, when an application directly initializes *Haura* in its own code and always writes checkpoints or other computed data in the same manner.

However, for applications which organize their data with tertiary interfaces like *JULEA*[1] by Kuhn (2017), for which an interface to *Haura* already exists, this is less problematic as they are more seldomly restarted and therefore restoration capabilities cannot offer a significant advantage.

Additionally, depending on the policy this might take up considerable storage space, which would need to be ensured to be available at the end of a session.

4.3.3. Reinitialization

One might also make the case for a blank reinitialization as access from future executions may be distinctly different to accesses from previous sessions. If we pick the pipeline example again, we create first some data, for example in a simulation, which writes, in regular intervals, the current state of the simulated system in some number of objects. We want to have, at best, always space available in the fastest tier to minimize I/O time. This becomes especially influential, when we cannot copy the simulation contents due to exceeding memory requirements or copying effort. Once written, a state will never be opened again in this session and may be demoted. Replicating this pattern in future sessions hinders optimization as we would, for access optimization, prefer to fully use the fastest tier for accessing previously written states, to perform some analysis on the just produced data.

Implementing this semantic difference is alternatively also possible with some identifier denoting which workflow or workload is expected. This could be implemented via user hinting, but requires additional information by the user or prediction capabilities via time-series data interpretation as done by Cheng et al. (2021).

Also, no further storage requirements are necessary to be defined as an “OutOfSpace” error is irrelevant to the policy state.

4.4. Messages

This section briefly explains the need for different message types and their variants. Both may be used by a policy, but none is required.

```rust
pub enum DmlMsg {
    // The three base operations of our store.
    // Largely relevant for clearing, and frequency determination
       ↪ in LRU, LFU,
    // FIFO and other policies
    Fetch(OpInfo),
    Write(OpInfo),
    Remove(OpInfo),
    Cow(OpInfo),
}
```

Listing 4.2: DML Message definition.

4.4.1. DML Message

We record two types of messages in *Haura*. The first concerns the actions of the DMU which produces, in this context, messages reporting the actions performed on a node in a tree. In Listing 4.2, we show all available variants of this message. The scope is rather basic as the DMU does not perform complex tasks concerning the actual content but mostly organizes the data synchronization and I/O job emission. The message contains a *Fetch*, *Write*, *Remove*, and *Cow* option. The first three variants are the basic operations the DMU performs and are regularly reported each time some disk operation has been committed. The *Cow* variant might be the most interesting option, as this notifies the *Migrator* that the given node has been moved from the active nodes [2] to the dead nodes[3]. Allowing for optimizations such as the removal of dead nodes from active to archive storage.

This message type may be used to implement an opportunistic promote or demote mechanic which lazily moves data between storage tiers. For this purpose, the DMU has been expanded to produce the described messages at important points in the object life cycle as we show in Figure 4.6. Processing of once produced messages is done in a message passing scheme, which decouples the writing and fetching process entirely from any influences possibly introduced by the migration policy. We have integrated the messages at all levels of the interface, except for *SuperBlocks* which are managed with raw access to predefined locations in each individual disk.

Importantly, DML messages only include nodes which have been, at some point in this session, written or fetched at least once. Any nodes which are on disk from previous sessions cannot be considered. Retrieving the disk offset from inactive parts of the storage system, for example with an initial scan, would be prohibitively costly as a considerable amount of data may be stored at the internal nodes, slowing down the startup, with not much improvement in performance. Especially huge storage backends make this choice undesirable.

```rust
pub enum DatabaseMsg<Config: DatabaseBuilder + Clone> {
    // Relevant for Promotion and/or Demotion
    DatasetOpen(DatasetId),
    DatasetClose(DatasetId),

    /// Announce and deliver an accessible copy of active object stores.
    ObjectstoreOpen(ObjectStoreId, ObjectStore<Config>),
    ObjectstoreClose(ObjectStoreId),

    /// Informs of openend object, adjoint with extra information for access.
    ObjectOpen(ObjectKey, ObjectInfo, CowBytes),
    /// Informs of closed object, adjoint with extra information for access.
    ObjectClose(ObjectKey, ObjectInfo),
    /// Frequency information about read and write operations on an object.
    ObjectRead(ObjectKey, Duration),
    /// Report the written storage class with the new size of the object.
    ObjectWrite(ObjectKey, u64, StoragePreference, Duration),
    /// Notification if a manual migration took place.
    ObjectMigrate(ObjectKey, StoragePreference),
    /// Notification similar to ObjectOpen but with different semantics.
    ObjectDiscover(ObjectKey, ObjectInfo, CowBytes),
}
```

Listing 4.3: Database Message definition.

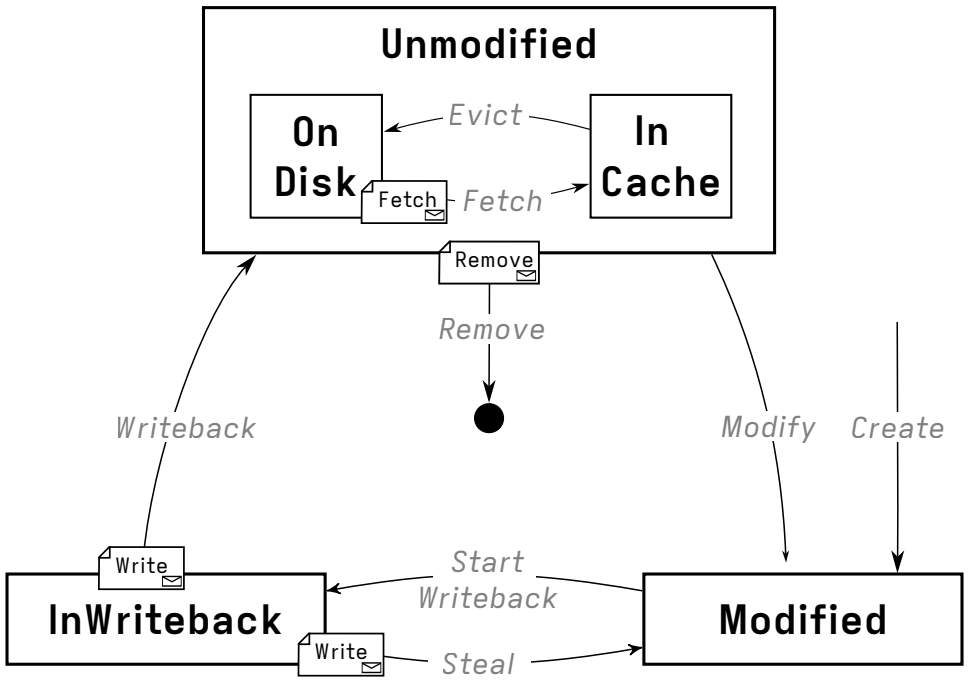


Figure 4.6.: The object lifecycle annotated with the according DMU messages. Graphic adopted from Wiedemann (2018) and Höppner (2021).

4.4.2. Object Message

The second message type handles object stores, objects and datasets, where migrations may also be performed proactively by the migration policy. To allow this, the *ObjectStore* and *Dataset* have been extended to allow for unscoped safe thread usage.

File like migration policies can be implemented based on this interface for objects, while *Datasets* are, due to their generality and therefore absence of migration functionality, only informational. Although, this information can be combined to enhance of the accuracy of the lazy node level migration based on DMU messages.

All possible variants of the Object Message can be seen in Listing 4.3. For brevity, a short explanation for each variant is added in the listing.

4.5. Node Identification & Migration

Previously, due to the nature of the storage, it was not possible to identify certain nodes in a tree. Existing access structures are solely based on the locality of a node on paths to key values whereas internal nodes are completely disregarded. Naturally, when observing the placement and eventual migration of these nodes we would like to be able to identify them. This is mainly concerning for leaf nodes as they normally carry the bulk of data. Internal nodes buffer some update messages and therefore contribute a substantial part of the total storage requirement but less so than leaves and they can be shrunken further by flushing operations.

[1]A flexible storage framework for HPC.
[2]Those which can be accessed from the current tree.
[3]Nodes which are only accessible from *Snapshots*.

```rust
pub enum ObjectRef<P> {
    Unmodified(P),
    Modified(ModifiedObjectId),
    InWriteback(ModifiedObjectId),
}
```

Listing 4.4: Variants of an object (node) reference.

To enable node identification, we added a transitive model, which can identify nodes by their history of storage positions. Reasonably, we can assume that a position will at a singular moment be unique and additionally only concerns objects in which we are interested, namely nodes consuming storage space. For identification of the node in the future we need to know which position this node was stored at, it is sufficient to be aware of the two most recently stored position as logic within a migration policy will keep, if required, track of the locations of individual nodes. Following this construct, an additional field, storing the previous storage location, has been added to the internally used object keys, which initially was only used for identification in the internal cache. This field is then send as an additional information in the DMU messages as shown in Section 4.4.1.

4.5.1. Migration Limits

While we have now solved the problem of node identification, more problems exist, when handling individual nodes. On further exploration of the possibility of node migrations, the most severe limitation is the actual access to nodes.

The design of the implemented B^{ε}-tree is monolithic with exclusive tree locking mechanisms to ensure the consistency of the contained nodes. References to a node are required to be unique, meaning that the reference from a parent to child node may not be replicated or used in any other location to access a specific node, doing so would violate the consistency of the storage stack as node references (in code also named “ObjectRef”) are modified on access to represent the new object state. The possible states can be seen in Listing 4.4. An access, fetching data from a node, would transpose the “ObjectRef” from the first variant “Unmodified”, which stores a disk offset and some data required for error checking, to the second variant “Modified”, which stores a cache reference. Importantly, a reference of the first variant cannot identify the location of the second variant.

Continuing on, an access from the *Migrator* could move a reference to the “Modified” state and later on an operation on the original tree, accessing this node, cannot find the cache entry, in which the already fetched node is located. Subsequentially, it fetches the data again and loads it as a new cache entry, this essentially duplicates the node. On eviction from cache, this leads to writing slightly different modified content to disk twice, with the loss of the location of the first modification and obstruction of storage space as the new location of the first node will never be freed.

Solving this problem, we need a measure to keep references in the tree consistent. Here two main objectives have to be combined. First, there has to be a low-overhead method to confidently identify *and* access nodes in an existing tree, be it via shared references or path identification. Second, the synchronization procedure needs to be able to locate modified

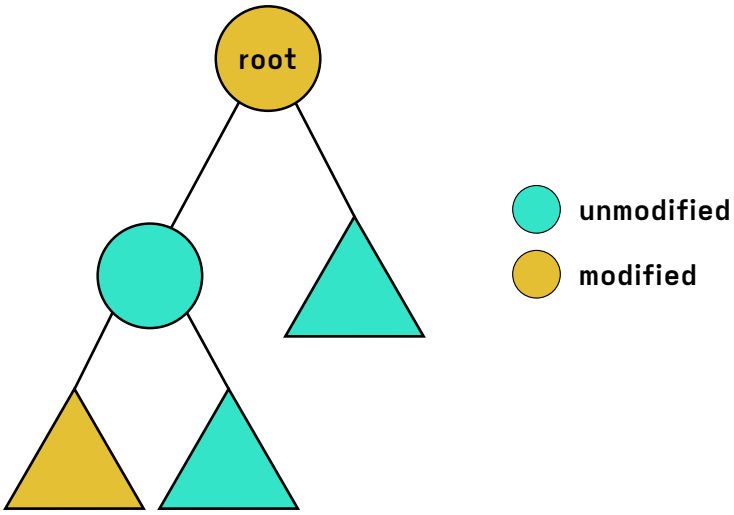


Figure 4.7.: A *modified* node with *unmodified* parents will lose state information.

nodes so that a write back may be initialized. We show this problem in Figure 4.7, where a synchronization is started from the root node, as is always the case in our code base, and the modified node in the bottom left is due to the unmodified parent not written back. This leads to data loss or even possibly irrecoverably corrupted the dataset. Beginning with the next session we may encounter an untreatable error on trying to open the previously modified but not synchronized subtree.

Naively, we can ensure consistency of references by initializing node references with locking mechanisms, such as a read-write lock, which is an optimized version of a mutual exclusion lock. This still incur some performance loss but less so than a full mutual exclusion lock, as read operations can be performed simultaneously. Although, this does not solve the second problem of missed synchronizations. Building upon this locking style solution, we would need some way to update the parents, which is only possible by keeping references to the them. This inevitably introduces cycles in the trees which would imply that on synchronization, if the root node is modified, which is almost always the case, all nodes of the tree would need to be updated to accommodate for cascading updated pointers to parents[4]. Therefore this solution is unfit as the overhead would be prohibitively large.

Alternatively, we could identify nodes based on their pivots, which would allow us to use the existing search property of the tree. We could use the node identification, proposed previously in this section, to identify the node in the tree. This would lead to costs of the migration operation at the same level as a regular query. Although, the implications of this have to be analyzed further, especially since the transitive model can lead to outdated identifiers. The question, how we ensure that, just before a query, a node is changed in a way where identification is no longer possible, for example on restructuring where pivots are moved due to splits and merges. The analysis of this is left for future work.

As a third alternative we can introduce actual separate identifiers in the tree, which would not change over time. But retaining search capabilities in this construction is again left for future work.

[4]Pointers will definitely differ due to CoW

4.5.2. System Storage Preferences

The messages we receive come, as explained, from the fetching and eviction and synchronization phase of the DML. This leads to the problem of combining user hints, in the form of storage preferences, and system decisions. If we assume that we want to migrate one complete node downwards to a different storage tier, all contained entries may not have any faster storage preference than the migration target tier. This is due to the storage preference being a *lower* bound. Therefore, we would need to modify the existing storage preference of keys[5] or block this node, meaning we may never migrate it below the current tier. Our remaining options are to skip this node and proceed with the next chosen by the policy or forcibly migrate this node below the actual user-specified tier.

The implementation of an assigned node storage preference, chosen by the automatic migration policy, facilitates an additional member of the leaf struct. This new member carries a system storage preference, which acts as an pinning of the storage tier range defined by the lower bound of the storage preference.

The determination of the storage tier based on the two storage preferences can then be structured as the following problem, which we can handle in two different approaches:

- $a > b$ - a is faster than b
- $\Rightarrow$ - result of the combination of preferences as proposed
- S_p - a system storage preference
- S_n - an unset system storage preference
- U_p - a user storage preference
- U_n - an unset user storage preference

$$S_p > U_p \Rightarrow S_p \tag{4.1}$$

$$S_p = U_p \Rightarrow U_p \tag{4.2}$$

Preference Honoring

$$S_p < U_p \Rightarrow U_p \tag{4.3}$$

This approach is comparatively less intrusive and more true to the user hinting. We depend on the user to make future-proof choices and manually move data, which they deem not critical, to lower levels. Keeping track of storage usage and picking data to migrate is of high effort to an user.

Users depend on the storage preference right now to define which level of minimum access latency they are willing to accept (lower bound). In general, we aim to fulfill this promise on our side, but on systems with at capacity disks this will, depending on the fallback configuration, either lead to allocations in different tiers (higher or lower ranked) or to errors when writing.[6]

[5] As done when performing migrations based on keys.

[6] These errors are no longer untreatable as we fixed a few bugs locking the cache state in an unresolvable configuration.

As nodes are allocated on the slowest desirable free tier, a conflict between user-given preference and system-decision preference will be the case in almost all auto migrations which demote the current node. Honoring the preferences, will not ease any situation of at capacity tiers as nodes will likely not be able to be moved.

Notable in this context is, that the fallback policy already does not honor the actual storage preference in trade-off to system stability. Though the treatment of these situations is agreed to by the user via a contract in the *DatabaseConfiguration*. Notably, the default values of the configuration do not perform any kind of fallback and simply error on full tiers.

Preference Abandonment

$$S_p < U_p \Rightarrow S_p \tag{4.4}$$

An alternative to honoring the user preference is the abandonment of contracts in the demotion case. Since we perform demotions in configurations where a large number of blocks are already allocated and future accesses might encounter full disks on their desired storage tiers, it can be argued that the ignoring of the initial user estimation in favor of current storage tier states is of advantage to the user due to an increased stability and performance. Furthermore, we preserve the storage preference of the user and may be able to restore the user desired distribution, once tier space has been freed.

4.6. Default Promotion & Demotion

Due to the design of the *MigrationPolicy*, we allow for an easy integration of new policies. Moreover, we provide a standard thread loop which can be used by any policy. This loop performs two main operations, *Promotion* and *Demotion*. The policy can choose to use both object based migration and node level storage hinting, each with their previously discussed downsides.

The default thread loop is designed to optimize for low latency accesses and eager migration upwards, while simultaneously, only demote on storage tiers which overstep a user definable threshold.

There is a trade-off to be made with this design, as this approach might perform more migrations than necessary especially when moving data between two adjacent tiers possibly alternating objects between them. A policy approach which applies migrations based on a temperature might perform better in some cases if it disregards the default implementation and uses a specialized implementation.

4.7. Space Accounting

A fundamental requirement for data migration decision making is the actual storage utilization at each moment. Previously, this information was encoded only in the “Segment” bitmaps, which track the actual spatial distribution of allocated buffers, handled by the DMU. To avoid

```rust
pub struct Superblock<P> {
    magic: [u8; 9],
    pub(crate) root_ptr: P,
    pub(crate) tiers: [StorageInfo; NUM_STORAGE_CLASSES],
}
```

Listing 4.5: The superblock structure.

expensive free space recomputation, from this, for our purposes, complicated data structure, we have added atomic counters, which represent the number of free blocks in tiers.

We located this space accounting component in the allocation handler of the DMU, to track and count the number of blocks allocated on disk. Furthermore, since the aggregated number of all disks of each tier will be required regularly and frequently in the migration logic, additional counters have been added to count the aggregated result next to individual disks. The choice for atomic counters allows for a safe usage and does not invalidate the requirements for the "Sync" trait (which is required for the compiler to validate that a shared reference is safe to use *and* modify over multiple threads).

Since these counters lose their state on reinitialization of the storage stack we have to ensure persistence over multiple sessions. Therefore, we allocated some space of the *SuperBlock* to contain this information, as the *SuperBlock* is 4KiB in size and until now only contained a single pointer to the root tree. The current structure of the *SuperBlock* is shown in Listing 4.5 and is ,as portrayed, serialized onto disk. In total, this addition of the superblock enlarged it to 110 bytes, which leaves plenty of space left for future modifications. Thus we see this extension as unproblematic for future references.

Basic error handling has been implemented by using space accounting information. For example, we can return errors early if not enough space is available on a tier to which we want to migrate an object to.

Cache Uncertainty

Omitted from the space accounting implementation, are size considerations of the currently kept cache values. This can lead to problems in some situations, as for extended error handling and space checking, we are not guaranteed that the space required for a node to be written is available. Even if, space accounting at the moment of writing indicates so. This is due to the fitting algorithm implemented to be performed during a node write-back, as the final size of a node is only available at this moment.

This results in a state of uncertainty in which we cannot ensure that all data currently held in cache can be written to disk. This is the case exactly when

$$f(t) - requested\_size < max\_cache\_size$$

where f represents the free space of a tier t. Note that, this is a worst case assumption and factors like cached nodes which have been written at least once, therefore eventually free some storage, due to CoW semantics, if they are not contained in a snapshot, and the placement of nodes in the cache besides the target tier are ignored. This uncertainty tends to be negligible

on the default cache size of 256 MiB but with an increasing cache size this will become a problem.

For future work, it might be valuable to investigate how this uncertainty can be reduced or eliminated. We propose that the cache should be aware on how much space is likely to be used up by its members on which exact tier.

4.8. Implemented Candidates

In this section we briefly explain and argue for two candidates of migration policies we have chosen and implemented in this thesis.

4.8.1. LFU / MFU

In cache eviction, the determination or approximation of the least frequently used items is a common technique, as we prefer to reduce the data traffic from disk to memory or memory to l3 cache. For example, in ZFS an approximation of LFU is used in the main cache, namely the ARC algorithm which generates LRU and a second layer LFU approximation, dynamically adjusting between them.

As already mentioned we used an implementation of Matani et al. (2021) as the basis of this policy due to it's complexity of $O(1)$. We extend this structure in this thesis with a MFU interface, which requires an addition of tail tracking, to the already existing head tracking, in the doubly linked frequency list and is therefore easy to realize. This again requires constant effort and does therefore not change the overall cost.

This algorithm, used internally in *Haura*, is written using unsafe Rust, due to the extensive use of self-referential pointers in the chosen data structure. We will briefly give a description of unsafe Rust here.

Unsafe Rust

"Unsafe" is a special keyword in Rust which deactivates most of the safety mechanisms, that enable Rust special runtime guarantees. Most importantly for our case, it allows raw pointer handling.[7] As seen in our example it is helpful in some situations to deactivate these mechanisms to allow a more unrestricted design of data structures, this though requires extensive testing of the produced code to prevent bugs from occurring. Critically, we import the algorithm as a "crate"[8] and hide the unsafe usage from the user, so we have to ensure that our use of unsafe does not trigger any bugs in foreign code. While implementing said features of this thesis, we actually found some bugs in the already existing implementation of the LFU cache. [9]

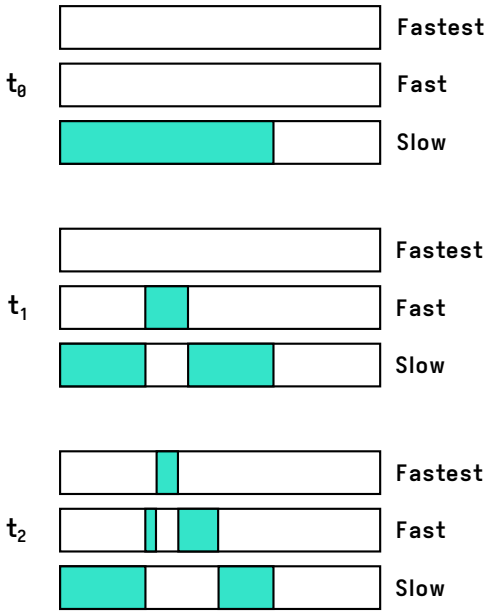


Figure 4.8.: Schematic view of a partial promotion of active regions in a singular object.

Version A: Node Hinting

Based on this implementation, we have implemented a frequency tracking for individual nodes based on the transitive identification already discussed in Section 4.5. To distinguish, and be able to recommend operations for specific tiers, we hold multiple of these LFU lists, each representing a single tier in our hierarchy. We know for accessed nodes at which tier they are located, as we receive these information from DML messages and update them according to the most recent operation performed. Including, moving nodes between tiers with preservation of their frequency, to allow for quick upwards mobility between tiers.

Importantly, the policy never performs an *active migration* of a singular node, due to construction restraints, as shown in Section 4.5.1. The policy only stores hints which the DMU then respects. While unproblematic for promotions, as we have a high probability to access hot data again and otherwise we do not need to promote this data anyway, this makes demotions difficult, as nodes which are preferred to be downgraded are also simultaneously less likely to be accessed and therefore migrated by hints.

Currently, migrations are always single stepped, so a node might be hinted to be promoted from *SLOW* to *FAST* and from *FAST* to *FASTEST* but not immediately from *SLOW* to *FASTEST*, even though for some cases this might be the better decision. Although, we argue that this would require an additional higher inertia for moving nodes to avoid a kind of double-caching in the *FASTEST* tier for data which is used very frequently in a short time period, as this should only be cached in the DRAM CLOCK cache. Implementation of this would imply again some boundary values to be set, which is an additional configuration effort and is therefore a requirement unrealistic to be fulfilled by the user and difficult to estimate on a general basis.

In Figure 4.8 we show for a single object schematically the promotion of a range of nodes (a selection of chunks in the object) from the user perspective. In timestep t_0 an object starts being

[7] https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html
[8] A dependency package in Rust.
[9] https://github.com/jwuensche/lfu-cache

located entirely on the *SLOW* tier, with regions of more active use being lifted to the *FAST* tier in t_1, while in t_2 the most active part of this range again is moved to the highest *FASTEST* tier. Parallel to this, we migrate the next most active part from the object on the *SLOW* tier to *FAST*. Furthermore, node migration also implies that the average size of a migration using this method will always be about the size of a node which is fixed in the tree at 4MiB, which allows for reasonable granular control of regions within an object.

This version has been considered first, but ultimately only partially implemented due to the demotion and access issues, which require a more major redesign of the storage stacks architecture discussed in Section 4.5.1, which will be left for future work.

Version B: Object Migration

For object migration, we use the same LFU implementation as for nodes, with the distinction of size buckets in which objects are inserted. These buckets are of a fixed number of instances, which represent each a certain size range. This additional differentiation, as proposed by Ge et al. (2022), aims to optimize for disk characteristics of storage devices as for larger serial data performance impacts are less severe than with comparable random access throughput, due to a lessened impact of for example search latency on HDD.

In total the amount a of LFU instances is determined by $a = m \cdot n$, where n is the number of tiers in the hierarchy and m is the number of buckets into which objects are inserted based on their size. For the range of sizes we decide to use the suggested range by Ge et al. (2022), which starts with zero-size to 256 KiB, then 256 KiB to 1 MiB, 1 MiB to 10 MiB, 10 MiB to 100 MiB, 100 MiB to 1 GiB, and last 1 GiB and larger. So in total, for a 4-tier setup, we create 24 LFU instances. Regarding the memory consumption, this leads to a small overhead as frequency nodes cannot be shared internally in the structure, though the impact of this is minuscule.

This design decision leads to an opinionated decision on the object distribution, as larger infrequent files are generally lower ranked than small infrequent files. Additionally, as we simply observe the frequency of objects, an optimal choice between very infrequent files and less infrequent large files cannot be easily made, the larger file will be demoted first if needed. We can reasonably assume that this is not optimal for some cases, and this problem domain might benefit from more extensive decision processes like multi object optimization. This is further discussed in Chapter 6.

Version C: Node Promotion + Object Demotion

We have shown that the eager key or object migration of version B has advantages over the node based hinting of version A, with version A allowing for a lazy “zero-cost” approach to migrate hot data. Combining both would bring significant benefits, as with active objects, we may reduce migration size to active regions and keep the number of write operations, when modifying data, the same even.

However, there are two road blocks which need to be solved beforehand. For one, we need to associate nodes with certain objects, this can range from zero to many objects and is dependent on the keys this node contains. This requires some interconnection between two detached layers of the storage stack, first the object store layer, which can produce information about objects, such as their size and committed operations, and second the DML, which handles all

```rust
pub struct LfuConfig {
    promote_num: u32,
    promote_size: Block<u32>,
    path_tier: Option<PathBuf>,
    path_delta: Option<PathBuf>,
}
```

Listing 4.6: Configuration options for the LFU policies.

nodes of the tree identically and would require additional logic to identify leaf nodes and then check its key values and propagate this information to the object store layer. This destroys the abstraction in the tree layer and produces a heavy interlinked structure with a worsened performance as this key lookup is of cost $O(k)$ where k is the number of keys in a leaf. This already, optimistically, assumes constant cost for information propagation and object matching. In total the cost for writing a tree would change from $O(l+i)$ where l is the number of leaves and i the number of internal nodes to $O(l+i+k \cdot \log_{B^\varepsilon} N)$ since the reporting, at worst, would be done on each level of the height of the tree.

As these major modifications and structure analysis have to be made before implementing this version, we have left it for future work.

Configuration

Some configuration options have been made available for these variants to control basic behavior, the types of which are shown in Listing 4.6. First, “promote_num”, which controls how many objects or nodes, depending on the version, can be updated in a singular time step. This can be used for example in the object case to limit the amount of small objects to be migrated to heighten inertia for small files. Second, “promote_size”, which controls the maximum size migrated in a singular time step. This mainly influences the speed of the migrations occurring in both node and object variants. The size is given in number of blocks, which is set to be 4 KiB. Third, there are two configuration options regarding state and operation metrics, which can be used for evaluation or debugging of policies. The “path_tier” option produces for each time step a JSON with all current locations of objects which is then appended to the given file. The “path_delta” option produces a CSV file containing all migration operations accompanied by additional information such as the size, the time step and the timestamp at the point of metric aggregation.

4.8.2. Reinforcement Learning

Originally, our chosen reinforcement learning approach has been proposed by Vengerov (2008). They propose a scalable model which uses independent agents (one for each tier) who can determine among themselves if a migration of an object should take place. This is important as it allows the model to have a relatively simple state description in contrast to a super agent over all tiers. This approach has been selected due to its scalability by another recent paper by Zhang et al. (2022), in which the authors apply this model to a cloud storage use-case and reaffirm the effectiveness of the approach. We have used both papers as references in our implementation which we explain in this section shortly.

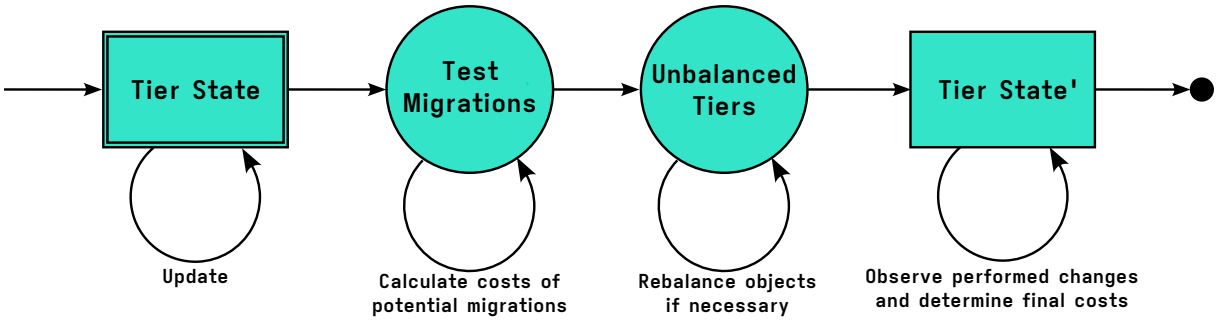


Figure 4.9.: Overview of the reinforcement learning iteration states. The “Test Migrations” and “Unbalanced Tiers” phases are the only actual operating on the storage stack.

Overview

Most important for our purposes is the objective of the policy which can be expressed as the state of the agent. The state is based on a temperature scale, which is applied to the objects similarly to the frequency scale as done in the LFU. We can define the state by three values:

- s_0 - average object temperature for all objects (active and inactive) in the tier.
- s_1 - weighted average object temperature (active and inactive), where the weighting is determined by the object temperature times the object size.
- s_2 - current latency of requests on the tier in the considered time step.

Significantly, the last variable differs from the definition given by Vengerov (2008) and Zhang et al. (2022) as they use the queuing time on the working queue to indicate the time required for data to arrive, while we use the total latency of the operation as an indication for how long we actually need to write or read the required data. We therefore, also differentiate between complete object access, where the full object is read or written, and partial access, where only a slice of the object is active.

To actually interpret the state and state changes of the tier state, we use Fuzzy Rulebase (FRB) to receive a discrete input, from our floating point state values, (in fuzzy categories named “Large” and “Small”) which we can then use to specify rules which our implementation should follow.

The most important aspect of this fuzzy approach is the possible overlap of rules allowing partial consideration in the final result. This is not possible with traditional approaches, like LRU, and may bring benefits in complex problems like data placement. The chosen fuzzy approach is at the same time the simplest categorization, with only two categories, and may be expanded if needed with additional categories. (Vengerov, 2008)

In the overview in Figure 4.9, we show the basic steps in updating the storage stack with our implementation. In the first step we simply update the initial state, based upon incoming recorded requests, and update the temperatures of each object. We explain this step and all following steps in the next subsections.

Temperature Update

The temperature τ of an object k is determined by their last access time, which is saved with other file information. To avoid high sensitivity and a double-caching as described before, the new temperature is only partially applied based on a factor α. Vengerov (2008) gives this formula to allow for a sliding temperature value.

$$\tau_k = \alpha\tau_k + (1 - \alpha)(T_{now} - T_{last})$$

This update is performed on start of the iteration in the update step of the original tier state. To ensure fairness for all entries a fixed time is used for reference which is taken at the beginning of the iteration.

Promotion Decision

Next, follows the actual performing of migrations, chosen from the active objects. For each object, we determine the costs it will cause in the future if it stays in the current tier and the future costs it will invoke if the object is migrated to the next higher tier. We migrate the object if the future costs of both tiers added are lower when the object is moved compared to when the object is *not* moved. In the formula given by Vengerov (2008), where C is the tier cost, s_0 the average file temperature, $\tilde{s_0}$ the average file temperature of the modified state, and t_0 the upper tier and t_1 the lower tier (on which the object currently resides).

$$C_{\text{up}}(t_1) \cdot \tilde{s}_0(t_1) + C_{\text{up}}(t_0) \cdot \tilde{s}_0(t_0) < C_{\text{not}}(t_1) \cdot s_0(t_1) + C_{\text{not}}(t_0) \cdot s_0(t_0)$$

When all *active* objects have been considered, all tiers will be checked again if they overstep their defined migration thresholds and the coldest files of a tier will be migrated downwards, if necessary, until enough space has been freed.

Learning

Following, the new state of all tiers is evaluated and compared to the initial state of the tier at the beginning of the iteration and the reward, which is determined as the average response latency of operation in this tier. The chosen learning agent is the TD(λ) agent, which is fit for an iterative approximation of the solution. Normally, this agent is used for Markov Decision Process (MDP), which are dynamic system decision makers, for a continuous time process, like our application, they are also known as Semi-Markov Decision Process (SMDP). Regularly, SMDP are aware of the entire state of the system, which is not given due to construction considerations, to reduce the complexity of the measured state, which makes our system a special case called Partially Observable Markov Decision Process (POMDP). Although, Vengerov (2008) showed that this agent may also be applied for the given POMDP problem. (Ge et al., 2022; Puterman, 1990; Vengerov, 2008)

Randomization

In our algorithm implementation there are a few points where randomized data is used to initialize from an unknown state, for example if an object is newly inserted it is assigned a random temperature. We have chosen a uniform randomness and put good faith in the chosen algorithm[10], but other options such as range restricted random number generation or fixed value initialization is possible and may yield slightly differing results.

Optimizations

Regarding possible optimizations, the temperature scaling can be improved further and should be improved if scaling to multiple hundreds of terabytes is required. As for now, the data is stored in a hash map, which results in a cost of $O(m)$ for finding the object with the minimum temperature in a tier, where $m \leq n$ and n is the number of objects in the database. The same cost has to be assumed for updating temperatures at start of an iteration. Though, due to the constant number of update iterations, one for each tier, and the fact that the sum of the number of objects per tier is in total at maximum m, the costs in total still remain $O(m)$.

It may be worth investigating if the LFU structure, or similar, can be used to reduce the costs in large scale systems.

4.9. Summary

We have presented a concept on how to realize and integrate migrations and automated migrations into the *Haura* storage stack in this chapter. In the design process, we have encountered multiple challenges where trade-offs or changes to the existing design were required, due to the incompatibility to necessary operations in data migrations. Also, basic functionalities like key metadata updates and space accounting have been added to enhance the usability of the storage stack.

Furthermore, we have envisioned an exchangeable policy architecture allowing for rapid experimentation setups and quick evaluations. The design of this interface required the definition of standardized messages, which can be used to understand and modify the current state of the storage system, to better fit criteria aimed to be optimized by the policy.

Finally, we have presented two policies, which have been implemented as a configurable policy. One is a static heuristic, offering both LFU and MFU, which includes three different variants acting on nodes, objects or a combination of both. The other is a learning heuristic using reinforcement learning. This implementation references previous related works and builds on the knowledge acquired from their experimentation. We construct the basic design again here and point for further reading to Vengerov (2008) and Zhang et al. (2022). We will evaluate both policies in Chapter 6.

[10] A crypthographically secure pseudo-random number generator provided by the “rand” crate.

Chapter 5.

Changes

This chapter contains various changes we have made to the *Haura* storage stack. The described changes are either only tangentially related to the topic of the thesis or more in-depth than necessary in the concept. We advise you to read this chapter if you are interested in the internals of the modifications done. For example, if you want to introduce further modifications in this domain.

5.1. Cache

To enhance the usability when creating benchmarks especially with measuring latency, we have added a method to clear the current cache, writing back any modified nodes and evicting them from cache. Internally, this is not so different from the usual eviction procedure, which performs this operation until the restricted cache size is adhered to. Importantly, not *quite* all nodes may be evictable, if nodes are pinned and do not get unpinned until the procedure has ended, we will discard them and focus on evicting the remaining nodes until no node may be evicted anymore.

We use this function extensively in the evaluation of this thesis. Especially, we have noticed an increase in reliability when using this compared to other methods like the replacement of the cache content by performing large operations on some unrelated dataset. Due to the limited usefulness in non-benchmark scenarios, we hide this feature behind a feature flag of the crate. More information can be found in the code documentation.

5.2. Dataset

With the extended parallel usage of datasets required for policies, we have encountered a problem with the current opening logic, as at most one instance of a dataset is allowed to exists simultaneously. To fix this, a wrapper around the actual *Dataset* type has been created which acts as the user interfacing dataset type, internally, on each call on this super structure a read-write guard prevents the unsafe parallel write usage. Calling operations in this manner, allows for fearless duplication of dataset instances.

Notably, if one of the datasets is actively closed others may continue to be used, as the closing logic checks the count of active references to the read-write lock.

```rust
pub struct ObjectInfo {
    pub object_id: ObjectId,
    pub size: u64,
    pub mtime: SystemTime,
    pub pref: StoragePreference,
}
```

Listing 5.1: The metadata struct stored for each object.

5.3. Object Store

We have introduced minor changes to the object store to supplement the new *Dataset* wrapper. The object store now supports cloning operations, so that, additional threads may take ownership of the object store instance, allowing easier parallel use. The closing of a cloned object stores behaves similar to the logic specified for the dataset, with an object store being closed definitively when the last reference has been closed.

Object Preference

To allow objects to continue using their previously given preference after having been closed, we have introduced the preference as part of an objects metadata. This field is updated on opening or writing to the object. We may use that field when an object is opened without preference given. We show the new metadata structure in Listing 5.1.

5.4. Tier characterization

Initially, we aimed to add more user hinting capabilities to the storage stack in this thesis but discarded these due to time and scope concerns. One of them is inspired by the "streamID" of Ge et al. (2022), which seems to integrate well with our storage design. In this hint, we assign to each tier a preferred access pattern or access pattern group. For example, for HDDs we may set a disk or tier to prefer serial read and write operations, while a tier backed by SSD may prefer serial read operations. If we further allow objects to set a majority access pattern type, we can match objects and tiers to be optimally fitting in respect to their preferred access patterns.

Especially, when creating new objects this allows for an easier determination of storage tiers. For older objects we will need to investigate how strongly the fragmentation introduced by CoW negates this optimization. For example a large older object with many rewrites may prefer to be stored on SSD even if it is only read sequentially.

Chapter 6.

Evaluation

In this chapter, we evaluate the implemented concept based on five different cases, which we will create and conceptualize beforehand. We design these cases to mimic common use-cases in the domain of HPC and databases. Importantly, differing access patterns are covered, which will act as basis of comparison between the two migration policies defined and the baseline, which is a run of each case without any migration performed.

First, we will discuss the implemented cases, which are divided into two categories, workloads mimicking a real workload on *Haura* and short benchmarks simulating specific situations, which might be encountered during real applications. Later on, we compare the results of each case, especially with respect to location optimality and I/O time reduction.

6.1. Setup

The system configuration consists of three tiers. The *FASTEST* tier is backed by DRAM and is 8 GiB in size. Notable is, that with the used internal cache of size 256 MiB, the DRAM is doubly used in the stack which might incur some performance loss.

The *FAST* tier is backed by a NVMe-SSD (Toshiba XG6 Series KXG60ZNV256G) on which ext4[1] is used. For *Haura* a 16 GiB file has been truncated which backs the data from the *FAST* tier. To avoid caching by Linux and side-effects by ext4, the I/O "Direct" mode is used when using the files. This bypasses common caches such as page caches and internal caches of ext4.

The *SLOW* tier is backed by a HDD (Western Digital WD5000AAKS) also containing an ext4 filesystem onto which we truncated a file of size 100 GiB, to use with *Haura* with the "Direct" I/O flag.

We allowed each policy a grace period where no action is performed. This is mainly intended for stability as not all bugs were fixed in the course of this thesis especially when a large percentage of the storage system is filled in a small amount of time. The recording of metrics is also started after this point.

To ensure the error-less functionality of the benchmarks, we set the migration threshold[2] quite low at 50%. In practice one would choose a value much higher, well above 90%, or simply determine a set size of storage kept as a buffer, for example 2 GiB.

[1] A filesystem made for the Linux kernel. `https://ext4.wiki.kernel.org/index.php/Main_Page`

[2] The threshold at which the migration policy does not try to fill a tier further.

```
"migration_policy": {
    "Lfu": {
      "grace_period": {
        "secs": 300,
        "nanos": 0
      },
      "migration_threshold": 0.5,
      "update_period": {
        "secs": 10,
        "nanos": 0
      },
      "policy_config": {
        "promote_num": 99999,
        "promote_size": 10240,
        "path_state": "tier_state.jsonl",
        "path_delta": null,
        "mode": "Object"
      }
    }
  }
```

Listing 6.1: Configuration of the LFU policy.

The specific configuration of the LFU policy can be found in Listing 6.1 and of the reinforcement learning policy in Listing 6.2. Both are an excerpt of the larger *Database* configuration and share some options as explained in Section 4.3.

We configured a `grace_period` of 300 seconds for both policies to avoid errors when quickly filling storage tiers within the specified update. These are still under investigation though unlikely to occur outside of benchmarks. Next, we defined a `migration_threshold` which is a floating value from 0 to 1 and denotes the percentage a tier is supposed to be filled to by the policy and when to migrate data from the tier. In real use-cases this would be, as already explained, either a much higher percentage or replaced with a total value in bytes. We set it to 0.5 as the latency measurements at the end of the *Mimic Workload* fills approximately half of the *FASTEST* tier. The `update_period` simply describes the time between two iterations in the *Migrator* main loop. This is also set to be quite low at 10 seconds for the benchmarks, in experimentation in this thesis we see that higher values for actual long-term use are feasible and perhaps desirable. Lastly, the `policy_config` is the policy specific configuration, in Listing 6.2 only concerning the metrics output. In Listing 6.1 we have the additional options of `promote_num` and `promote_size` which limits the size of objects or nodes which are allowed and the number of objects or nodes to be promoted in one iteration respectively. Importantly, the size is given in the number of blocks, so a size of 10240 would be equivalent to 40 MB. Which puts the load this *should* generate at just around 4 MB/s. The `mode` option specifies which LFU version we want to use according to Section 4.8.1.

```
"migration_policy": {
    "ReinforcementLearning": {
      "grace_period": {
        "secs": 300,
        "nanos": 0
      },
      "migration_threshold": 0.5,
      "update_period": {
        "secs": 10,
        "nanos": 0
      },
      "policy_config": {
        "path_state": "tier_state.jsonl",
        "path_delta": "tier_delta.csv"
      }
    }
  }
```

Listing 6.2: Configuration of the reinforcement learning policy.

6.2. Mimic Workloads

This section contains a description of the implemented mimic workloads. They can be roughly categorized as synthetic and snapshot distributions, whereas synthetic means that we take reference values of file size distributions and fill objects with randomized data of these sizes. A snapshot distribution is data generated by two scientific applications which serve as an example for realistic HPC size distributions.

6.2.1. Synthetic Distribution

The first kind of distribution is designed to imitate a file size distribution of very large scale scientific HPC systems. We investigate two surveys (Welch and Noer (2013) and Meister et al. (2012)) for this, both scouting filesystems of large data centers like the DKRZ, the Lawrence Livermore National Laboratory (LLNL), and Los Alamos National Laboratory (LANL). These surveys have shown that while there are distinctive patterns to be found in the data, especially the LANL which observes large spikes at certain file sizes probably due to certain applications being run frequently, producing similar sized output. All surveyed systems are populated with a majority of small files (<64 KiB). This might be in some systems up to 60% of files, while the majority of storage space is occupied by larger files, small files only account for 7% in this constellation. (Welch and Noer, 2013)

6.2.2. Snapshot Distribution

The snapshot distribution is based on data generated by two scientific applications, namely XCompact3D[3] (by Laizet and Lamballais (2009)) and NWChem[4]. Next to these applications, scripts, input data, documentation, log, and graphics are included in the snapshot representing a complete run of these two scientific applications with multiple checkpoints. The scope and variety of data can be used well to represent a user directory in a scientific cluster environment. The total size of the snapshot is about 30 GiB.

In the actual benchmark, we represent each file with an object in our storage stack. The key to this object is the complete path to the file.

6.2.3. Methodology

In both distributions we create three distinct groups from the initial set of objects. The first, represents a small amount of objects (5 %) which is accessed often (90%) of iterations, we call this group "Often". The second, includes a slightly larger group of objects (15 %), which are occasionally accessed (20%), we call this group "Occasional". Lastly, the majority of objects (80%) are seldomly accessed (1%). These represent the bulk of data in the distribution, we call this group "Seldom".

We track for each group the mean tier onto which it is stored, as well as the storage tier of individual objects. This is included in the policy as configuration option to write metrics into two files. One contains of each iteration the complete state of the objects in the store, as recorded by the policy. Another, reports all migrations performed by the policy during an iteration.

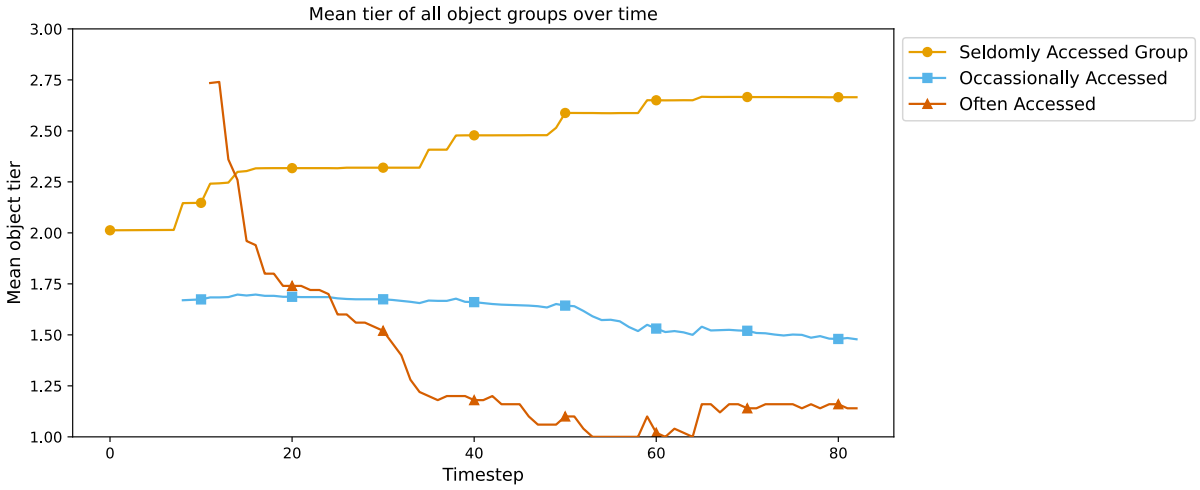
The complete benchmark is divided into two phases. First, the benchmark fills the storage system with data randomly generated or read from a snapshot. The tier in this case is chosen at random and it is taken care that no tier is overfilled, to avoid errors early on. Second, a "conditioning" phase is performed, where we use the access pattern with access probabilities as described before. This allows some time (20 min) for the policy to distribute data based on the current accesses. In the end, we perform measurements of latency on uniform samples from each group. We expect that, the "Often" group will experience the lowest latency, followed by, the occasionally accessed objects, and then the seldomly accessed objects. Also, to avoid falsifying the results, we empty the cache after each access to minimize the influence of previous access to the current measurement.

We compare in these benchmarks the LFU policy, the reinforcement learning policy, and as a baseline, the storage stack without any migration policy.

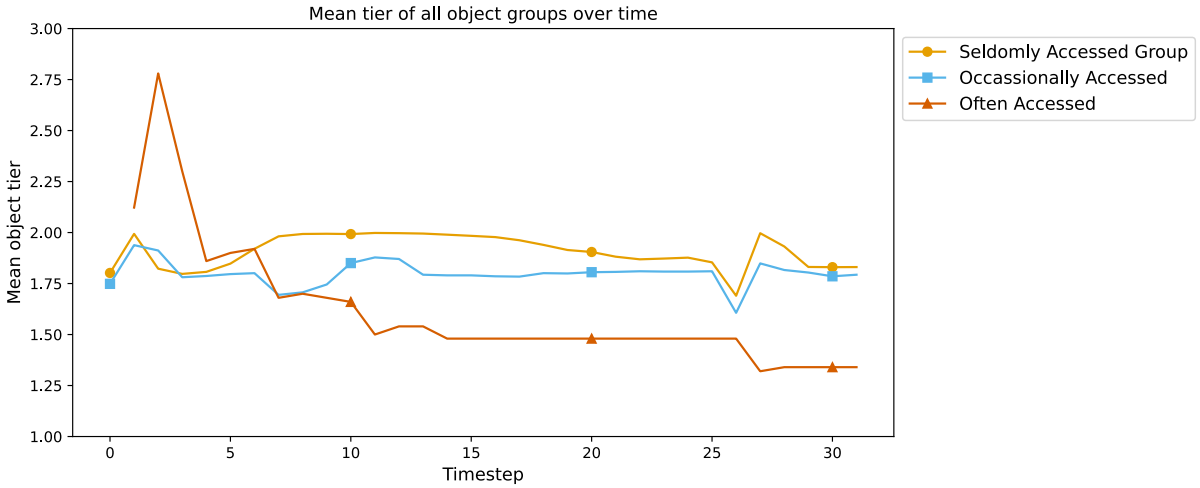
6.2.4. Results

In this section, we compare the results of the benchmarks performed with each policy. We first investigate how the policy choice affects the mean object distribution in the individual groups and compare these results with our expectation based on their access probabilities. Then, we

[3] A solver for incompressible and slightly compressible flows.
[4] A toolkit for quantum chemistry and molecular dynamics. `https://nwchemgit.github.io/`



(a) Reinforcement learning policy



(b) LFU policy

Figure 6.1.: Mean tier level of object groups over time.

will show the exact object distribution and investigate how the size of an object affects the choice of storage tier made by the policy. Lastly, we compare the access latency observed when accessing uniformly sampled objects in each group and determine characteristics and effectiveness of each policy therefrom.

Synthetic Distribution - Group Means

In Figure 6.1, we show the mean object tier levels of the three defined object groups in each run with different policies used. To determine a floating mean value, we map the tiers to floating point numbers as follows, *FASTEST* is equivalent to 1.0, *FAST* is equivalent to 2.0, and *SLOW* is equivalent to 3.0. On first sight, we can determine that the initial spread of Figure 6.1a was much less homogeneous than in Figure 6.1b, which can occur due to the random nature of the distribution. The "Often" group was on average on the *SLOW* tier, with a value of 2.75, it is almost entirely stored there. In comparison, in Figure 6.1b the groups are much more dispersed over all tiers, ranging from 1.75 to 2.0 on average. Both policy manage to identify the "Often" group quickly after only a few timesteps. Although, the migration of the "Often" group in the

reinforcement learning policy is performed much faster, quickly decreasing the average from 2.75 to 1.75 and then slowing down the migration from 1.75 to 1.2. Interesting in Figure 6.1b is that, with the migration to faster tiers in the "Often" group, objects from the "Occasional" group are moved to the *FAST* tier, while in Figure 6.1a objects from the "Seldom" group are moved to the *SLOW* tier. This shows, with our knowledge of the given access pattern, that the choice of objects to demote is more effective in the reinforcement learning policy than in the LFU policy.

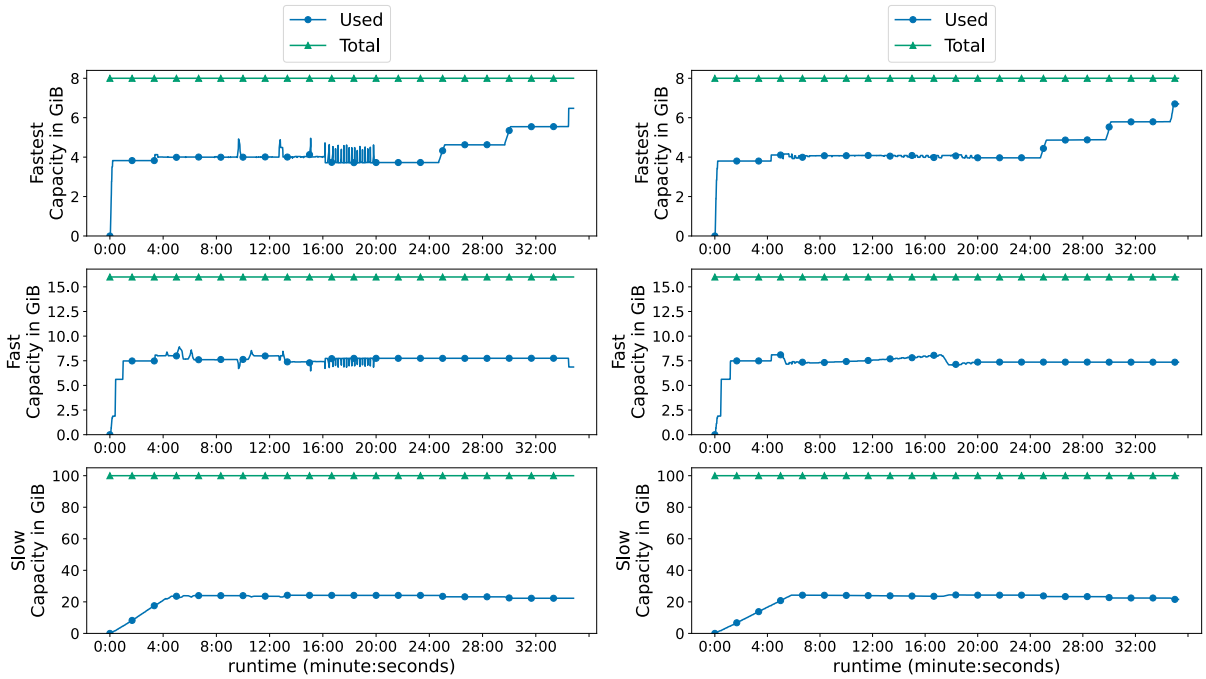
Notable is, Figure 6.1a exhibits a much larger spread of means than Figure 6.1b in the complete duration of the benchmark. All objects from the most accessed tier were even briefly located on the *FASTEST* tier, indicating a well reacting policy to the defined access pattern. Figure 6.1b shows to be less reactive and seems to encounter troubles when differentiating occasionally and seldomly accessed objects, as they are tightly coupled in the mean curve over time.

Also notable is, over time the "Seldom" group is treated due to this tight coupling similar to the "Occasional" group and similar to the "Occasional" group in Figure 6.1a with both curves decreasing till the end of the benchmark. This indicates that, while the identification must be improved to filter out falsely identified objects, the trend in both policies seems promising in terms of future access latency.

Synthetic Distribution - Tier Usage

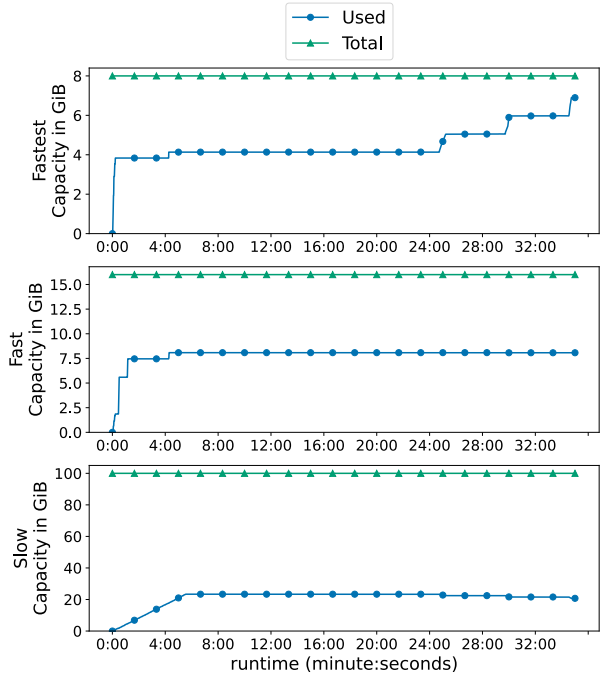
In Figure 6.2, we see the usage of storage tiers for each policy. A brief description of these graphs is sufficient as not too many differences exists. We can see that, as expected, no movement of data is occurring in Figure 6.2c, we merely see an increase in the end from the performed write measurements. More active, is Figure 6.2b in which constant but slow movement of data can be observed. This is also expected, due to the construction of the LFU policy where an upper bound of blocks has been configured in this benchmark. Tweaking this value could increase the migration speed and offer greater results, but this would put higher base pressure on the storage system and is difficult to estimate beforehand. This static value is one of the major downsides of the LFU policy as it is momentarily implemented. Feasible, would be the extension of the policy with further hinting or detection of "active" phases in which I/O is performed by a user and "inactive" phases if this is not the case. This could trigger more aggressive migration speeds, further utilizing unused resources, but increasing power consumption.

Figure 6.2a shows a greater deviation in migration speeds and exhibits multiple phases visible from the usage graph. First, migrations from the *FAST* to *SLOW* tier are occurring, promoting objects first upwards and then demoting others to fulfill the migration threshold again. This behavior represents the promotion of active objects in the *SLOW* tier which have been accessed in the first timesteps. Second, active objects are promoted from the *FAST* to *FASTEST* tier in short spikes over about one fifth of the total runtime. These could be either the previously promoted objects, but due to the multiple occurrences we suggest that these are a combination of already promoted and objects originally stored in this tier. Third and last, a phase of frequent migration is occurring with multiple burst of objects in short succession being promoted from *FAST* to *FASTEST* and the subsequent rebalancing operations.



(a) Reinforcement learning policy

(b) LFU policy



(c) No policy

Figure 6.2.: Tier usage over time for each policy.

Synthetic Distribution - Latency Measurements

In Figure 6.3, we show an overview of latency measurements for each run. The graphic is again divided into results for the three groups "Seldom", "Occasional", and "Often" additionally split horizontally, with the upper plot representing the *read* latency and the lower plot the *write* latency. For each group, multiple objects of different predefined sizes are sampled and all values are shown in the form of a box plot, with the highlighted line denoting the median value. In describing the observed latency, we will first focus on the *read* latency for each size and later on discuss the *write* latency.

In Figure 6.3a, we show the achieved latency of the reinforcement learning policy. Focusing on values of size 1 GB, we can see that the run exhibits the best latency in the "Often" group and "Occasional" and "Seldom" achieving almost the same latency. For one, this shows that the policy is well adapting to the frequency of access each object has been experiencing. Furthermore, we also see sampling bias, as we will show in Figure 6.5a. The "Occasional" group stores large objects on tier *FASTEST* and *SLOW*, apparently objects stored on the *SLOW* tier have been sampled here.

For objects of size 64 KB and 256 KB, we see that the latency of all groups is rather similar, only a minor speed up is achieved in the "Often" group with the median latency of 256 KB accesses being slightly lower than in the other groups. Additionally, we see an unusual large spike on a single access of a 64 KB object. We assume that this is an outlier caused by some other random influence, as the median is almost at the bottom of the box, and therefore we will omit it from our analysis.

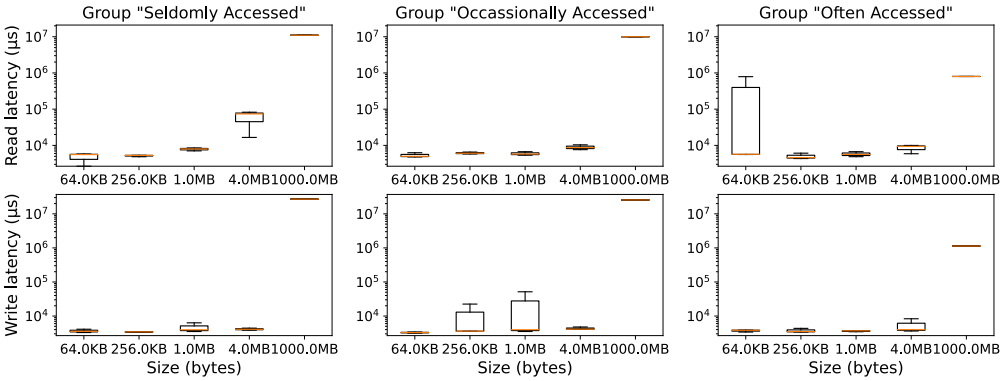
The difference between them become more pronounced on larger objects, such as 1 MB and 4 MB. This increase in difference can be attributed to the increased access throughput between layers, which becomes more influential in larger objects. Deducing from the spread of latency, we know that 4 MB objects of the "Seldom" group are stored on tier *SLOW* and *FAST*, "Occasional" objects on *FAST* and "Often" objects on *FAST* and *FASTEST*.[5]

Observing the same groups under the LFU policy in Figure 6.3b, we see for small objects a similar situation with latency being low and relatively close over all groups, with a slight advantage of the "Often" group over the others. However, larger objects of size 1 MB and 4 MB do not have a discernible difference in access latency when comparing each group to another. When analyzing the access latency for objects of size 1 GB, we see that no meaningful difference exist between access latencies besides those induced by fluctuations in the system. This behavior is expected by the LFU policy, as large objects are preferably demoted.

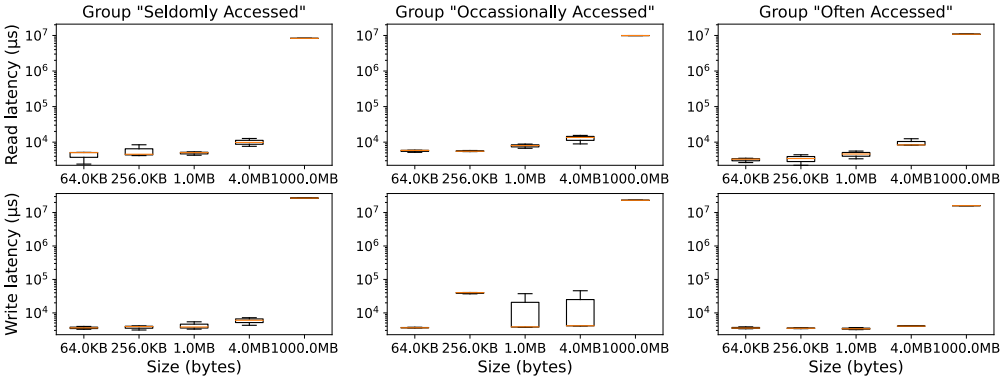
Lastly, we can compare the *read* latencies to the baseline behavior, shown in Figure 6.3c. Notable is that, both policies bring an advantage compared to pure random distribution. Due to the implementation of the benchmark, which fills the storage stack *first* with "Seldom" values, the "Often" group is predisposed to be stored on larger tiers such as *SLOW* or *FAST*, as can be seen in the on average higher latencies of the "Often" group in this run. Both policy migrate correct objects from lower to higher tiers and choose fitting objects to be demoted. Notably, LFU performs worse than the reinforcement learning policy with *read* access latency differences being more noticeable in the latter.

When considering the *write* latencies, we see a similar view to the just described *read* case. Importantly, due to the write optimized nature of the B^{ε}-tree small insertions, such as for small

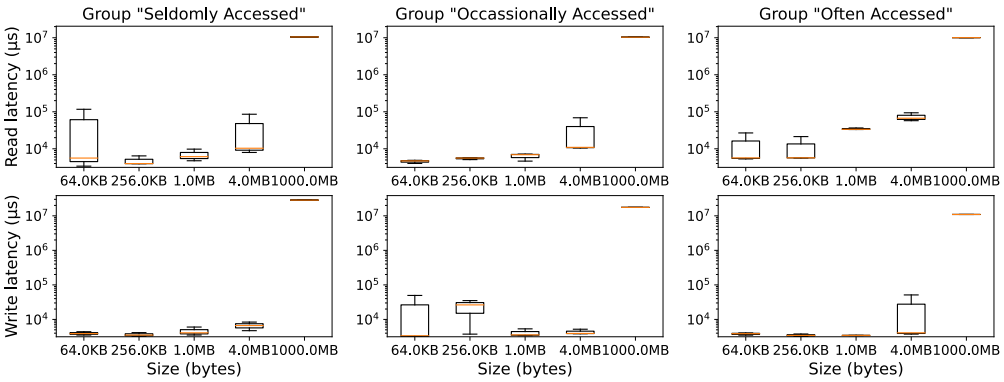
[5] We back this up with our observations in Section 6.2.4



(a) Reinforcement learning policy



(b) LFU policy



(c) No policy

Figure 6.3.: Measured latency after conditioning with each policy.

objects, are inserted in the root and therefore we expect the latency to be quite similar between all groups. Only when observing objects of size 4 MB and larger we may see a perceptible difference. Also, since the tree is not deamortized, some insertion operations may require a longer time than others, as rebalancing operations are occurring based on the state of the tree.

Starting off with Figure 6.3a again, we see that they perform almost identically except for operations likely including rebalancing as mentioned before and a significant difference in write performance on objects of size 1 GB. Here, the differently chosen storage tiers are of more significance as this size widely oversteps the capabilities of internal nodes to store all new key value updates. This results in a distribution similar to the read latency observed in all three groups. The observations we made there can also be applied here, with the reinforcement learning policy adapting well to the defined pattern.

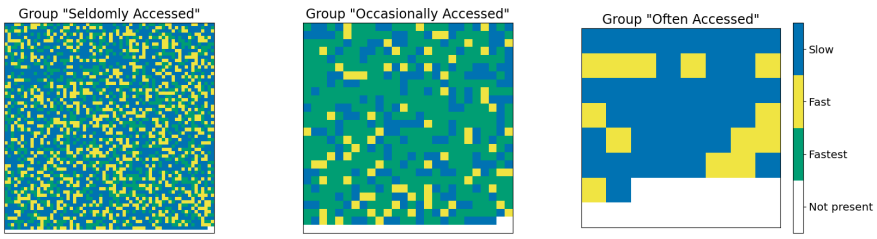
In Figure 6.3b, we show the results for the LFU run. Here, two characteristics are interesting in the observation. First, the results are again similar to the ones observed in the *read* latencies, with small object insertions perform equally good on object size smaller or equal to 4 MB. This is likely due to the optimizations of the B^{ε}-tree and the case that internal nodes depending on their children are likely stored on the highest tier *FASTEST*. Second, there is a range of objects in the "Occasional" group, starting from 256 KB to 4 MB which likely experience a high number of rebalancing operations. Besides these oddities, the *write* latencies are as expected with a slight improvement of objects of size 4 MB in the "Often" group.

Considering the comparison to the default behavior seen in Figure 6.3c, we see not much difference to the optimized behavior of the migration policies. Solely for large objects they are considerably better than no policy at all. This leads us to the conclusion that in a B^{ε}-tree storage stack objects are worth to be migrated if they surpass a threshold size or need fast *read* access. The write optimization of the B^{ε}-tree is sufficient even for key groups of size equal to the node size to effectively minimize the *write* latency.

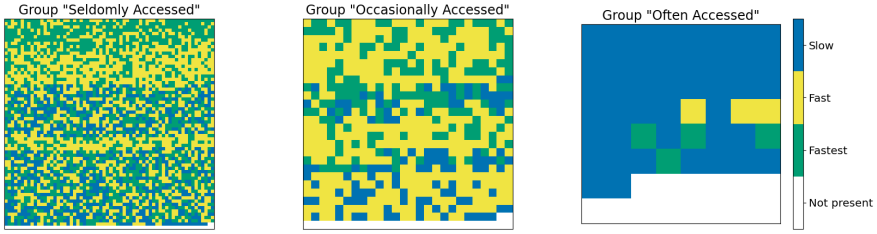
Synthetic Distribution - Detailed Object Tier Distribution

In Figure 6.4 (earliest recorded complete state) and Figure 6.5 (last recorded state), we show a detailed view of the existing objects in the synthetic distribution for the LFU and reinforcement learning policy. Each group is represented by a grid, with each cell being equivalent to an object. The size of objects is depending on their row, with the bottom row being consisting of objects of the maximum size of 1 GB and the top one of 64 KB objects. In Figure 6.4, we show an early stage of the migration process. While in Figure 6.5, the last performed step, before the measurements have been performed, can be seen. In the digital appendix to this thesis, a video can be found for each run showing the movement of objects over time.

Starting from the "Seldom" group, we see that in Figure 6.5a a majority of objects are stored in the lowest *SLOW* tier. Some objects are located in the *FAST* and no objects are in the *FASTEST* tier. This is a well-fitting distribution to the defined access patterns of objects and is close to the expected result. The different sizes of objects do not seem to take a great influence in this group on the choice of where to start which object, as we observe a relatively uniform distribution of objects stored in the *FAST* tier. This leads to the effect that even some larger objects which are never accessed are stored in a high tier, occupying a relatively low latency storage range. Therefore, small objects have a relatively high access latency, which might be undesirable as we

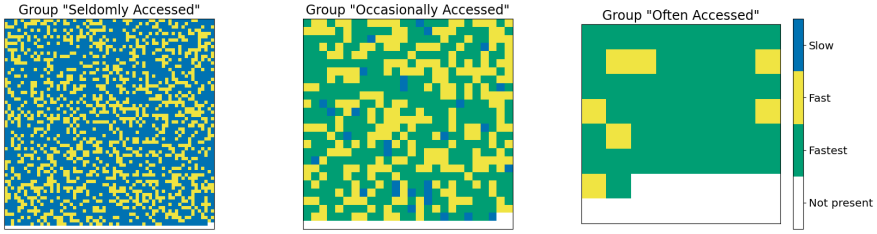


(a) Reinforcement learning policy

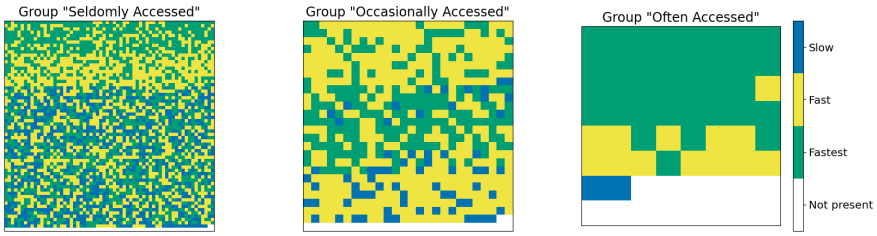


(b) LFU policy

Figure 6.4.: Detailed object overview of each policy in the early stages of the benchmark. Objects are portrayed as grid cells. Groups from left to right are “Seldom”, “Occasional” and “Often”.



(a) Reinforcement learning policy



(b) LFU policy

Figure 6.5.: Detailed object overview of each policy at the end of the benchmark. Objects are portrayed as grid cells. Groups from left to right are “Seldom”, “Occasional” and “Often”.

want to avoid high latency of these when compared to larger objects which can take advantage of a higher streaming throughput of devices such as HDD (Ge et al., 2022).

In Figure 6.5b, we show the same group under the LFU policy. As already seen by the lower mean average tier in Figure 6.1b, more objects are dispersed onto different tiers. We can observe

that smaller objects tend to be stored on the *FASTEST* and *FAST* tiers with larger objects also stored on *SLOW*. This is to be expected, as the size buckets used in this policy should produce this effect. Although, the number of objects stored on *FASTEST* is quite high and the average mean compared to the groups shows that not too much difference exists between the groups. The LFU policy does not discern small objects which are part of the "Seldom" group well enough.

The "Occasional" group in Figure 6.5a is similarly uniformly distributed as the "Seldom" group is. We can see that a large part of objects is stored in *FASTEST*, followed by *FAST*, and lastly a few objects in *SLOW* again independent of their size.

In Figure 6.5b an interesting observation can be made. First, fewer objects can be found in *SLOW* tier with small objects being in majority on the *FAST* tier and medium-sized objects on *FASTEST*. Importantly, large objects are as desired on either *SLOW* or *FAST* showing that for this group placement of LFU turned out much better than in the "Seldom" group.

Both policies identify objects in the "Often" group well, with objects in Figure 6.5a being distributed independently of size and Figure 6.5b distributing small objects on *FASTEST*, medium-sized on *FAST*, and large on *SLOW*. Especially, when comparing this to previous groups the LFU policy shows inefficiency which results in non-optimal distributions.

Snapshot Distribution - Latency Measurements

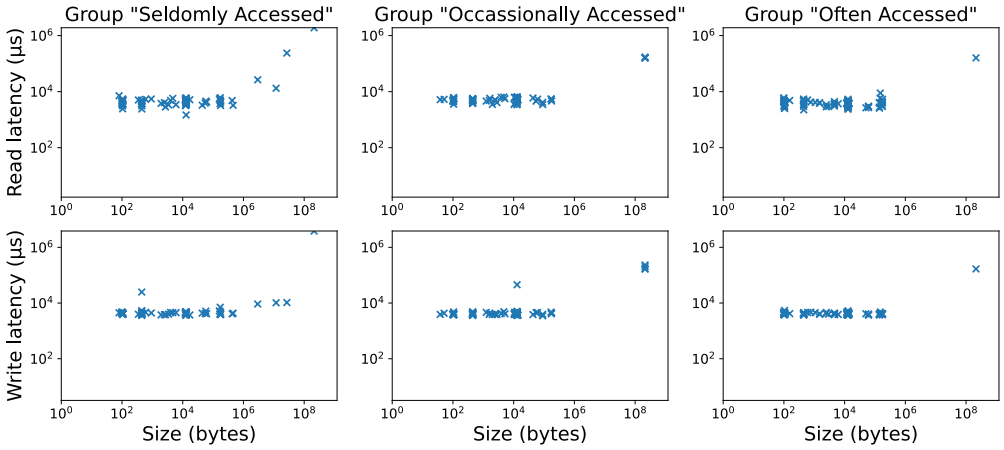
The snapshot distribution shows in general a much smaller object size. Sampled objects range from 10^1 bytes (10 bytes) to about 10^8 (100 MB). The data points in Figure 6.6 are chosen to be shown as a scatter chart rather than a box plot, as with the real distribution of objects sampling based on an exact size would have introduced a bias as not all ranges of object sizes had sufficient duplicate object sizes.

We again start with the *read* latency and show the unique characteristics of each run, and then compare the *write* latencies. Additionally, we compare the results of this distribution to the ones discussed in Section 6.2.4.

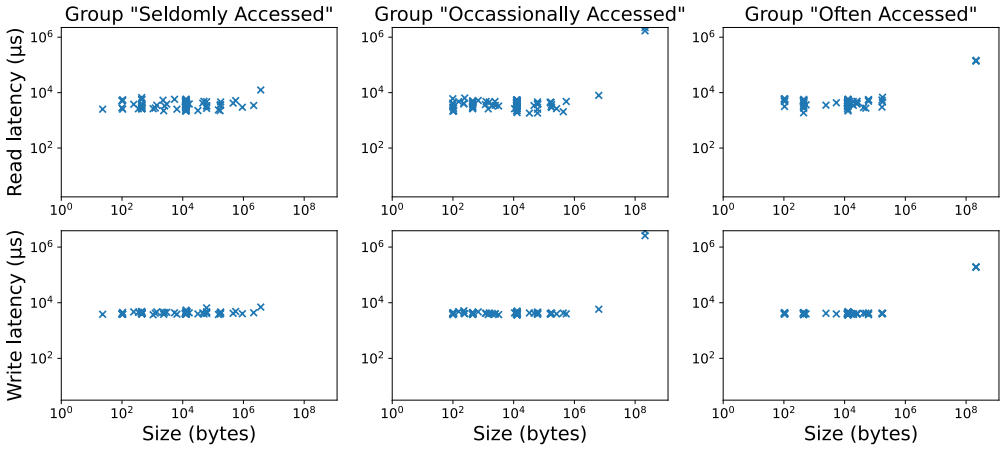
In Figure 6.6a, we see that a majority of accesses are requiring the same amount of time. This is due to the size of nodes is set at 4 MB, so each object smaller than 4 MB will still require the whole node to be read when accessing a singular object. Naturally, only when increasing the size beyond $4 \cdot 10^6$ do we see a difference in the charts. For the "Seldom" group this increase is steep with an object of size 10^8 requiring $> 10^6 \mu s$. In more often accessed groups "Occasional" and "Often" this operation takes about $10^5 \mu s$. This shows that just as in the synthetic distribution the reinforcement learning policy determines a well-fitting placement for the used objects.

Similarly well-performing is the LFU policy in Figure 6.6b. Notably, large objects such as the 100 MB objects in group "Occasional" are stored in lower tiers compared to the reinforcement learning policy. This reflects again, the tendency to store large objects lower than small objects of the same frequency. Otherwise, they perform almost identically.

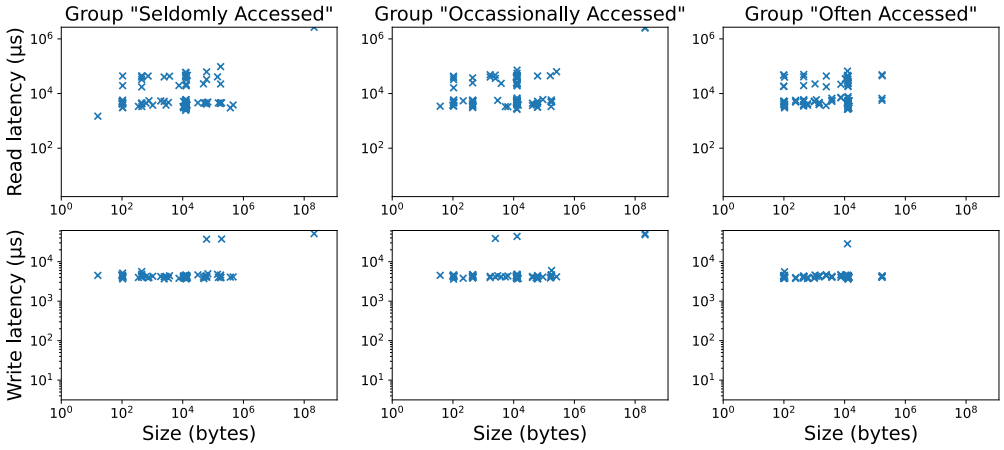
When comparing both policies to a run without any migration occurring we observe that *read* latency values are much more spread by default with the lowest values, akin to the runs with active policies, at around 10^4, and the highest values at just below 10^5. This shows that also for the snapshot distribution, both policies bring an improvement in performance when observing the *read* latency.



(a) Reinforcement learning policy



(b) LFU policy



(c) No policy

Figure 6.6.: Measured latency of the snapshot distribution after conditioning with each policy.

On analyzing the *write* latency of each group we will identify a familiar pattern. Just as argued in Section 6.2.4 the optimizations of the B^{ε}-tree equalize for small values insertion times with all objects smaller than 4 MB achieving a time of just below 10^4. To be named is the decrease in the severity of incrementation of *write* latencies when looking at Figure 6.6a of objects in the "Seldom" group.

Furthermore, a few rebalancing operations can be seen in Figure 6.6c, but aside from these even in the purely random distribution we can see that insertions are performing perfectly fine for small objects.

Snapshot Distribution - Write Behavior

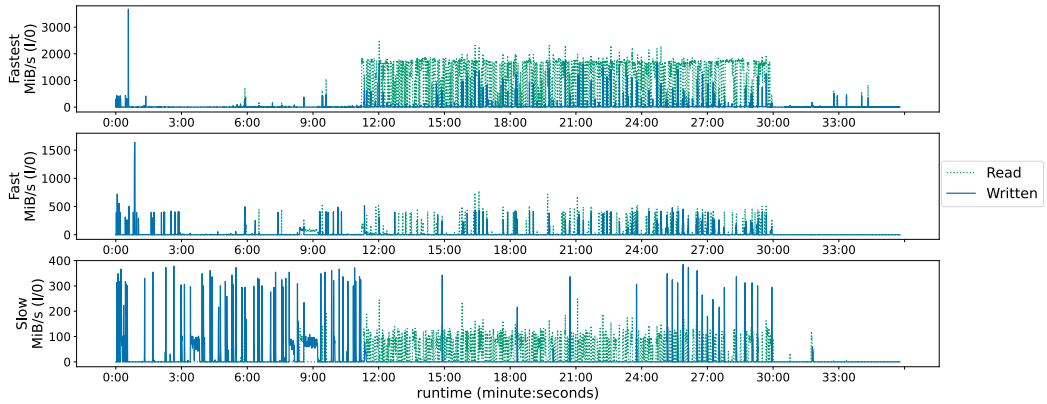
Lastly, we will have a look at the number of migrations and I/O pressure the policies in the snapshot runs are applying to the storage stack. For this, we will investigate the I/O metrics produced by the SPL. All traces are visualized as line charts in Figure 6.7. Three charts are given for each run with the top representing the *FASTEST* tier, the middle *FAST* tier, and the bottom the *SLOW* tier.

As base load Figure 6.7c is shown, where no migration policy was applied. Evident is the division into three distinct phases as we explained in Section 6.2.3. First, the initialization takes roughly 11 minutes in total. Second, the conditioning phase which finishes at the 30 minute mark. Third and last, the measurement phase lasts around 5 minutes. The first and last phase is largely indifferent between each run, therefore we will concentrate on the conditioning phase. In the default behavior, with random initialization, each of the three tiers is used with *SLOW* with constant use of about 50 - 100 MiB per second. The *FAST* tier is more momentarily used with a base throughput of 100 MiB/s and regular peaks of about 400 MiB/s. The *FASTEST* tier is most extensively used with 500 MiB/s more or less constantly. Naturally, since the conditioning phase only performs read operations, no writes are registered in this phase.

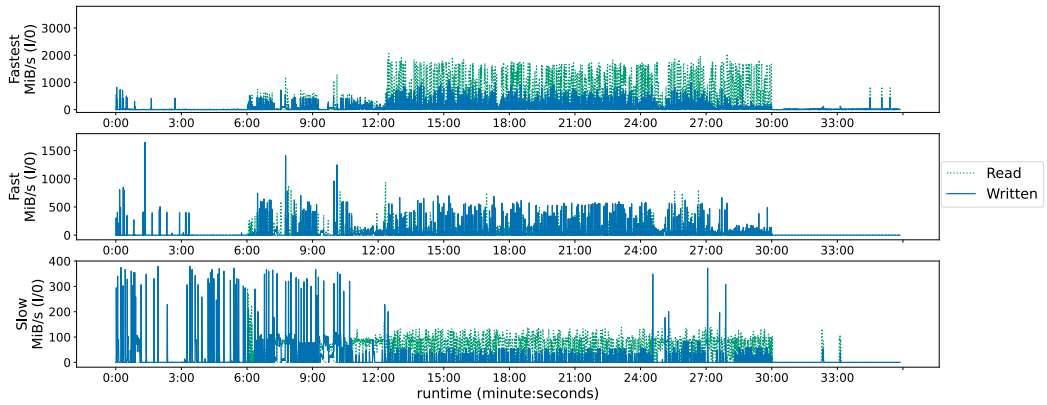
Comparing this base load to Figure 6.7a a strong increase in the usage of the *FASTEST* tier can be seen. The read average load totals at around 1.5 GiB/s showing that the throughput is strongly increased when performing read operations. A possible reason is simply that a greater number of objects is located on this tier so that in the metric interval (which is 500 ms) more data is fetched pushing the throughput number. Even for the bottom tier *SLOW* an increased throughput can be noticed, which could correlate to fewer search operations occurring relative to the data fetched, as the storage medium (an HDD) is highly susceptible to performance loss on such use. Also in the bottom tier, we see that most write operations are occurring in the last 5 minutes of conditioning. This means that most data is first migrated upwards and later data is moved downwards, or that simply not many objects were migrated from the bottom tier at first. When analyzing the use of the *FAST* tier, a reduction in throughput can be seen. The aforementioned spikes still exist but the base load is reduced to almost 0 MiB/s. We assume that with the increased use of the *FASTEST* tier, many active objects are moved upwards from the *FAST* tier, explaining the strong increase in throughput.

Overall the reinforcement learning policy introduces a slightly increased write use but enables increased read use of the system.

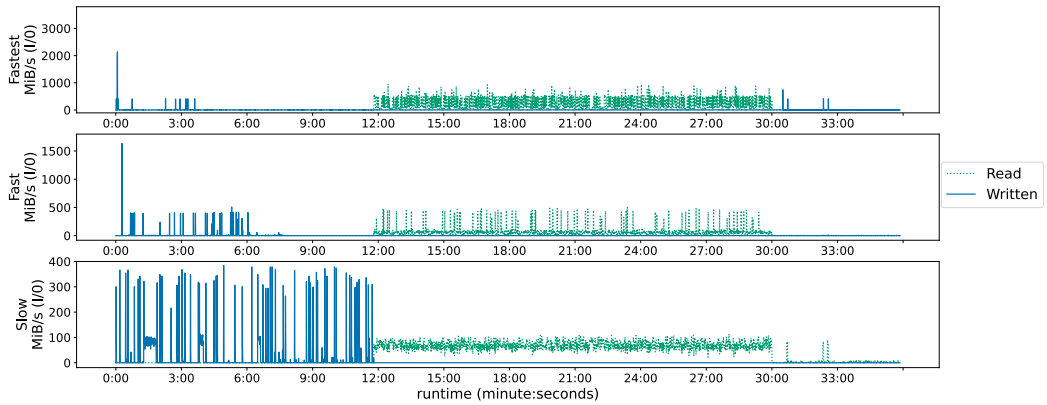
Examining the LFU policy, we notice a strong increase in write operations in all tiers as well as an increase in reading operations in all tiers. This is similar to observations made by Zhang et al. (2022). While the LFU policy achieves a similar read throughput as the reinforcement learning



(a) Reinforcement learning policy



(b) LFU policy



(c) No policy

Figure 6.7.: Measured latency of the snapshot distribution after conditioning with each policy.

policy, it likely is induced by the constantly occurring migration operations. From the perspective of an application, this would not result in a strong increase of read performance as in the reinforcement learning policy run. Notable is, the use of the *FAST* tier is increased compared to the reinforcement learning policy.

6.3. Specific Use-Case Benchmark

Next, we discuss two short benchmarks we added to cover more scenarios that are common in HPC and other domains. First, we aim to test how well the policies act on a dynamic set of objects. For this we benchmark the policies in a checkpoint-like burst pattern which is intended to behave like a scientific application, performing no I/O for a set amount of time, followed by high-throughput large serial writes and repeating (Kuo et al., 2014). This pattern is common in HPC and technologies like burst buffers are sometimes used to speed-up I/O and reduce total runtime (Wang et al., 2021, 2004).

Second, we experiment if the migration on a node basis can bring a noticeable difference in the current state. For this, we created a benchmark application that performs a random read-and-write pattern on a single object.

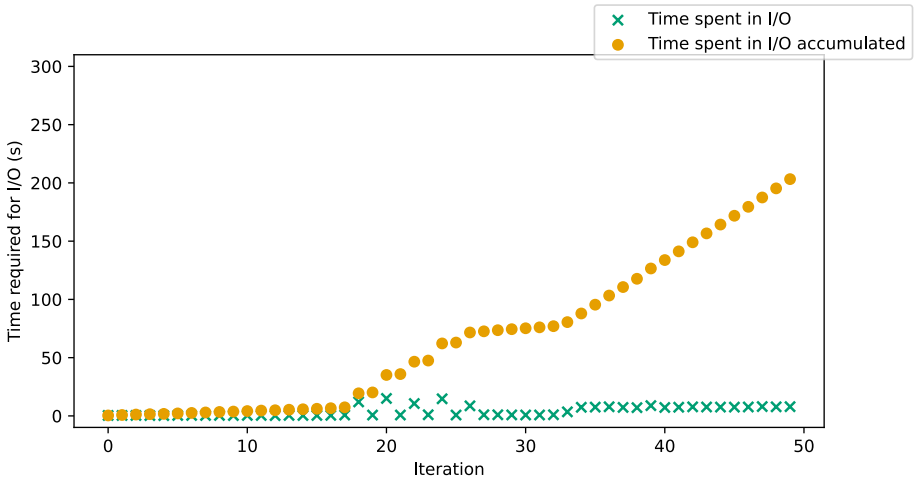
6.3.1. Checkpoints

This benchmark is run with all three policy options. The base loop works as follows. After the storage initialization is finished a loop is entered which writes 5 objects in each iteration of a total size of 1 GiB. After this is finished the application waits 5 seconds to simulate some computation workload, with a small random variance of maximum 400 milliseconds. In total we write 50 generations of snapshots surmounting in 50 GiB of total data written. The setup is the same as in the "Mimic Workload" case with 8 GiB *FASTEST*, 16 GiB *FAST*, and 100 GiB *SLOW* storage. Additionally, to simulate how the storage stack is envisioned to be used, we do not actively check the available size on each tier before writing, but instead define a fallback configuration that if the *FASTEST* tier does not have enough storage space available, we use the *FAST* tier, then the *SLOW* tier.

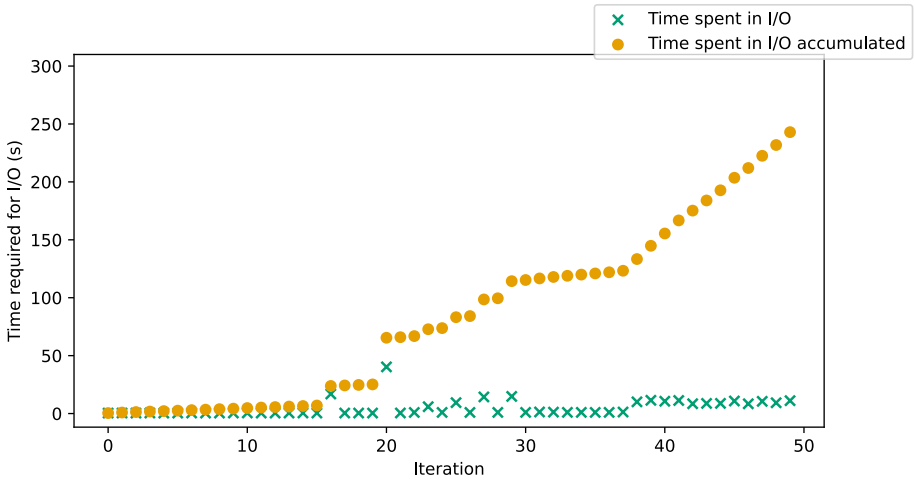
For each iteration we store the time required until all I/O operations are finished. We are interested in the observation of individual write delays as well as the total runtime of the benchmark.

In Figure 6.8 we show the time spent in I/O operations for each individual iteration and the accumulated total time in a scatter chart. We first discuss the general observations of the accumulated time compared between all three runs, followed by individual events in each run and challenges in relation to the current implementation of objects.

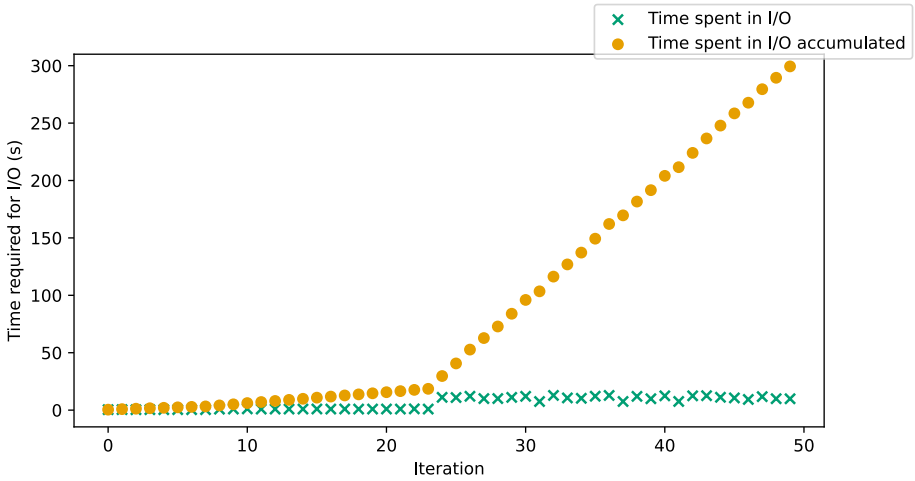
When we compare all policies we see a noticeable difference in total runtime. The "No policy", see Figure 6.8c, exhibits the longest total runtime with just above 300 seconds. Shorter is the LFU policy run, see Figure 6.8b, which for the same number of iterations around 250 seconds. The shortest runtime was required by the reinforcement learning policy run, see Figure 6.8a, which again reduces the time by about 50 seconds to 200 seconds in total. This distribution of performance is familiar from the previous benchmarks and confirms that even in cases dynamic sets of objects our observation of quality between the policies holds.



(a) Reinforcement learning policy



(b) LFU policy



(c) No policy

Figure 6.8.: Time spent in I/O in the *Checkpoints* benchmark

When comparing individual iterations we notice a striking difference between the policies which perform migrations and the runs without migration occurring. While the maximum time spent in I/O of each iteration is around 10 seconds in Figure 6.8c, the maximum time in Figure 6.8a is 20 seconds, and even 45 seconds in Figure 6.8b. This is due to the additional traffic produced by the migration policies, even though this does result in a total runtime reduction, it has to be kept in mind if predictability of duration between iterations are of concern to the user. To alleviate this side effects we induce in the system we propose a possible solution here. We may reduce the amount of migrations happening eagerly by introducing a delayed migration of large objects. Feasible would be that we monitor the usage of the system and initiate migrations whenever tiers are idling with no I/O occurring for some amount of time. This is a rather basic metric to detect possible usage but should suffice for the targeted purpose. A separate thread may be used for this, so that migrations initiated by the user can use this interface too.

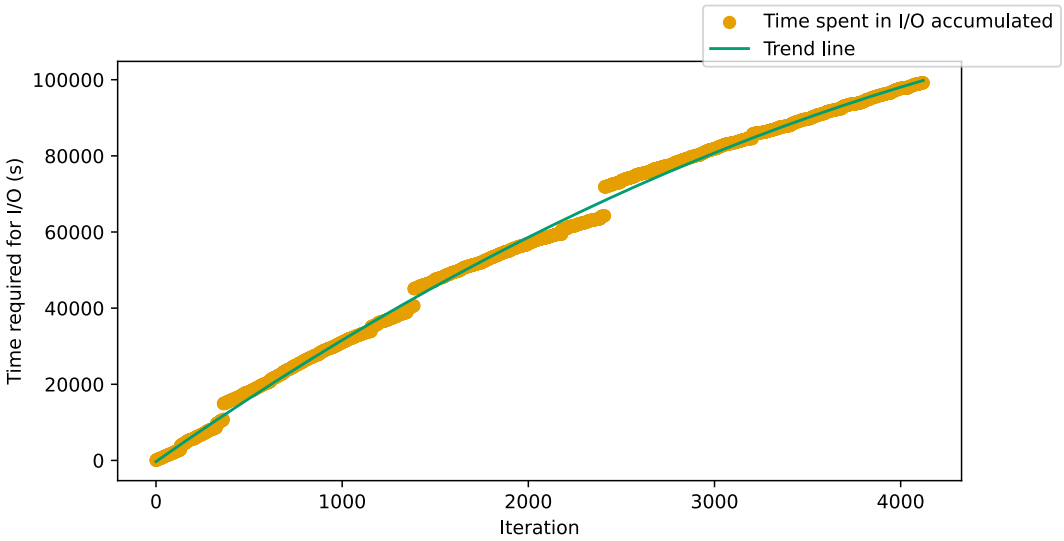
Also, the timing of migration iterations may amplify this temporary increase in required I/O time as the for each iteration of the application takes about five seconds, and the configured iteration time of the migration policies has been set to ten seconds. So every two iterations we expect an increase in I/O time due to this overlap. We see this pattern in both migration runs in Figure 6.8a from iteration 18 to 27 and in Figure 6.8b from iteration 23 to 29. In the detection of storage stack activity we may also be able to decouple from this pattern, possibly reducing I/O time further.

Lastly, another major impact on the current I/O time is the detection of objects and their *correct* storage tier. At the moment we only have the indication given by the object store, which we know does not necessarily represent the actual storage tier the object is stored on. If we define a fallback policy which is *not* the identity, as we have done in this benchmark, we lose accuracy of this indication. In the end of this benchmark, both policies end up diverted by this fact as migration stops and we get the linear increase of accumulated I/O time just as in Figure 6.8c. Fixing this behavior requires some measure to track the actual storage of objects given by the DMU. The design of such an interface is currently left open as many implications have to be considered to achieve a simple enough design which does not introduce unreasonable amount of cluttering and handling logic from the DMU to higher layers like the Dataset and Object Store.

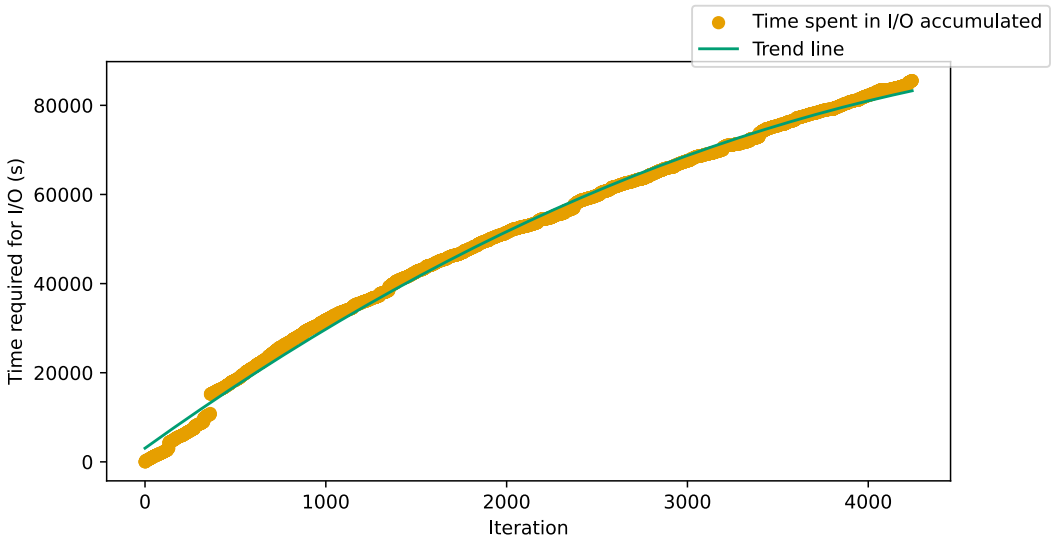
6.3.2. Node Promotion

As the last experiment, we chose a case focused on node promotion. In this, we create an object of size 20 GiB in the bottom tier *SLOW* and select 1024 locations in this file of maximum length 12 MiB and expected size of 6 MiB. These locations are read from *or* written to in sequence, starting anew from the first location when the last has been processed. One run that we measure has performed only *read* operations, another only *write* operations. The total runtime of this benchmark is set to be 300 seconds.

To accommodate node promotion we use the LFU policy with the configuration shown in Listing 6.3. The major changes for this benchmark are the modification of the `grace_period` so that migration begins with the initialization of the storage stack and the change of the `mode` option from “Object” to “Node” as explained in Section 4.8.1. Minor tweaks have been made to the `update_period`, `migration_threshold`, and `promote_size` to better fit this benchmark. Raising again the problem of expert knowledge required for a well-behaving static policy.



(a) Series of *reading* operations



(b) Series of *writing* operations

Figure 6.9.: Time spent in I/O in the *Node Promotion* benchmark

In Figure 6.9, we visualize these runs with two plots showing the accumulated I/O time over the complete duration. We make two important observations from these scenarios. First, the individual I/O time per iteration decreases as we can see by the overlaid trend lines, both of which flatten to the end of the run. This confirms a basic assumption that we identify the correct nodes and the promotion of nodes actually reduces I/O time for future operations in both read and write cases.

Second, we see singular operations of much higher runtime than the average operation. This can be seen in Figure 6.9a at iterations 400, 1500, and 2500. At these points, we perform abnormally more write operations by evicting parts of the cache and migrating them, due to the presence of promotion hints, to a higher storage tier.

When performing *write* operations in Figure 6.9b this becomes less of a concern. Importantly, we still have a single operation requiring above-average runtime at iteration 400 but all others are largely unnoticeable. This is due to the “zero-cost” promotion in combination with CoW

```
"Lfu": {
  "grace_period": {
    "secs": 0,
    "nanos": 0
  },
  "migration_threshold": 0.8,
  "update_period": {
    "secs": 20,
    "nanos": 0
  },
  "policy_config": {
    "promote_num": 99999,
    "promote_size": 52428,
    "path_state": "tier_state.jsonl",
    "path_delta": null,
    "mode": "Node"
  }
}
```

Listing 6.3: Configuration of the LFU policy in the *Node Promotion* benchmark.

which we can take advantage of here. All nodes will be written completely anyway and with the promotion hints, we simply redirect allocations to a higher storage tier. The result is a smoother and faster migration process.

6.4. Summary

We designed three distinct experiments with multiple variants in this chapter and utilized them to discover aspects of the policies and the storage stack. To evaluate the performance and quality of the migration policies we have inspected multiple metrics too gain a holistic image of the storage stack and policies under investigation. Some of these important metrics are latency, tier utilization, runtime, and object distribution.

We found that both newly implemented policies deliver an advantage to the status quo, although the reinforcement policies matches or outperforms the LFU policy in most regards. Furthermore, we observed multiple interference factors introduced by the migration of objects *and* nodes. In addition, we found that using the CoW characteristic can reduce these interference when performing write operations. Also, when using a B^{ε}-tree as underlying data structure the migration of small objects for workflow consisting in majority of write operations is only worth considering if this object observes a large number of write operations for any data small enough to be stored in internal leaves our construction of multi-tier trees is in almost all cases providing a similar speedup.

Chapter 7.

Conclusion

In this thesis, we analyzed the combination of data migration techniques with a hierarchical CoW B$^{\varepsilon}$-tree based storage stack.

We surveyed and categorized existing approaches, and created constraints that a model for a general-purpose storage stack has to fulfill. These include the generality of the approach, application or system-specific approaches may not be considered as this would require great effort for any user to gain an optimized storage stack. This includes approaches which have shown for example great gain in performance when prefetching data like the model by Cherubini et al. (2017).

We have extended the *Haura* storage stack by an exchangeable policy layer, which receives standardized messages and may manipulate data with existing interfaces or emit storage hints to the central storage organizer, the DMU. In the design of these messages and action interfaces, we found several challenges of which some remain unsolved due to high effort, which constrained the choice of migration policies in this thesis. Of these challenges, the multi-tier tree architecture (with its uncertainty in final storage choice) and out-of-place tree node manipulation remain to be solved to remove all constraints which we have created.

We have chosen two policies to implement in this newly created migration interface. First, a "classic" LFU based policy that tracks nodes and object access frequencies and migrates them based on the environment of each tier. Second, a reinforcement learning policy originally proposed by Vengerov (2008) has been implemented.

To evaluate these policies we created three scenarios with multiple variants and compared the quality, effectiveness of each policy, and total system load compared to the default behavior. We found that the application of a policy regardless of choice improves the latency of the system, with the reinforcement learning policy generally outperforming the LFU policy. This confirms findings by papers such as Zhang et al. (2022). Furthermore, these findings underline the need for a better intelligent data management approach in future storage systems as much potential is still present, especially with the materialization of ever more complex storage architectures.

We also made observations regarding the combination of write-optimized data structures, CoW, and migration techniques guiding the future development of policies and concepts in the *Haura* storage stack.

Future Work

In this thesis, we have encountered multiple obstructions, when implementing and designing migration policies. We have introduced workarounds, restrictions, or trade-offs to achieve a working solution, but for long-term usage and more extensive experimentation, there are multiple problems, which need to be addressed.

Cache Uncertainty With the addition of space accounting, we were able to introduce some safety features in the storage stack but not all problems can or may be solved with this. When a large cache is used, we encounter uncertainty in our space information, as we cannot predict which tier new nodes will be stored at. To fix this, we may allocate nodes preemptively which may lead to problems if the node is not already of the maximum size. Or we keep a ledger of cache space accounting information that can predict the likely distribution[1] and combine this knowledge with the actual space accounting ledger.

Multi-tier Objects When migrating, we assume that objects are of a single tier, but with node promotion, we destroy this restriction and no longer adhere to a basic assumption. Now is the question, how do we account for multi-tier objects? We may define ranges on objects for which we know the storage location, but updating these data structures becomes a lot of work, especially since the DMU decides where to store the data and objects are never notified. This data structure can for example be a map that the DMU can gain access to. But this breaks essential layer isolation as the DMU now needs to inspect the internals of the nodes it manages. We may want to think about an additional layer in between the datasets, object stores, B^{ε}-tree, and the DMU to deal with these additional responsibilities.

Node Demotion Due to the design of the B^{ε}-tree as a monolithic structure, it is currently not reliable to demote certain nodes as the node migrations are currently performed via a hinting mechanic. We initially experimented with relaxing this constraint but found a more major redesign of the architecture has to be done to avoid using extensive locking structure at any given node. Discussion about this topic has already started in the working group but no definitive design has prevailed yet.

Robustness We have in the process of this thesis added a significant number of tests to the storage stack as we found that, especially in constellations where small limited space is available, operations are not reliable and on some occasions would return errors or create deadlocks, which were at the first glance out of place.

Further Experimentation with Policies Since the framework enables an easy exchange of policies is now formulated and tested, we may experiment with more policies and see how well they perform. Interesting here are lifetime prediction policies as done by Thomas et al. (2021); Monjalet and Leibovici (2019) and other more complex algorithms see Cheng et al. (2021); Shin et al. (2019); Ren et al. (2019); Shi et al. (2020); Yang et al. (2017).

Prefetching and Prediction As already mentioned, data prefetching and access prediction is a major axis of optimization with which we could hide most high latencies introduced by mass storage. Several approaches exist for this and based on the surveyed paper some kind of workflow classification could bring interesting results. Although, this topic is

[1] Or targeted distribution via the storage preference.

quite experimental and likely to produce domain-specific solutions, creating an approach that we could apply to a number of scenarios is feasible. The combination with online learning as done with the current policies would be an interesting research topic.

Bibliography

Bender, M. A., Farach-Colton, M., Jannen, W., Johnson, R., Kuszmaul, B. C., Porter, D. E., Yuan, J., and Zhan, Y. (2015). An introduction to B^{ε}-trees and write-optimization. *;login: magazine*, 40(5). (Cited on pages 11, 12, 13, 83, and 85)

Bishop, C. M. (1994). Neural networks and their applications. *Review of scientific instruments*, 65(6):1803–1832. (Cited on page 24)

Brodal, G. S. and Fagerberg, R. (2003). Lower Bounds for External Memory Dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, page 546–554, USA. Society for Industrial and Applied Mathematics. (Cited on pages 11, 13, and 83)

Cheng, Y., Zhang, F., Hu, G., Wang, Y., Yang, H., Zhang, G., and Cheng, Z. (2021). Block Popularity Prediction for Multimedia Storage Systems Using Spatial-Temporal-Sequential Neural Networks. In *Proceedings of the 29th ACM International Conference on Multimedia*, MM '21, page 3390–3398, New York, NY, USA. Association for Computing Machinery. (Cited on pages 23, 28, 37, and 76)

Cherubini, G., Kim, Y., Lantz, M., and Venkatesan, V. (2017). Data Prefetching for Large Tiered Storage Systems. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 823–828. (Cited on pages 23, 28, and 75)

Chrobak, M. and Noga, J. (1999). LRU is better than FIFO. *Algorithmica*, 23(2):180–185. (Cited on page 20)

Corbato, F. J. (1968). A paging experiment with the multics system. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC. (Cited on page 21)

Corbet, J. (2021). The multi-generational LRU. *Linux Weekly News*. (Cited on page 20)

Davis, Michael C., Bahyl, Vladímir, Cancio, Germán, Cano, Eric, Leduc, Julien, and Murray, Steven (2019). CERN Tape Archive - from development to production deployment. *EPJ Web Conf.*, 214:04015. (Cited on page 14)

Ge, X., Cao, Z., Du, D. H. C., Ganesan, P., and Hahn, D. (2022). HintStor: A Framework to Study I/O Hints in Heterogeneous Storage. *ACM Trans. Storage*, 18(2). (Cited on pages 25, 28, 29, 31, 48, 51, 54, and 65)

Höppner, T. (2021). Design and Implementation of an Object Store with Tiered Storage. (Cited on pages 15, 40, and 83)

Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285. (Cited on pages 23 and 24)

Kuhn, M. (2017). JULEA: A Flexible Storage Framework for HPC. In Kunkel, J. M., Yokota, R., Taufer, M., and Shalf, J., editors, *High Performance Computing*, pages 712–723, Cham. Springer International Publishing. (Cited on page 37)

Kuhn, M. (2021). Lecture "Data Reduction" in Parallel Storage Systems. (Cited on pages 8 and 83)

Kuo, C.-S., Shah, A., Nomura, A., Matsuoka, S., and Wolf, F. (2014). How file access patterns influence interference among cluster applications. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 185–193. (Cited on page 70)

Laizet, S. and Lamballais, E. (2009). High-order compact schemes for incompressible flows: A simple and efficient method with quasi-spectral accuracy. *Journal of Computational Physics*, 228(16):5989–6015. (Cited on page 58)

Lüttgau, J., Kuhn, M., Duwe, K., Alforov, Y., Betke, E., Kunkel, J., and Ludwig, T. (2018). Survey of Storage Systems for High-Performance Computing. *Supercomputing Frontiers and Innovations*, 5(1):31–58. (Cited on pages 7, 14, 15, and 85)

Matani, D., Shah, K., and Mitra, A. (2021). An O(1) algorithm for implementing the LFU cache eviction scheme. *arXiv e-prints*, page arXiv:2110.11602. (Cited on pages 27, 31, and 46)

Megiddo, N. and Modha, D. (2004). Outperforming LRU with an adaptive replacement cache algorithm. *Computer*, 37(4):58–65. (Cited on page 20)

Meister, D., Kaiser, J., Brinkmann, A., Cortes, T., Kuhn, M., and Kunkel, J. (2012). A study on data deduplication in HPC storage systems. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. (Cited on page 57)

Meng, X., Wu, C., Li, J., Liang, X., Bin, Y., Guo, M., and Zheng, L. (2014). HFA: A Hint Frequency-based approach to enhance the I/O performance of multi-level cache storage systems. In *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 376–383. IEEE. (Cited on page 25)

Monjalet, F. and Leibovici, T. (2019). Predicting File Lifetimes with Machine Learning. In Weiland, M., Juckeland, G., Alam, S., and Jagode, H., editors, *High Performance Computing*, pages 288–299, Cham. Springer International Publishing. (Cited on pages 23 and 76)

Peterson, Z. N. J. (2002). Data placement for copy-on-write using virtual contiguity. Master's thesis, University of California, Santa Cruz 2002. (Cited on page 18)

Petrank, E. and Rawitz, D. (2002). The Hardness of Cache Conscious Data Placement. *SIGPLAN Not.*, 37(1):101–112. (Cited on page 19)

Puterman, M. L. (1990). Chapter 8 Markov decision processes. In *Stochastic Models*, volume 2 of *Handbooks in Operations Research and Management Science*, pages 331–434. Elsevier. (Cited on page 51)

Ren, J., Chen, X., Tan, Y., Liu, D., Duan, M., Liang, L., and Qiao, L. (2019). Archivist: A machine learning assisted data placement mechanism for hybrid storage systems. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pages 676–679. (Cited on pages 23 and 76)

Rybintsev, V. O. (2020). Optimizing the parameters of the lustre-file-system-based hpc system for reverse time migration. *The Journal of Supercomputing*, 76. (Cited on page 8)

Shi, W., Cheng, P., Zhu, C., and Chen, Z. (2020). An Intelligent Data Placement Strategy for Hierarchical Storage Systems. In *2020 IEEE 6th International Conference on Computer and Communications (ICCC)*, pages 2023–2027. (Cited on pages 23 and 76)

Shin, W., Brumgard, C. D., Xie, B., Vazhkudai, S. S., Ghoshal, D., Oral, S., and Ramakrishnan, L. (2019). Data Jockey: Automatic Data Management for HPC Multi-tiered Storage Systems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 511–522. (Cited on pages 25 and 76)

Thomas, L., Gougeaud, S., Rubini, S., Deniel, P., and Boukhobza, J. (2021). Predicting File Lifetimes for Data Placement in Multi-Tiered Storage Systems for HPC. In *Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*, CHEOPS '21, New York, NY, USA. Association for Computing Machinery. (Cited on pages 8, 9, and 76)

Vengerov, D. (2008). A reinforcement learning framework for online data migration in hierarchical storage systems. *The Journal of Supercomputing*, 43. (Cited on pages 22, 23, 27, 29, 31, 49, 50, 51, 52, and 75)

Wang, C., Mohror, K., and Snir, M. (2021). File System Semantics Requirements of HPC Applications. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '21, page 19–30, New York, NY, USA. Association for Computing Machinery. (Cited on page 70)

Wang, F., Xin, Q., Hong, B., Brandt, S. A., Miller, E., Long, D., and McLarty, T. (2004). File System Workload Analysis for Large Scale Scientific Computing Applications. (Cited on page 70)

Welch, B. and Noer, G. (2013). Optimizing a hybrid SSD/HDD HPC storage system based on file size distributions. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12. (Cited on page 57)

Wiedemann, F. (2018). Modern Storage Stack with Key-Value Store Interface and Snapshots Based on Copy-On-Write B^{ε}-Trees. Master's thesis, University of Hamburg. (Cited on pages 15, 40, and 83)

Woodrow, T. S. (1993). Hierarchical storage management system evaluation. In *Third NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 187–216. NASA. (Cited on page 15)

Xu, Q., Siyamwala, H., Ghosh, M., Suri, T., Awasthi, M., Guz, Z., Shayesteh, A., and Balakrishnan, V. (2015). Performance Analysis of NVMe SSDs and Their Implication on Real World Databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15, New York, NY, USA. Association for Computing Machinery. (Cited on page 7)

Yang, Z., Hoseinzadeh, M., Andrews, A., Mayers, C., Evans, D. T., Bolt, R. T., Bhimani, J., Mi, N., and Swanson, S. (2017). AutoTiering: Automatic data placement manager in multi-tier all-flash datacenter. In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8. (Cited on pages 14, 15, and 76)

Yildiz, O., Zhou, A. C., and Ibrahim, S. (2018). Improving the effectiveness of burst buffers for big data processing in hpc systems with eley. *Future Generation Computer Systems*, 86:308–318. (Cited on page 9)

Zhang, T., Hellander, A., and Toor, S. (2022). Efficient hierarchical storage management empowered by reinforcement learning. *IEEE Transactions on Knowledge and Data Engineering*. (Cited on pages 23, 27, 49, 50, 52, 68, and 75)

Acronyms

ARC . . . . . Adaptive Replacement Cache

CoW . . . . . Copy on Write
CPU . . . . . Central Processing Unit
CSV . . . . . Comma Separated Values

DKRZ . . . . . German Climate Research Center
DML . . . . . Data Management Layer
DMU . . . . . Data Management Unit
DRAM . . . . . Dynamic Random Access Memory

FIFO . . . . . First-In First-Out
FRB . . . . . Fuzzy Rulebase

HDD . . . . . Hard Disk Drive
HPC . . . . . High Performance Computing
HSM . . . . . Hierarchical Storage Management

JSON . . . . . Javascript Object Notation

LANL . . . . . Los Alamos National Laboratory
LFU . . . . . Least Frequently Used
LLNL . . . . . Lawrence Livermore National Laboratory
LRU . . . . . Least Recently Used

MDP . . . . . Markov Decision Process
MFU . . . . . Most Frequently Used
MLC . . . . . Multi Level Cells

NVMe . . . . . Non-volatile Memory Express
NVRAM . . . . . Non-volatile Random Access Memory

OLCF . . . . . .Oak Ridge Leadership Computing Facility

POMDP . . . . . .Partially Observable Markov Decision Process
POSIX . . . . . .Portable Operating System Interface

QLC . . . . . .Quad Level Cells

SLC . . . . . .Single Level Cells
SMDP . . . . . .Semi-Markov Decision Process
SMR . . . . . .Shingled Magnetic Recording
SPL . . . . . .Storage Pool Layer
SSD . . . . . .Solid State Disk

TLC . . . . . .Triple Level Cells

List of Figures

List of Tables

Listings