Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Masterarbeit

Simulation of Storage Tiering and Data Migration

vorgelegt von

Kira Isabel Duwe

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Informatik
Matrikelnummer:	6225091
Erstgutachter:	Prof. Dr. Thomas Ludwig
Zweitgutachter:	Dr. Michael Kuhn
Betreuer:	Dr. Michael Kuhn

Hamburg, 2017-09-18

Abstract

The ever-present gap between the growth of computational power in contrast to the capabilities of storage and network technologies makes I/O the bottleneck of a system. This is especially true for large-scale systems found in HPC. Over the years a number of different storage devices emerged each providing their own advantages and disadvantages. Fast memory elements such as RAM are very powerful but come with high acquisition costs. With limited budgets and the requirement for long-term storage over several decades, a different approach is needed. This led to a hierarchical structuring of different technologies atop of one another.

While tape systems are capable of preserving large amounts of data reliably over 30 years, they are also the most affordable choice for this purpose. They form the bottom layer of the hierarchy, whereas high-throughput and low-latency devices like non-volatile RAM are located at the top. As the upper layers are limited in capacity due to their price, data migration policies are essential for managing the file movement between the different tiers in order to maximise the system's performance. Since data loss and downtime are a concern, these policies have to be evaluated in advance. Simulations of such hierarchical storage systems provide an alternative way of analysing the effects of placement strategies. Although there is consent that a generic simulator of diverse storage systems able to represent complex infrastructures is indispensable, the existing proposals lack a number of features. In this thesis, an emulator for hierarchical storage systems has been designed and implemented supporting a wide range of existing and future hardware as well as a flexible topology model. A second library is conceptualised on top offering a file handling interface to the application layer as well as a set of data migration schemes. Only minor changes are required to run an application on the emulated storage system.

The validation shows a maximum performance of both libraries in the range of 7 to 9 GB per second when executed in RAM. Analysing the impact of the used block size lead to the recommendation to use at least 100 kB in order to maximise the resulting performance.

Acknowledgements

My warmest appreciation goes
to Professor Ludwig for enabling this thesis,
to Michael Kuhn for his kind supervision, helpful advice and the proofreading,
to David and Florian W. for also writing their thesis and sharing (not only) the agony,
to Jakob for his constant but helpful teasing and his encouragement,
to my parents and brother for always supporting me.
I thank my dear little sister for visiting several times a week, for providing me with food and hugs and for being the best sister one can wish for.
I thank Florian S. for his amazing support and imperturbable belief in me.

Contents

1. Introduction

This chapter provides an introduction in the problems faced in today's HPC systems. Furthermore, an overview is given about different approaches to lessen the negative effects. Afterwards, the requirement for an appropriate simulation of hierarchical storage systems as well as viable data migration schemes atop is motivated. Finally, the goals of this thesis and its structure are outlined.

1.1. Motivation

Over the last few decades, societies came to rely more than ever on technological progress. From medical care to food production, to communication systems a variety of fields is heavily influenced by this development. Especially, in the area of scientific research, this does enable the possibility to solve more and more complex problems. Often, the acquired new insights are extremely relevant to determine how to ensure even pure survival e.g. through disaster prevention. But also long term developments such as climate change will affect not only today's but also prospective generations. The according scientific application require the computational powers of supercomputers. The increasing complexity of the processed problems as well as the growth of computation speed due to improved hardware leads to rapidly increasing data sizes. This imposes a serious problem as the development of the storage and network technologies is considerably slower. The result is an increasing gap between the performance of computing and storing devices making *Input and Output (*I/O*)* operations a bottleneck.

In Figure 1.1 the exponential growth of computational speed is depicted in comparison to the development of storage and network capabilities. This illustrates why it is not sufficient to approach this problem by buying enough fast memory to lessen the gap. Even if it was not an overly expensive proposal, the computational power improves so fast that after a short period of time the situation will be the same as before. A more feasible solution is to combine different storage technologies in a hierarchy and use the faster devices to serve as caches for the lower layers. This also enables the long term storage with tape systems on the one hand, while on the other hand providing an acceptable performance to the application layer through the use of *Non Volatile Random*

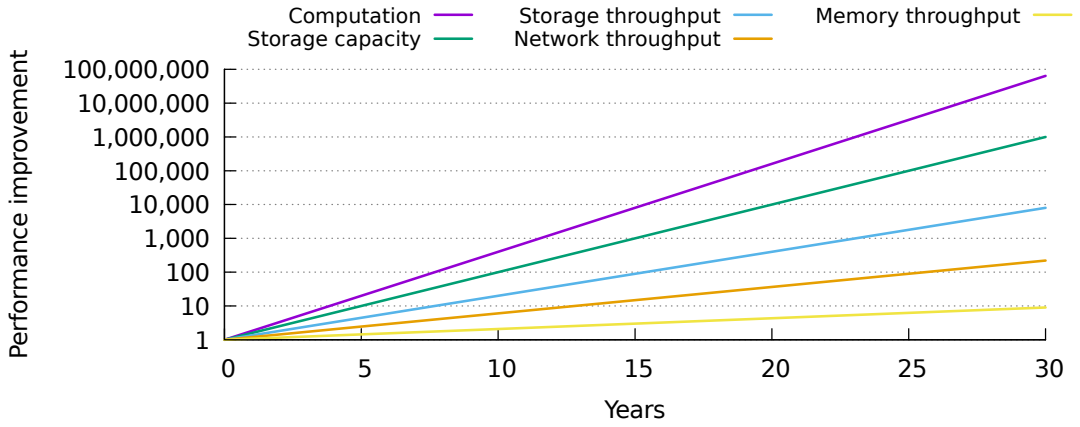

Figure 1.1.: Increasing gap between computational speed, storage capacity, storage and network throughput and the storage speed
Taken from [Kuh17a]

*Access Memory (*NVRAM*)* and *Solid State Disks* (SSDs). By layering different storage devices, the one's advantages can be utilised to compensate for the disadvantages of others. The high latency of a tape drive has a reduced negative impact when combined with a fast device.

1.2. State of the Art

The origin of the I/O performance gap has been presented in the previous section. On the left side in figure 1.2, the situation in relation to the memory hierarchy is illustrated. Uppermost reside the fast hardware technologies like caches and Random Access Memory (RAM) while the parallel file systems form the bottom layer. The performance difference in between them significantly reduces the performance of the upper layer. On the right-hand side, the modifications are summarised to reduce the I/O bottleneck. Several layers are added improve the performance of the lower levels.

These changes complicate the data movement between the individual layers. A wide variety of approaches to maximise the performance have been developed over the last decades. In the best case, they function without introducing any complexity to the application level.
Zhang et al. proposed a scheme including an automated lookahead mechanism for data migration in multi-tiered storage systems containing SSD tiers [ZCD+10]. The optimal window size is adaptively determined by a greedy algorithm working on block level I/O profiles. They use this approach to analyse the impact of the granularity to the performance of data migration.

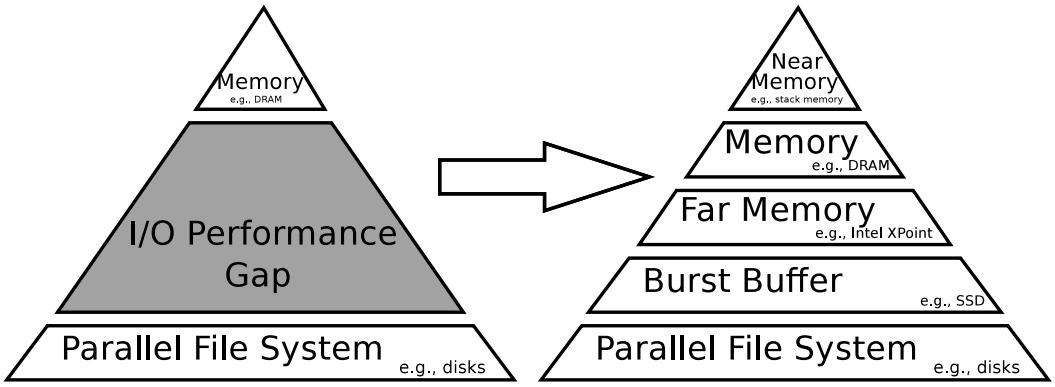

Figure 1.2.: An example HPC storage hierarchy to fill the I/O performance gap between main memory and disk. Shaded in grey is the problematic area where suitable solutions are not present.
Taken from [DBW+16]

A different suggestion is the *Data Elevator* presented by Dong et al. which intercepts I/O calls and moves the data to a fast tier such as a burst buffer providing persistent storage. The migration to the final destination is handled asynchronously later on. Their evaluation shows that the data elevator is four times faster than writing straight to the disk-based Parallel File System (PFS).

In recent years, simulation has become a rather important part in research making innovation without large budgets more accessible.
The behaviour of systems not affordable can be presumed and used for evaluating new ideas. It is ideal for experimenting with new approaches without inducing any downtime of a real system. Especially when testing system designs and file systems, possibly lost data is a concern. Even with recovery mechanism for an HPC storage system, it will take too long to restore deleted data due to the scale of the systems.
Simulation also enables the possibility to aid with the acquisition of new hardware by examining how current applications will behave on future devices. Often, the development periods are extended by difficulties in the manufacturing process leaving scientists waiting. In these situations, simulating those technologies provides the opportunity to continue research. However, as such a storage system consists of lots of different technologies a generic simulation is not trivial. Several approaches for a more specific scenario have been proposed which are explained in detail in chapter 3.

1.3. Goals

The ultimate goal of this thesis is to enable evaluation of data migration approaches on hierarchical storage systems. In HPC the ever-present gap between the growth of computational power in contrast to the capabilities of storage and network, technologies result in not fully utilised super computers. For applications to run as efficient as possible and to lessen the effects of the I/O bottleneck system specifics have to be considered. However, the developers who are writing those applications do not necessarily have the required knowledge. The increased performance of Central Processing Units (CPUs) allows solving more and more complex problems. Therefore, it becomes increasingly difficult to have detailed insights to all parts of the application. Additionally, applications are not only written by computer scientists who specialised in HPC programming but also by meteorologists or physicists. To handle this problem a lot of different software layers exist encapsulating knowledge in order to simplify the program development.
Solving more complex problems often also involves an increased amount of computed data. This is problematic as the systems do not dispose of the ability to store everything on fast memory with the highest bandwidth. Due to limited budgets, only a certain amount of fast hardware is available. To minimise waiting times of the application where data is read or written data management is essential. Furthermore, in climate computing, most of the results must be stored for years often even decades to later verify the used simulation models. The emerging requirements for acquiring such a system and efficiently managing the data on it are diverse. Therefore, several desirable tasks can be derived.

- Creating a simulation for hierarchical storage systems capable of supporting a wide variety of scenarios and hardware devices with a focus on the representation of:
  - not only old hardware but also current and future devices as well
  - homogeneous tiers consisting of different types of hardware
  - flexible interconnection for complex infrastructures
  - large-scale HPC systems
- Enabling an emulation of such storage systems to run application in
- Simulating the file system functionality for persistent storage and the possibility of data migration on top
- Designing a data migration library with:

- An intuitive API to the application layer to provide access to the emulated system with only small adjustments to the original application
- Different data replacement schemes such as Least Recently Used (LRU), Least Frequently Used (LFU)
- Different placement strategies between and inside tiers
- Different selection strategies to load files back into the fastest tier when accessed
- Persistent storage on underlying file system

1.4. Thesis Outline

Chapter 1 contains the motivation of the thesis as well as an overview of storage hierarchies and the challenges induced by them. Chapter 2 introduces the essential knowledge of file systems and hierarchical storage systems as well as on the used hardware. In chapter 3 existing simulators for *Hierarchical Storage Management (*HSM*)* and data migration are discussed. The overall design for the libraries FFS and DML is detailed in chapter 4. Internal design decisions regarding the storage emulator and its implementation are stated in chapter 5. Chapter 6 covers the design and current state of the realisation of the data migration library. In chapter 7 the planned measurements are explained. Finally, in chapter 8 a summary of the thesis is presented as well as an outlook on future work.

Summary

This chapter provided an introduction to today's HPC systems and the I/O bottleneck which is induced by the increasing gap between computational power and the speed of storage devices. An overview of proposed solutions is given and later used to derive the goals of this thesis.

2. Background

This chapter provides the necessary background information on file systems, storage hardware and the structuring of latter into hierarchical storage systems. First, the concept of a file system and the architecture of parallel distributed file systems is given. Afterwards, detailed insights into the memory hierarchy and its components are presented. Following, hierarchical storage management, tiering and information lifecycle management are explained enabling the analysis of methods to evaluate the value of a file in a specific environment.

2.1. File Systems

The analysis of storage hierarchies and data movement strategies atop requires detailed insights into several topics. Besides an understanding of the physical hardware also knowledge about the data management is crucial. Therefore, the following section provides an introduction into the most important features of file systems.

2.1.1. Essentials

A File System (FS) is responsible for organising and storing data. The majority of file systems is integrated into operating systems and also referred to as kernel file systems. Using the *File System in Userspace (*FUSE*)* framework implementing a file system running in the user space is possible. However, due to the increased number of context switches and mode transitions, a noticeable overhead is induced [RG10, Duw14].
File systems provide an interface to the underlying storage device and link an identifier such as the file name to the corresponding physical addresses of the storage hardware. This allows for a far more comfortable and simplified usage of storage devices.
In order to support a variety of file systems the *Virtual File System (*VFS*)* was introduced which is a kernel software layer offering a uniform *Application Programming Interface (*API*)* [BC05]. The interface semantics are defined by the *Portable Operating System Interface (*POSIX*)* standard.

Therefore, applications do not need to have any special knowledge about the internal structure of a file system. While this enables portability between different systems it also limits the possibilities of experimenting with new approaches.
The two basic components for structuring the file system namespace are files and directories which can contain files and other directories.
The essential concept of the VFS is the *common file model*.
One popular feature is often summarised with "everything is a file" meaning also directories are regarded as files holding a list of files and directories. They are accessed using their name called *path* in this context. Opening a file returns a file descriptor which is a unique non-negative number to identify the opened file and handle further I/O operations. Closing a file frees the file descriptor making it available again.
In a file system, the stored information is divided into data and meta data, representing the actual file content and additional information like file size and access times.

2.1.2. File System Types

Over the years, a number of different file systems have emerged. The development of the new storage and network technology and the rapidly growing data sizes lead to new and more and more sophisticated and specialised approaches. Prominent examples of file systems running on one client are ext4, B-tree file system (Btrfs) and Zettabyte File System (ZFS). Nevertheless, managing the exploding amount of data required for solutions which provide a significantly higher capacity and throughput. Distributing data over several storage devices offered the possibility to use their combined capabilities.
This resulted in the class of distributed file systems. A popular example is Nework File System (NFS). However, the highly parallel applications and large-scale I/O in High Performance Computing (HPC) demand for an even extended solution. Parallel distributed file systems such as General Parallel File System (GPFS) [SH02] and Lustre [BZ02] run on the majority of the TOP500's supercomputers [TOP17].

The abstract design of a parallel distributed file system is illustrated in figure 2.1. Principally, the components can be split into two groups namely clients and servers. Clients do not possess local storage, normally [Kuh15, p.16].The file system storage is only reachable through network communication as it is connected to the servers. In order to perform an I/O operation, the clients need to send requests via the network.

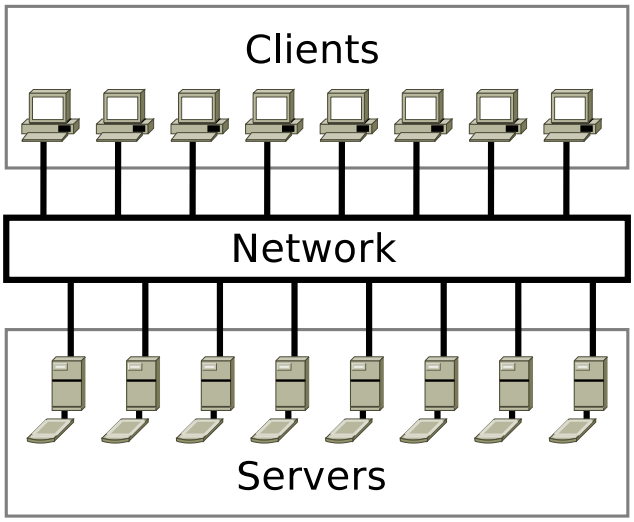

Figure 2.1.: The general system architecture of parallel distributed file systems
Taken from [Kuh15, p.16]

This is why latency and network bandwidth are critical to the performance of a parallel distributed file system.

2.2. Storage Hardware

Over the years research in areas such as semiconductor technology lead to enormous improvements especially in storage hardware. In 1956 a high-performing system as the IBM 350 disk storage unit had a capacity in the magnitude of several Mega Byte (1 000 000 Bytes) (MB) and a latency of 600 ms [arc]. The sizes today's systems reach are in the range Peta Byte (1 000 000 000 000 000 Bytes) (PB).
It exists a variety of technology with different features. Each performs well for a certain purpose and is less suitable for another. This circumstance resulted in a hierarchy of storage devices. Figure 2.2 illustrates the criteria which determine the layering.
After outlining the hierarchy the following subsections exemplify the mechanics and the salient abilities for the specific technology.

2.2.1. Memory Hierarchy

The memory hierarchy as depicted in figure 2.2 is affected by several parameters. The acquisition costs and the throughput grow from bottom to top. Tape drives are the most inexpensive and slow storage devices while having the largest

capacity. The time for which the data is stored is by far the longest and they also require the most space because of their physical size. Central Processing Unit (CPU) register form the counterpart being the smallest, fastest but also most expensive device type storing the data only for a few CPU cycles. Marked in red are those technologies which are volatile meaning they lose their content without power supply. Therefore, they are not sufficient for long-time storage. Yet, due to their high-throughput, they meet the requirements to serve as buffering storage.
By exploiting the specific features system designers aim to reduce costs through the use of inexpensive and high capacity devices for the majority of a storage system and to include the fast and expensive devices to increase the performance. Unfortunately, this introduces a lot of complexity for efficiently managing data. Approved data migration mechanisms are discussed in detail in section 2.3.

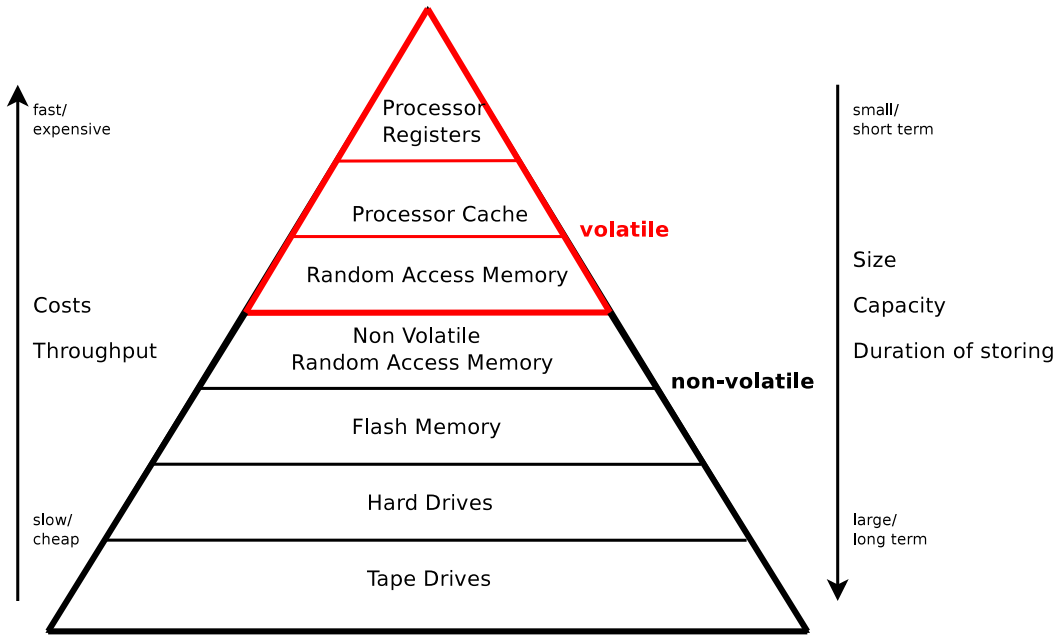

Figure 2.2.: Illustration of the memory hierarchy with volatile technologies marked in red and the non-volatile in black
Based on [Las07]

- **CPU registers** are the fastest and most expensive memory. They are directly integrated into the CPU. The resulting independence of the accesses from bus systems enhances the speed and contributes significantly to the performance of the CPU. Register capacities range between several Bit to kBit.

- **CPU caches** are also organised as a hierarchy. Typically, three levels are supported in modern CPU architectures. The related capacities are in the order Kilo Byte (1 000 Bytes) (KB) to MB. (see also 2.2.2)

- **RAM**, also referred to as main memory, is considerably larger than the level above with sizes of MB to Giga Byte (1 000 000 000 Bytes) (GB). (see also 2.2.2)

- **NVRAM** is the newest development in this listing. It characterizes a functionality similar to RAM extended to non-volatile storing. This minimizes the gap that previously existed between the performance of RAM and SSD which is illustrated in table A.3. (see also 2.2.2)

- **Solid State Drive (SSD)**, despite the name, do not contain moveable parts making them less prone to mechanical failures. They consist of flash memory which leads to a shorter lifecycle than HDDs because of a limited number of write operations. The provided access latency with an order of 100 μs is considerably faster compared against HDD latencies of about 10 ms (see table A.3). The prices range between (add value €/B) and (add value), the capacities between GB to Tera Byte (1 000 000 000 000 Bytes) (TB).

- **Hard Disk Drive (HDD)**s are magnetic storage devices which involve fast rotating platters to store the information. The rotational speed and the storage density limit the decrease of access latency to around 10 ms with storage capacities of GB to TB

- **Tape** also belongs to the category of magnetic storage devices. However, it does not use rotating platters but magnetic tape for storing the data. As the latency penalties, induced by the internal mechanics, are rather high it is not suitable for providing fast access. Nevertheless, the ability to preserve data for 30 years makes a good fit for long-term data storage as well as their capacities of several TB.

Table A.3 enables a direct performance comparison for the individual technologies. The *latency* of a system describes the time interval passing between an external impulse and an internal reaction. For storage devices, this is the time period between sending the request and the according access.

The classification of devices in a broader sense such as *primary storage* or online storage is not consistent. However, there is agreement primary storage implies direct access from the CPU [Com]. *Secondary storage* summarizes all devices persistently storing data which is not directly accessible by the CPU

Hierarchical level	Latency	Throughput
L1-Cache	≈ 1 ns	≈ (best) 1150 GB/s *
L2-Cache	≈ 5 ns	≈ (best) 500 GB/s *
L3-Cache	≈ 10 ns	≈ (best) 250 GB/s *
RAM	≈ 100 ns	≈ (best) 42 GB/s *
NVRAM	≈ 1,000 ns	
NVMe	≈ 10,000 ns	≈ (best) 3 GB/s [Sam16]
SSD	≈ 100,000 ns	≈ (avg) 500 MB/s [Ada17]
HDD	≈ 10,000,000 ns	≈ (avg) 136 MB/s [Sea16]
Tape	≈ 50,000,000,000 ns	≈ (best) 300 MB/s

Table 2.1.: Latency of device hierarchy levels
Latency values taken from [Kuh17c, p.5]
*Throughput values of Intel i7-5820K 6C/12T [Ryz17]

but via additional connection technology. Finally, *tertiary storage* includes all hardware used for long term storage such as tape archives.
This differentiation is not completely equal to the categories *online*,*nearline* and *offline*. Presumably, primary and secondary, depending on the specific circumstances even tertiary storage, correspond to online storage. The concept of a nearline device has no counterpart. All tertiary storage devices not included in online storage belong to offline storage.
The distinction between online, nearline and offline is in the provided availability for I/O as listed below. These definitions have been taken from [Ton10]. Neither the rotational speed nor the purpose is crucial for this classification.

- **Online** storage is able to perform I/O instantly. This level is formed of DRAM, SSDs as well as HDDs that are always spinning.

- **Nearline** storage denotes those storage devices that are not immediately available but need no external procedure to become online. This applies for *Massive Array of Idle Drives (*MAID*)* systems which consists of an array of HDD. The predominant part of them is not running. However, they can be sped up on demand. The design has been developed for *Write Once, Read Occasionally (*WORO*)* usage. Also, automated tape systems belong to nearline storage.

- **Offline** storage is neither instantly available nor can it be automatically started. Removable media like USB memory sticks or CD/DVD fall into this category just as tape cartridges lying on a shelf.

2.2.2. Storage and Network Technologies

Cache

The performance of CPU and RAM differed and still differs by several orders of magnitude. This situation worsens when the hardware at the top of the memory hierarchy is replaced with a newer model having an even increased speed. To support such a system modification, it is not sufficient to just enlarge the internal cache of the CPU. Often, changes to the lower part of the system need to be made, too [Mä94]. To keep the necessary adaptation as small as possible an additional hierarchy was introduced which is presented at the top of table A.3.
These cache level hierarchy aims to smooth the transition between the CPU and the rest of the system.
Another reason for a hierarchical approach is the effort it takes to produce a large and fast cache. It is far more affordable to have a small fast first level cache and an additional significantly larger cache which is still more powerful than the memory to be cached.
The management of these hardware caches and all occurring consistency problems are dealt with at the hardware level. Therefore, the software layers on top do not need to concern themselves with several different caches. They just interact with the cache as a whole.
In general, the following analysis does not only apply to CPU caches but also other types such as page or network caches. The basic assumption of caches is known as the *locality principle*. It argues that due to the cyclic structure of programs and the usage of linear arrays to store related data, addresses in the area of already used addresses are more likely to be accessed in the close future [BC05]. Hence, a cache stores the most recently used data and code.
The cache is split into lines which can be mapped to the main memory in three common ways. One possibility is the *direct mapping* where each main memory line is always projected to the same physical address in the cache. The opposite approach is called *fully associative* meaning there is no mapping at all. The combination of both is realised in the *n-way set associative* cache where each line in main memory can be stored each of n lines of cache [BC05].
Each of these options comes with their strength and weaknesses. To illustrate them first a brief summary of the general cache functionality.
A request sent to the cache for a certain line can either be successful when the line is already present in the cache called *cache hit* or it fails with a *cache miss* if the line is not available. In case of a cache hit the requested data is returned. More complicated is the handling of a cache miss. The block needs to be loaded from a layer below. As cache sizes are limited in order to store it some existing

data needs to be replaced and written back.

Depending on the current mapping there are different types of cache misses:

- **Compulsory misses** are not avoidable and occur when first accessing the memory. An approach to decrease the number of compulsory misses is the use of a prefetcher loading data which they assume to be accessed in the near future.
- **Capacity misses** happen because of the limited size. The only solution for this problem is a larger cache.
- **Conflict misses** take place when blocks are replaced because they are mapped to the same cache line even if there is free space left. This is possible when the cache is not fully associative. The limitation for a main memory line to be stored only to a certain number of lines is not effective as a replacement is issued even when the rest of the cache is empty. This is the reason why the Operating System (OS) page cache is fully associative.
- **Coherency misses** are only present in multi-processor systems. To avoid inconsistencies between the cache of different CPUs solutions as the write-invalidate protocol are deployed. When the write in one cache invalidates a block which is also loaded into a different cache, then this instance needs to be replaced and updated as well leading to this specific type of miss.

A good replacement strategy is vital to the performance which is illustrated in the following. The *Average Memory Access Time (*AMAT*)* is calculated as stated in equation 2.1 [ADAD15, ch.22,p.2].

$$AMAT = (P_{Hit} * T_M) + (P_{Miss} * T_D) \tag{2.1}$$

T_M and T_D represent the costs of accessing memory respectively disk. P_{Hit} and P_{Miss} describe the probability of a cache hit or miss ranging between 0 to 1 with $P_{Hit} + P_{Miss} = 1$. Assume T_M as 5 ns and T_D as 10 ms (see A.3).
The average access time is calculated for the hit rate of 90% and 99.9%.
$P_{Hit} = 0.9$: $AMAT = (5 \text{ ns} * 0.9) + (10 \text{ ms} * 0.1) = 1,000,005 \text{ ns}$
$P_{Hit} = 0.999$: $AMAT = (5 \text{ ns} * 0.999) + (10 \text{ ms} * 0.001) = 10,005 \text{ ns}$
This example demonstrates how a few cache misses lead to a drastically increased AMAT. Therefore, it is crucial to keep the possibility of a cache miss a low as possible thus needing the best replacing mechanism possible.
The optimal replacement strategy was proven in 1966 by Belady [Bel66].
Actually, the solution is astonishingly simple yet not easy to implement. The block accessed furthest in the future needs to be replaced which results in the fewest misses overall. However, determining this block is a challenging task and

can only be approximated if no test run with the same parameters is performed beforehand.
Knowledge about the theoretical optimum still offers a good reference point to evaluate other strategies.

- **Random**: As the name suggests an element is picked randomly and replaced making it easy to implement.
- **First In, First Out (FIFO)**: With this concept, the oldest element is replaced regardless of the access frequency or pattern. It is easy to implement but far from an optimal approach. Actually, the situation for FIFO and the random strategy is even worse. They suffer from what is called Belady's Anomaly [BNS69]. Belady et al. found that increasing the cache size does not imply an increase in the hit rate. In fact, it can get worse with a larger cache. The advantage of FIFO over a random approach is implementing a counter is easier than to generate a random number [Bel66].
- **LRU**: In contrast to the above, this algorithm considers the history of accessing. The element which has not been accessed the longest is evicted. LRU is not impacted from the anomaly because it has the stack property [MGST70] meaning the content of a cache with size N is always included in a cache of size N + 1. This is why enhancing the cache will raise the hit rate. LRU has a run-time complexity of $O(1)$ which in addition to its behaviour makes it the algorithm most broadly used [SMM10]. Its performance for different workloads is shown in figure 2.3
- **Not Recently Used (NRU)**: This algorithm involves a time interval which acts as a threshold determining whether a line was referenced lately or not. Like LRU and LFU it is based on the locality principle resulting in a better hit rate if the application also follows the principle which is not always the case.
- **LFU**: Similar to LRU it also takes past accessing behaviour into account. For this replacement strategy not the passed time since the last access to a file is relevant but the total number of accesses. This is why, LFU is superior for scenarios where the lines are solicited in round robin fashion and do not fit in the cache at the same time. With LRU all requests result in a miss because the lines are constantly loaded and replaced. In comparison, LFU will end up with a quite acceptable hit rate. The workload is pictured in Figure 2.3 termed "looping sequential". The best known run-time complexity was $O(\log n)$ until in 2010, Shah et al.

presented an algorithm with a runtime complexity of $O(1)$ for all operations [SMM10]. Before, LFU was implemented using a binomial heap structure to get the item accessed the least. The new approach is based on two double-linked lists, one for storing the access frequency and one for all items sharing the same access frequency. The detailed pseudo code can be looked up in [SMM10].

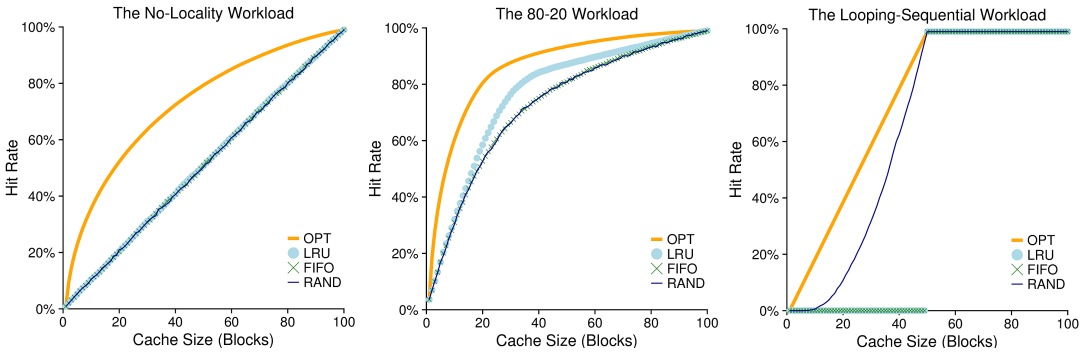

Figure 2.3.: Evaluation of the replacement policies 'random', 'FIFO' and 'LRU' in comparison to the theoretical optimum for the following workloads: no-locality, 80-20 and looping sequential
Taken from [ADAD15, ch.22,p.9ff]

Figure 2.3 illustrates the performance of the random replacement algorithm, FIFO, and LRU in contrast to the optimum for two different types of workloads. For each run, 100 unique pages are accessed 10,000 times in total while varying the cache size from 1 to 100. On the left, the situation for the no-locality workload meaning random accessing is pointed out. Obviously, for this kind of request sequence the three are equally bad compared to the optimal algorithm because the assumptions regarding the behaviour of the application FIFO and LRU work with do not apply. Therefore, the hit rate is directly proportional to the growth of the cache size. This situation changes when the workload is modified. For the second benchmark 80% of the requests access 20 % of the pages which are referred to as "hot" pages. All policies have an increased hit rate in comparison to the random workload. However, LRU performs better at keeping the hot pages. As shown in the example above this performance difference may be crucial depending on the access latency for loading missing lines from a lower layer.
The last workload is termed "looping sequential" which describes the following access pattern. After, 50 unique pages are accessed the same pages are referenced in the exactly the same sequence as before. In total, 10,000 accesses are performed as in the other setups. This is a behaviour often found in database

applications [ADAD15] representing the worst case for both FIFO and LRU. Even with a cache size of 49 all requests result in a miss. Only if the cache size is large enough to hold all accessed pages the hit rate is above zero. The random policy is able to cope with this situation notably better because it does not evict a file right before it will be accessed again.

RAM

RAM is the abbreviation for *random access memory*. Usually, this term is used for volatile memory which allows read and write access. However, there are also technologies providing random access that are non-volatile such as ferroelectric RAM (FRAM), magnetic RAM (MRAM, and phase change RAM (PCRAM) or which provide read-only memory (ROM). In the following, the more common definition is utilised [Wik17e].
The access time of RAM does not depend on the physical data location because it does not contain moveable components like heads. In contrast, to HDDs or tape drives RAM is directly addressable and does not have to be accessed sequentially or in blocks. It is organised in a matrix layout of memory cells. Larger devices address more than one bit; the specific length depends on the architecture. There exist a number of different technologies. One important distinction is whether the RAM is reliant on a clock signal. Asynchronous devices provide the data only after a certain time which depends on the used materials. Their maximum throughput is lower than the one of a synchronous device whose operation is controlled by a clock signal. Both of the functional principles below can be as either synchronous or asynchronous.

- **Static RAM (SRAM)**: This type of memory provides data remanence while still being volatile as the data is lost when the power supply is interrupted. In contrast to DRAM, it is more expensive but also more powerful and therefore used in CPU caches.

- **Dynamic RAM (DRAM)**: In comparison to SRAM it needs to be periodically refreshed to maintain its content. As it is considerably cheaper the main memory of a system often consists of DRAM.

NVRAM

As the name suggests NVRAM is a type of RAM which retains the stored information without external power supply. Although flash memory is a prominent member of this category it induces problems as it only provides access to larger blocks making it insufficient for small and fast caches. Moreover, its

limited number of write-cycles is a significant problem for its usage as a cache as they involve enormous amounts of I/O operations. Other approaches either combine volatile RAM with additional batteries or incorporate technologies like FRAM, MRAM or PCRAM. Most of them store information similar to DRAM by charging a small capacitor. Also, the utilisation of semiconductors such as silicon carbide (SiC) is evaluated for realising bistable NVRAM cells. In 2015, Intel and Micron Technology presented a new NVRAM concept referred to as 3D XPoint. It was announced to have a smaller latency than NAND memory and a larger number of write-cycles [Wik17a]. While NAND flash persists information by using different electric potentials in field-effect transistors 3D XPoint is based on the alteration of the bulk resistance enabling a higher packing density. The memory is organised as a stackable grid structure containing the actual storage elements at the crossing points. In April 2017 the Intel Optane SSD became available to open market, having a latency 1000 times faster than NAND flash with a density increased by a factor of ten compared to DRAM [Pet17]. As illustrated in Table A.3, NVRAM fills the gap between RAM and SSDs while providing persistence making it suitable as caches for HDDs and SSDs and for storage of data which is frequently accessed and modified.

SSD

Like the storage devices discussed above SSDs also do not contain mechanical parts enabling a faster and constant access to the memory than provided by HDDs. They are either based on flash memory or DRAM or a combination of both called NVDIMM where the data is written from DRAM to the flash memory to persist. DRAM is far more powerful but also more expensive than flash memory and consumes more energy. For that reason, the former type is used in the majority of systems. SSDs do not offer byte but block granularity. The block size is often set to the page size of 4 Kibibyte which guarantees that only one page needs to be rewritten. SSDs can only write erase blocks because of the used hardware mechanism. This concept results in the requirement to read the complete block, modify it in memory, erase the old state and write it back to the SSD which makes write operations considerably slower than read operations. Depending on the quality of a flash cell it can only perform between 3000 and 100000 write and therefore erase operations [Wik17c]. Flash memory stores its information on a floating gate. When erasing a cell electrons tunnel through the oxide layer surrounding the gate which requires high voltages. This damages the oxide layer which finally loses its insulating ability. Thereby, the electrons drain off the gate causing information loss. However, this does not mean the whole memory is unusable. Wear levelling mechanisms aim to elongate

the life span. With *dynamic wear levelling*, the least used free block is chosen for an operation increasing the number of write-cycles by a factor of 25. Albeit, the free memory is worn fast when the larger part of blocks is already filled. An even attrition is achieved by *static wear levelling* which selects the block least used. This approach necessitates a more complex controller managing situations when the block is currently in use, moving the data to a different block. Static wear levelling enhances the write cycle by an order of hundred compared to no wear levelling. Furthermore, Soundararajan et al. proposed the utilisation of HDDs a write cache to enhance the life span of SSDs [SPBW10]. Their hybrid storage device contains a log-structured HDD and migrates data periodically. Evaluating this approach showed a doubling of the SSD lifespan and a decline of the average I/O latency by 56%. The decreasing acquisition and maintenance costs and increasing performance make SSDs a continuously more prominent alternative to HDDs when considering the TCO [EDD13]. For applications inducing a random I/O access pattern like database applications, they are already in use

HDD

An HDD consists of one or multiple rotational platters coated with a magnetic surface. They are bound to a spindle which is spinning at a constant rate in case of power supply. The velocity of the spindle is measured in *Rotations Per Minute (*RPM*)* which currently ranged between 7200 RPM and 15,000 RPM . The disk head which reads and writes the information is connected to a disk arm that is responsible for positioning the head at the requested location. This induces the seek time which is spent for repositioning the head for each request. It is an important property of an HDD especially when comparing different storage devices as it increases the access time which is calculated as follows.

$$T_{I/O} = T_{seek} + T_{rotation} + T_{transfer}$$

The average seek time is one-third of the full seek time which is based on the seek distance [ADAD15, ch.37 p.9]. As the access time is a crucial feature to consider when optimising the performance and utilisation of a storage system, the average seek distance is characterised below.
It is the sum over all possible seek distances between two tracks x and y on a platter divided by the number of different possible seeks.

All possible distances can be described as: $\sum_{x=0}^{N}\sum_{y=0}^{N}|x-y| = \int_{x=0}^{N}\int_{y=0}^{N}|x-y|\mathrm{d}y\mathrm{d}x$

$$= \int_{x=0}^{N} (\int_{y=0}^{x} |x-y| \mathrm{d}y + \int_{y=0}^{N} |y-x| \mathrm{d}y) \mathrm{d}x$$

$$= \int_{x=0}^{N} \Big((xy - \frac{1}{2}y^2)\Big|_0^x + (\frac{1}{2}y^2 - xy)\Big|_x^N \Big) \mathrm{d}x$$

$$= \int_{x=0}^{N} (x^2 - Nx + \frac{1}{2}N^2) \mathrm{d}x = (\frac{1}{3}x^3 - \frac{N}{2}x^2 + \frac{N^2}{2}x)\Big|_0^N = \frac{N^3}{3}$$

This results needs to be divided by the number of different possible seeks which is N^2. This leads to an average seek distance of one-third as stated in 2.2.

$$\textbf{Average seek distance: } \frac{\sum_{x=0}^{N}\sum_{y=0}^{N}|x-y|}{N^2} = \frac{\frac{N^3}{3}}{N^2} = \mathbf{\frac{1}{3}N} \tag{2.2}$$

Besides the performance limitations and the discussed seek times, HDDs are still widely used for large storage systems as their price is significantly lower than of SSDs [EDD13]. In hierarchical storage systems, HDDs typically form a high-capacity tier below the high-velocity tier of SSDs in order to fulfil the demands made on HPC systems [WLC+14].

Magnetic Tape

Magnetic tape is usually made of long and narrow plastic film coated with a thin and magnetisable layer. To protect the tape from environmental influences it is enclosed in a cartridge which is inserted into a tape drive to read and write data. Over the last decades, several tape data layouts have been developed such as linear, linear-serpentine and helical layouts. The linear approach is the simplest one allowing multi-channel read and write operations [Lü16]. Mounting tapes is mostly handled by automated tape libraries today. The tapes are held in a shelving system also containing robots for the cartridge insertion. Modern tape systems offer a variety of features including self-describing data formats increasing the portability of data, data reduction and compression mechanisms and data encryption. Tape storage systems are more resistant to external impact than most other storage devices. As the current enclosures are sufficiently keeping moisture out, reducing the possibility of fire is the most important task for guaranteeing long-term storage because the tape is highly flammable. Given a suitable precaution in case of disaster magnetic tape is very durable offering reliable data storage for 30 years. Due to its low cost and energy efficiency coupled with its high capacity, tape storage systems will likely remain present for the near to mid-term future [Lü16].

2.2.3. Ethernet

Ethernet is a technology which specifies software protocols as well as hardware for linked data networks and was originally intended for Local Area Networks (LANs). It enables the transmission of data frames between devices connected to the LAN. Since 1990 it became the most widely used LAN technology replacing competing wired LAN standards such as Token Ring and *Fiber Distributed Data Interface (*FDDI*)*. It is generally compliant with the Institute of Electrical and Electronics Engineers (IEEE) standard 802.3 [Wik17b]. Participants of a LAN send messages via their shared wiring system where every network interface has a MAC address, a globally unique key of a length of 48 bits. Ethernet sends data via the transmission medium using baseband processing and time-division multiplexing where the access is managed by the Carrier Sense Multiple Access Collision Detection (CSMA/CD) algorithm. In the beginning, the communication bus was realised by a coaxial cable and evolved over the last decades to twisted pair and to fibre optic links. The currently specified bandwidths are 10 MBit/s, 100 MBit/s (Fast Ethernet), 1000 MBit/s (Gigabit Ethernet), 10, 40 and 100 Gbit/s.

2.2.4. InfiniBand

InfiniBand is a high throughput and low latency network communication standard. It does not belong to the IEEE-802.3 Ethernet standard but supports the Ethernet data format and is connected with copper cables which are also used for 10 Gigabit Ethernet allowing transmission distances of about 15 meters [Wik17d]. To enable the transfer over longer distances fibre-optic media converter can be used. *a*re used as network interface controller.

In 1999, the combining of two competing designs resulted in InfiniBand and the formation of the InfiniBand Trade Association (IBTA) including Compaq, Dell, Hewlett-Packard, IBM, Intel, Microsoft and Sun Microsystems. These companies joined forces in order to develop a solution to the I/O bottleneck. InfiniBand is mostly used in HPC systems. By 2009, 181 systems of the TOP500 list employed InfiniBand while 259 were interconnected by Gigabit Ethernet [Ste09]. The intended evolution of the link bandwidth is illustrated in Figure 2.4 where SDR is the single data rate, DDR the doubled data rate and so forth. In Table 2.2 the actual signalling rates and the according adapter latency is shown.

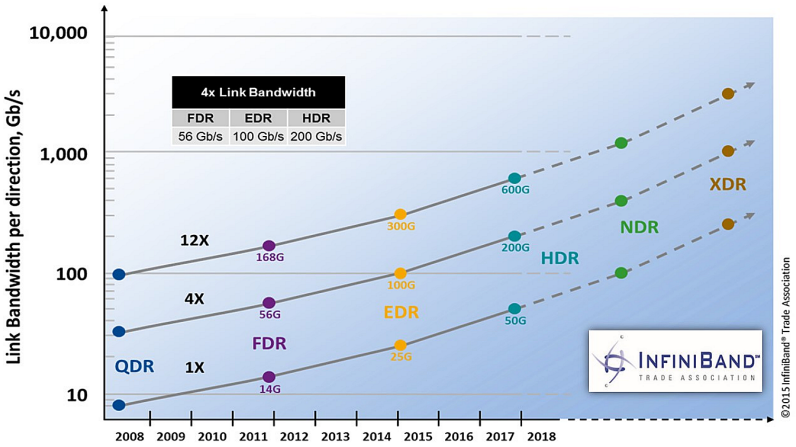

Figure 2.4.: InfiniBand roadmap picturing the evolution of the link bandwidth
Taken from [Mel17]

.	SDR	DDR	QDR	FDR10	FDR	EDR	HDR
Signaling rate (Gbit/s)	2.5	5	10	10.31	14.06	25.78	50
Speed for 4x links (Gbit/s)	8	16	32	40	54.54	25	-
Speed for 8x links (Gbit/s)	16	32	64	80	109.08	100	-
Speed for 12x links (Gbit/s)	24	48	96	120	163.64	200	-
Adapter latency (μs)	5	2.5	1.3	0.7	0.7	0.5	-
Year	2001*	2005	2007	2011	2011	2014	2017

Table 2.2.: Signalling rate of InfiniBand and the related atency of the adapter taken from [Wik17d].

2.3. HSM and ILM

Storage tiering or tiered storage has become a catchphrase in recent years [Pet06]. Despite the frequent usage, there is some confusion regarding the meaning. Often, the terms *Information Lifecycle Management (*ILM*)*, HSM and tiering are mixed up. Therefore, the differences are highlighted in the following. They are illustrated based on the definitions by the Storage Networking Industry Association Data Management Forum (SNIA-DMF) [PS04]. Throughout this thesis, this distinction is maintained.

- **Tiering** or tiered storage captures the forming of a storage hierarchy based on certain requirements such as performance, security, and costs. Different examples concerning the hierarchy design as well as the wide list of possible requirements were already presented in section 2.2. Placing the data on the formed tiers follows a previously defined mechanism. There

are three main approaches:
  - static: The application is responsible for distributing the data, assigning it to specific tier.
  - staged: This mode is used for data archiving.
  - dynamic: The data is actively moved by HSM or ILM policy.

- **HSM**: The rising of different storage technology offering a wide variety of features led to the memory hierarchy explained in section 2.2. The higher levels act as caches for the lower ones improving the access times for frequently accessed data which is kept at the faster layers. The data movement is scheduled by algorithms between high-performing and expensive devices on the one end and inexpensive, slow devices with a large capacity on the other end of the hierarchy. Those scheduling algorithms are often time-dependent and are thereby capable of handling time-dependent information well. A vast amount of data, however, may be time-independent and is not dealt with in sensible way when only focusing on access frequency [JXW08]. Jin et al. present a model to determine the information value. This is discussed in detail in 2.3.1

- **ILM** is not one dedicated service but the combination of tiered storage hardware, storage software, possibly middleware and services working on top. It is an expression risen from the marketing of the storage industry. It summarises the strategies and applications trying to achieve an optimal way of efficiently storing, providing and archiving information based on their value and usage.
  In their roadmap, Storage Networking Industry Association (SNIA) outlines a path how to establish a broader ILM-based practice by introducing more elaborate tiering strategies. In figure 2.5 their proposal for a general approach is shown. They describe identifying the value and classification of information as a fundamental first step.

There also exist quite a number of terms such as *data migration policy*, *data movement strategy*, *ILM policy* all describing an at least partially automated process of moving data between different storage tiers.
For this thesis, no further distinction is made using them equally.

2.3.1. Information Value Evaluation Modelling

If not noted otherwise, this section is based on the work of Jin et al. [JXW08]
As shown in figure 2.5, the essential step to enable valuable data migration

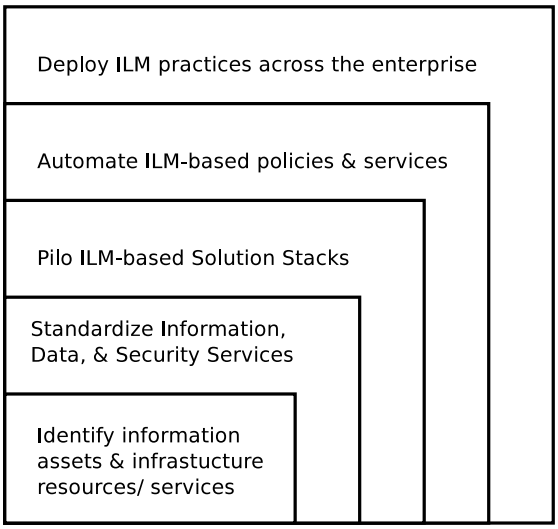

Figure 2.5.: Roadmap by SNIA towards an ILM implementation
Taken from [Pet06]

policies is to determine the value of information. There has been quite some effort in trying to come up with an appropriate evaluation model.
To illustrate the difficulty of accurately capturing the value of information in correspondence to the surrounding system an approach towards a formal is given. This is used as a model basis in the design of the DM library. Jin et al. presented an information value evaluation model resting on a variety of parameters.
Including concepts from economic theory such as use-value as well as the supply and demand relationship, they derive the following definition of information value. A piece of information means a file.
Hence, information demand denotes the usage which can be deduced from the access pattern. Accordingly, information supply refers to the provision of information or the quality of information service. The information use-value is indicated by its ability to satisfy the user's needs which then leads to the information value.
The information use-value determines the result of the evaluation model named *Output of Information Value Evaluation (*OIVE*)*.
If the demand exceeds the supply OIVE increases reaching its upper bound when a file cannot be accessed at all for a certain period. The other way round OIVE decreases when a storage device is not accessed and the storage space and bandwidth is wasted.

The OIVE is calculated by the equation 2.3 for file d over a period of time t.

$$OIVE_t(d) = f(t) * N^2 * (1 + M \cos\theta) * \frac{size(d)}{B} \tag{2.3}$$

Equation 2.3 consists of the information supply and demand, the degree of sharing and the information association of a file.
N^2 is the degree of sharing which identifies the number of users accessing this file.
Two files are considered associated if they are often accessed after one another. This circumstance is noted as $(1 + M \cos\theta)$.
To every file belongs an access vector $A_i = (a_{i1}, a_{i2}, ..., a_{in})$, $i = 1, 2, ..., l$, with a_{ij} being the number of accesses on jth day for the information sequence i and l the number of files in the system.
The degree of association for two files is stated in equation 2.4.

$$\cos\theta_{ij} = \frac{A_i \bullet \mathrm{A_j}}{|A_i||\mathrm{A_j}|}, (i \neq j), A_i \bullet \mathrm{A_j} = \sum_k a_{i\mathrm{k}} a_{j\mathrm{k}}, |A_i| = \sqrt{\sum_k a_{i\mathrm{k}}^2} \tag{2.4}$$

$$\cos\theta = \frac{\sum_i \cos\theta_{ij}}{l}, (i \neq j) \tag{2.5}$$

The association of a file to every other file over all possible information sequences is calculated as equation 2.5.
M in equation 2.3 counts those values of $\cos\theta_{ij}$ exceeding a threshold th.
$\frac{size(d)}{B}$ describes the supply as the ratio of file size to the systems' bandwidth B. If there is more than one instance of a file B is the sum of bandwidth of all related storage devices.
The information demand is defined as $f(t) = \frac{1}{(\Delta t)^2}$, with Δt the passed time between t and the time of the next access. The probability distribution of access time intervals is presumed to behave as a *Poisson Process*.
In order to calculate $f(t)$ the arrival rate λ is essential which is assumed to be the average arrival interval.

2.3.2. Evaluation Criteria for Data Migration Policies

The distribution of data across a storage system can be oriented towards different goals. In some contexts reducing the energy consumption is more important than further increasing performance. However, most of the time the aim is to decrease the access time and thereby enhance the reached throughput. In

order to evaluate the applicability of file placement strategies, Wijnhoven et al. brought together several policy criteria they found in their literature review [WA10]. They have been originally published Chen et al. as well as Turczyk et al. [Che05, THBS06].

- The data movement needs to function with sparse to no human intervention.
- The time-dependent value of a file is an essential parameter for the policy.
- The evaluation process is not based on solely one file attribute.

Different approaches have been taken over the years, e.g. Poore et al. [Poo00] advanced from a security risk management perspective. Their model analyses how information utilisation is linked to transactions and thereby revenue. Although it is principally suitable for determining the information value its usage is too time-consuming and not adaptable to changes [JXW08].
In Table 2.3 four file retention policy determination methods meeting the requirements above are specified. Chen et al. introduce an approach determining the file value throughout its entire lifecycle by relying on the frequency and the recency of its use [Che05]. While they consider the changes over time neither the domain knowledge of administrators or users is included. In addition, the file value is not inevitably captured by its access frequency. Contracts, for example, have a high value despite their less frequent usage.
By taking the passed time since the last access, the file age, the number of accesses and the file type into account Turczyk et al. aim to derive the probability for future use [TFLS08]. The disadvantage of their method is that despite considering various file characteristics the actual content and context of a file do not contribute to the result [WA10].
Zadok et al. target the decrease of used storage by determining policies to enable the usage of free disk space to other users [ZOS+04]. By highly depending on the experience of data administrators as well as users subjective allocations are a common result [WA10].
Shah et al. developed the ACE framework for creating file retention policies [SVSA06]. It has been designed with the following objectives. First, it incorporates business valuation of the data based on their metadata. Second, it allows identifying the separate tiers of available storage quality. Third, providing data migration schemes to assure informed utilisation of the resources. As stated in Table 2.3 it fulfils all the criteria. It does, however, involve some problems such as the policy specification by data users, not administrators. The users do not have a sufficient overview of the system and the key metadata attributes. Also, the development of the policy is far too time-consuming when trying to realise a

complete set of policies including all files. Wijnhoven et al. evaluate the relation between file attributes and the use value of files as the ACE framework does not recommend specific guidelines how a policy can be developed depending on the users' evaluation of files [WA10]. The concluding results are oriented towards business applications making the recommendations not completely applicable to HPC systems. While discarding the file type as an indicator they suggest to integrate the position of a user which is mostly the same for HPC users.

Criterion	Chen	Turczyk	Zadok	Shah
Little human intervention	x	x	x	x
Frequency of use	x	x		x
Measurable metrics	x	x	x	x
Classification of data	x	x	x	x
Knowledge of data users			x	x
Cost reductions	x	x	x	x
System performance				x
Business value of data				x

Table 2.3.: Assessment of the methods proposed by Chen et al., Turczyk et al., Zadok et al. and Shah et al. [Che05, TFLS08, ZOS+04, SVSA06]
Taken from [WA10, p.4]

Summary

This chapter presented the essential knowledge to understand the following analysis of hierarchical storage systems and the current solutions to lessen the performance penalties induced by the I/O performance gap.

3. Related Work

This chapter provides an overview over related works which are discussed in detail. After the current situation is outlined several simulation tools are presented including their strength and weaknesses.

3.1. Simulation Tools

The evaluation of theoretical models has always been an essential point in scientific research regardless of the specific domain. With the rapid technological evolution, computer simulation became a vital step in the validation of theories. This is especially true for areas which involve the analysis of very small or very large systems, e.g. particle physics. HPC by providing the appropriate resources has developed into an individual field of interest. The difficulties faced like the growing gap between the computational power and the speed of storage devices lead to a variety of simulations in order to optimise the system's performance. The discussed diversity of storage systems results in a big number of possible approaches focussing on certain aspects. In the following, three simulations for storage hierarchies are presented which have been published in the last year as well an approach how to simulate future hardware.

3.2. DUX

Krish et al. proposed *DUX*, which is short for an "*application-attuned dynamic storage management system for big data processing frameworks*"[KWI+16].
They analyse the behaviour exhibited applications running on SSDs and HDDs which alone is not a new idea as the trade-offs have been evaluated before, e.g. by Narayanan et al. [NTD+09]. However, the overall aim is to identify to which degree certain workloads profit of the SSDs features and which applications might as well run on HDDs. Most data migration strategies used in tiered storage system include a number of values in order to determine the data to be moved.
Albeit, these approaches do not consider which applications actually benefit the most by running on an SSD tier. So, instead of increasing the percentage

of faster hardware in the system they try to use the system more efficiently. Therefore, DUX provides the possibility to profile applications and the related I/O and to make decisions at run time based on the available hardware. Three optimisations are pointed out. First, the data placement is based on the I/O pattern. Second, the intermediate data I/O between the individual chunk replicas in the HDFS (Hadoop File System) is moved according to application characteristics. The third proposal affects the prefetching of the data from the tier below. The information value of a file is initialised with an average value. After a specific time interval has passed the files are rerated and the value for the next interval is predicted. This advice is then evaluated on synthetic Facebook workloads generated by MapReduce cluster traces. The main result is the finding that, when 5.5 times less SDDs are incorporated in comparison to a solution including only SSDs, DUX introduces only 5% overhead.

While these are promising conclusions the simulation lacks certain features. On the one hand, it enables only heterogeneous systems meaning an SSD tier and an HDD tier, not a mixed tier consisting of both. On the other hand, the integrated devices comprise neither faster candidates as NVMe nor long term storage devices as tape systems. Also, the simulation does not offer a possibility to evaluate an application by providing an emulation of the whole storage system.

3.3. OGSSim

Gougeaud et al. came up with an approach called *OGSSim* an abbreviation for *Open and Generic Storage systems Simulation Tool* meeting some of the above requirements. They find that none of the existing simulators and their extensions satisfies them. Several approaches have been made like *DiskSim* by Ganger et al.[BSSG08], emulating systems like *OpenSSD* or a combination thereof like *VSSIM* by Yoo et al [YWH+13]. DiskSim, for example, endorses a highly detailed model including disks, controllers, buses, device drivers, schedulers, caches. VSSIM simulates SSDs by merging a trace based simulator with an emulator using a virtual machine as a basis and running additional software atop allowing for a multitude of fine-grained design options. All these tools provide sophisticated simulations but only for one specific type of hardware. However, neither of them is suitable to simulate the diversity and scale of an HPC system.
OGSSim is designed to offer support for heterogeneous, very large systems including a variety of different configurations. The largest drawback is that despite the announcement to publish the code it is nowhere to be found. This

circumstance caused San-Lucas et al. to implement a similar tool which is discussed subsequently. The internal design of OGSSim consists of several components each of which is executed by a thread and communicating with the other modules via message transmission using the Zero Message Queue (ZeroMQ) API.

- **Workload**: This module builds the internal request array a given an input trace file. OGSSim offers different request formats. The basic format includes a timestamp, the type of operation, the address and the size. It can be enhanced by host and process identifiers.
- **Hardware Configuration**: Taking an XML file as an input this module establishes the hardware structure in shared memory. Structuring elements are tier, volumes and devices, where a volume is a number of devices with a Redundant Array of Independent Disks (RAID) or Just a Bunch Of Disks (JBOD) organisation.
- **Simulation Configuration**: Another XML files specifies the configuration parameters for the simulation. For example, it is customisable whether the logging should include warnings or only errors and whether certain events such as device failure can happen.
- **Pre-processing**: The pre-processing component is responsible for the instantiation of the volume driver module as well as for launching the simulation process.
- **Volume Driver**: This module is engaged in the layout creation for the volume type and the subsequent instantiation of the device driver.
- **Device Driver**: Specific sub modules corresponding to maintenance tasks are forming the device driver. They generate requests for the execution module regarding e.g. garbage collection, wear levelling or defragmentation.
- **Execution**: The function of the execution module is to calculate several metrics necessary to evaluate the performance of a system. These are the request response time, the service time, the transfer time, and the waiting times for the components of the storage system. All of them are written to the output file together with additional information like the request index or the device.
- **Performance**: After the simulation finished the performance module allows a variety of different visualisations, e.g. graphs or histograms, for the results.

The validation of the proposed simulation tool took place by comparing the simulation results for a read resp. write request stream to the performance of actual hardware devices. At the time of the publication, Gougeaud et al. only tested the behaviour for SSDs. They regard the maximum difference of 15 % as significantly accurate. While they aimed for a universal simulation tool, OGSSim consists of only two types of devices, namely SSDs and HDDs. The latter are not even validated for this publication. Nevertheless, they present a promising simulator with highly customisable hierarchies and different execution scenarios as the normal and the degraded mode which contains the simulation of hardware failures.

3.4. StorageSim

As previously mentioned, the implementation of Gougeaud is not accessible which is why San-Lucas et al. proposed their own realisation named *StorageSim*. Their main focus lies on providing a fast simulation that is capable of handling big data workloads which they locate at around 8000 operations per second. StorageSim offers adaptable tiers and different hardware device types to customise the storage systems and enables evaluating several layouts. It includes three simple data migration policies. The first is a random placement based on a hash function. The second strategy is SSD caching where all files are placed in the HDD tier in the beginning and copied to the SSD when accessed, whereas the third policy classifies the files based on their age and places the older ones at a tier with lower throughput. LRU is used as a replacement strategy. The evaluation of StorageSim is performed with two different experiments. In the first setup, the effects of varying the size of the fast tier are analysed, whereas the second studies the impacts of an added caching layer. San-Lucas et al. state their simulation is able to reproduce the insight that a caching layer decreases the load to the underlying storage system.
While this work sounds promising at first, the presented tool lacks a number of important capabilities. First, StorageSim only supports read and write access. Creating and deleting files is not possible. Furthermore, there is no distinction between the behaviour of a read or a write call which seems at least doubtful. Also, while abstracting from a complex system is not only viable but necessary, evaluating data migration policies without considering the costs of data movement between tiers might not be reasonable.

Summary

In this chapter related works have been discussed. While they each have their strength they also have their weaknesses or shortcomings.
Desirable features include the simulation not only old hardware but current and future hardware as well. Also the possibility to analyse different hardware types and heterogeneous tiers is required to maximise the achieved performance of systems. Furthermore, models of data migration policies incorporating the cost of data movement as well as the support for complex hierarchies is wished for. Finally, the solutions should be sufficient to simulate HPC systems.

4. Design

This chapters details the requirements for a generic simulation tool.
First, it recalls the thesis goals and presents a basic approach consisting of several subtasks. Afterwards, the model objectives are specified. A special focus lies on outlining the key components of the two libraries and their purpose.

4.1. Modelling

The development of capable algorithms to determine the file movement in any system consisting of more than one type of storage hardware is a challenging task. A lot of separate circumstances influence the performance of an application on certain hardware. All those factors have to be considered and incorporated into the migration policies in order to maximise the improvements. As the according systems, which are to be enhanced, are usually operating on their original purpose, e.g. climate simulation, they are not available for thorough experimenting on their internal structuring. For that reason, the number of tools simulating these large-scale systems has significantly increased. Most of them are devised for a rather specific scenario. This leads to the consumption of a considerable amount of resources especially researchers. They are involved in creating their own fitting instruments, instead of using them for evaluation. Therefore, a more general simulator could establish a basis which can be adapted to individual requirements. The FFS library is designed to serve this purpose in the long run. Certainly, the implementation of a tool providing all necessary capabilities would have gone beyond the scope of this thesis. It is only a first step towards a generic and highly configurable emulation and simulation systems with the means to support HPC environments.

Recapitulating the thesis goals, discussed more in-depth in section 1.3, the main focus lies on creating the possibility to evaluate hierarchical storage systems as well as data migration policies to run on top.
Ideally, the results allow for more educated decisions regarding the acquisition of hardware for future systems as well as suitable tiering and data management strategies. Not only hardware devices have to be considered but all those

components adding additional limitations like transfer rates and access times. Furthermore, the functionality of a surrounding file system is essential to allow for an even slightly realistic tiering environment.
As discussed in chapter 3.1, there exist simulations taking traces of a discrete event simulator as an input and calculating the resulting outcome of certain operations on a specified system. However, this kind of simulation does no provide the possibility to just run the real applications on the system. This issue is being addressed with the approach presented in the following.
The broad assignment arising from the requirements can be split into smaller individual parts.

- Creating models for a variety of hardware devices for storing and transferring data
- Developing a flexible model for structuring the devices to simulate hierarchical storage systems
- Conceiving a simulation of file handling related system calls
- Outlining a simulation of a distributed file system running on the simulated system to provide a transparent view for user
- Designing an interface for different data migration approaches and eviction strategies
- Establishing an interface for file handling operations highly resembling the system call interface to enable simple adaptation of existing benchmarks and user applications to be executed on the simulated system
- Providing a strategy for permanent storage of not only the HSM system but also the file system content so it can be used again after the application finished which instantiated the system.

This list of diverse subtasks led to a layered design approach.
The bottom is formed by the underlying file system such as tmpfs, ext4 or ZFS. To avoid the unnecessary complexity of recreating already present functionality the actual data handling is accomplished by this file system.
The remaining functions are divided into two large layers. The first layer is covering the representation and simulation of the hierarchical storage system and necessary file system operations. The second consists of the realisation of the distributed file system and the data migration policies. This layering enables a modular library structure which can be more easily adapted.
The policy implementing is now independent of possible changes to the storage

model. Also, by separating the storage system simulation from the tiering part it can be reused more flexible for different projects. Those two layers find the corresponding implementation in two stacked libraries.

- **Fake File System Library** (FFSL) simulates the hierarchical storage structure and the file system atop. The application calls to this library are actually performed on the simulated system mimicking the behaviour of a real storage system. Hence, the FFS system is highly dependant on the speed of the underlying file system. Therefore, it is advised to mount it in a tmpfs to enable fast devices and connections.
- **Data Migration Library**(DML) is responsible for taking the application calls and translate them into the internal call equivalents and to manage the file distribution over the simulated HSM by using data migration policies.

An overview over the different library levels and their interaction is given in figure 4.1. A client application uses the API of the *Data Migration Library*(DML). The DML itself utilises the API of the *File System Simulation Library* (FFSL).
Finally, the FFSL makes use of the API of the underlying file system provided by the operating system.

To present the general execution order, an application is assumed which used the default file handling system calls such as `open`, `read`, `close`.
Originally, the file specified by "*path/to/file1.txt*" was opened using either `open(path, flags)` or `open(path, flags, mode)` depending on whether file creation was wanted or not. This call is now replaced with the according function of the DM library `dm_open_file(path, flags, mode)`.
Inside the DM layer the internal path for the external path "*path/to/file1.txt*" is looked up, if the creation flag is not set. Otherwise, the internal path is built as is explained in detail in chapter 6. The resulting path "*system/ssd_type/ssd1/file1.txt*" indicates where the file is stored inside the simulated system. It consists of the mount point of the simulated system "*system*", the device type "*ssd_type*" and device name "*ssd1*", followed by the file name "*file1.txt*".

4.2. FFS - Simulation of HSM

In chapter 2 a detailed overview of storage devices is given. To simulate not only single device instances but a large number organised in a hierarchical manner is difficult. One of the most challenging tasks is to determine which

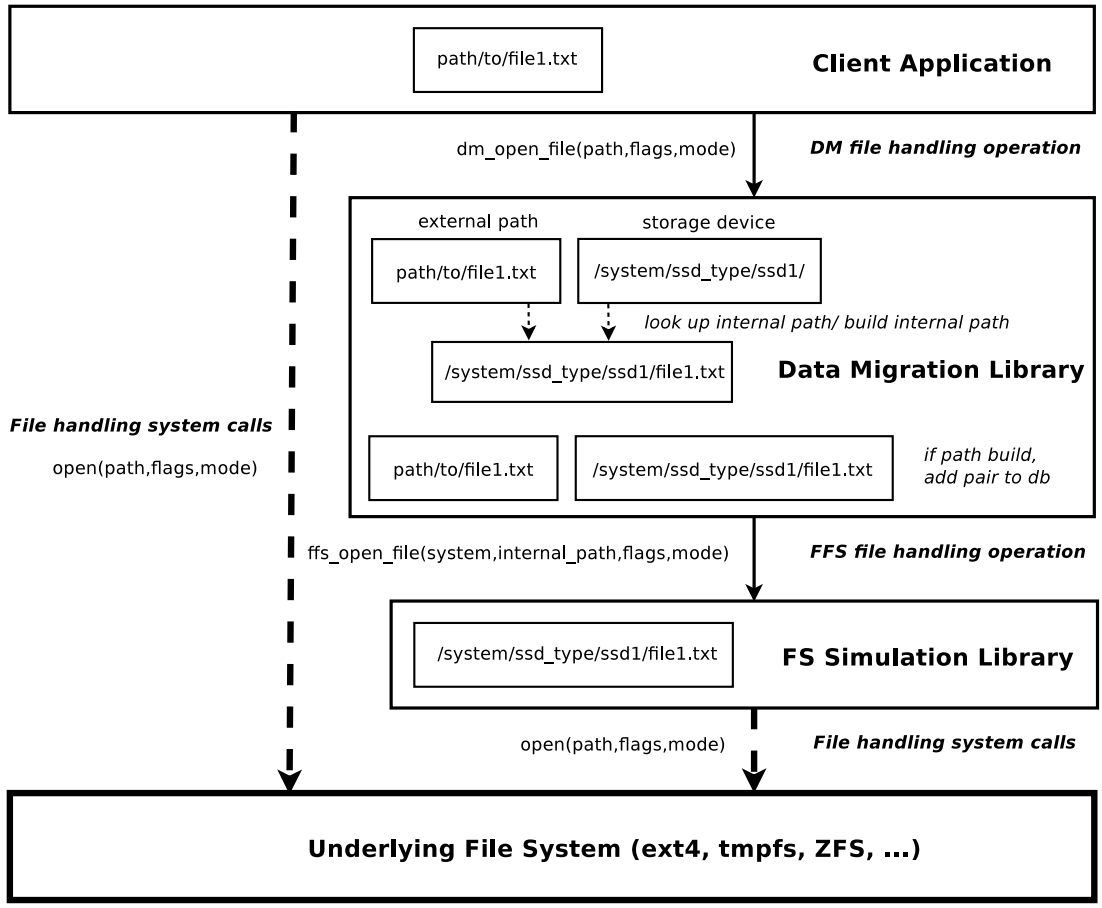

Figure 4.1.: Overview how the different libraries are layered on top of an underlying file system. The path an opening call of a user application takes throughout the layers is indicated by the arrows.

information are vital for a basic model. In the following, the presumed data path is explained, highlighting all relevant components on the way. The overall system architecture is that of a distributed file system as presented in figure 2.1. An application is running on a client, also referred to as compute node, which has no local storage. Therefore, it needs to access the storage via a shared network. The storage device is an instance of a device type and belongs to one tier. The FFSL functionality in its current state is listed below.

System Configuration

The main concern when designing the system for simulating HSM was to keep the interface as simple and small as possible while allowing a wide variation of

internal structuring.
At the same, adjustments to the individual usage scenario must be possible. Therefore, fine customisation of system specifications as well as the description of the actual storage system can be made in a configuration file. This enables neat initialisation and finalisation from a user's perspective. The only mandatory parameter is the location where to mount the simulated system. And while the characteristics and the system definition can be summarised in the configuration file, it is also possible to create the system step by step using the HSM and device handling APIs. The system set up can be exported to a new configuration file anytime.

Modelling Existing and Future Hardware

The goal was to enable a generic and flexible model of hardware devices. In order to supply insights regarding the acquisition of new systems the model needs to describe future hardware as well. Since it is not clear what exact capabilities prospective devices will have an abstract representation is required to also capture coming device behaviour. The information regarding a storage device can be split into two parts. On the one hand, there are features equal to all device of a certain technology like the latency or the capacity. On the other hand, the possibility of identifying an individual device and its statal details as the used capacity is important for a sensible model.
The first type of information is gathered in what is referred to as a *device type* whereas the latter is termed *device instance* (see also 5.1.2).
A device type is characterised by data regarding the mechanics which will be present in future versions. For example, an HDD(2.2.2) will always include a rotational disk and therefore have a noticeable access latency.
However, the scale of those information may decrease. Hence, the internal representation must be able to capture considerably smaller values. To fulfill this requirement the selected data types are suitable for a range between several hours even days and nano seconds.

Hierarchy Modelling and Tiering

In section 2.3 the emergence of HSM systems and their purpose have been defined. To simulate them including the concept of tiering is vital. As discussed in chapter 3 the current simulation tools are often not capable of supporting different devices types and none implements the possibility for tiers consisting of more than one device type.
In order to enable evaluation of the effect of mixed tiers the FFSL comprises not only homogeneous but also heterogeneous tiers. It also offers a generic

connection type to represent a variety of linking technologies like Infiniband or Ethernet.
The incoming connection and one or more device types form a tier. Thereby, complex hierarchies can be built by attaching the connection of a tier to the tier level above while also providing an approach to implement simple systems where every tier is directly linked to the client via a network.

4.3. FFS - File System Functionality

Apart from the storage structuring aspect the FFS library also includes the capabilities to simulate the file system's behaviour. The configuration defined by the HSM handler determines the procedures on top.
The file handler of the FFS is responsible for mapping the incoming calls to the corresponding system calls provided by the underlying file system.
It specifies an internal path format to manage the simulation of the accessing and manipulation operations as `open`, `close`, `read` and `write`.
An individual path format for the FFSL is crucial to keep the amount of knowledge about the system's implementation as small as possible. The calls provided by the API need to resemble the standard system calls to the greatest extent. Thereby, the changes which need to be made to the applications are reduced. The parameters for the said functions are the same as for a system call.
Albeit, additional information about the device characteristics are important in order to simulate a sensible behaviour. However, the FFSL does already know about the device and device type features which can be added to the API indirectly. By changing the accepted path format from *some/path/to/a/filename.txt* to *mountpoint_of_simulated_system/device_type/device_instance/filename.txt* the FFS library can now extract all necessary information because of the specified device type and device instance. As shown in figure 4.1, the conversion from external to internal path is performed within the data migration library. The combination of those two library layers offers the most comfortable usage. All calls to the DML take exactly the same parameter as the system call equivalent. The details are encapsulated in those two layers and discussed extensively in the following chapters. Another feature is to keep the built storage system as well as the contained files even after the application is finished. Everything necessary to resume running the system is summarised in the system's configuration file, data base file of the DM library and the directory structure holding the actual files. In order to match the performance of fast devices, the simulation is run in a tmpfs. If persistent storage is required the according structure is moved to a different mountpoint also specified in the configuration file.

Essentials for Distributed FS in DML

So far the FFSL only provides the support for operations of a local file system. To enable data migration on the hierarchical storage system a few additional functions are required. As highlighted in chapter 2.3 data migration policies manage the data distribution over storage tiers. For the necessary file movement, additional information must be specified when calling FFSL such as connection type. Either the file is relocated inside the same tier which is a probable scenario when supporting heterogeneous tiers or to a different tier. To keep the system model inside the FFSL as clear as possible the intra tier connections are handled only within the DML.
Besides, moving files also import and export functionality is granted which does not include any kind of reading or writing simulation and therefore is a much faster way to open and save files inside as well as outside the system.

4.4. DML - Data Migration on HSM

Like the FFS library the DM library also contains a wide functionality which can be divided into different aspects. By this separation the specific tasks become more obvious simplifying the understanding resulting in a clearer implementation. In the following, the key areas of responsibility are stated.

Distributed File System

The DM library is designed to work on top of the FFS layer which provides the simulated storage tiers on the on hand and the file system operations on the other hand. Atop this foundation the DML builds a file system distributing files over the system according to a specified policy in order to increase the reachable performance of the whole system. It mediates between the application expecting a system call interface and the storage simulation of storage and file system of the underlying FFS.
Therefore, it stores the external paths received from the application and translates them into internal paths as defined by the path format mentioned earlier in section 4.3. It also offers the possibility to store additional information regarding a certain file enabling more sophisticated migration strategies.
The DML relieves the user from getting into details with the internal representation of the storage system and allows to simply change the function names of the call while keeping the parameter just as they would be for a system call.

Policy Handling and Replacement Strategies

The data migration is managed by the policy handler. It supplies several algorithms how to distribute the data across the system. Hence, it decides the target of a movement execution. In contrast, the replacement strategy determines the source coming into play when a certain condition is met, e.g. when a device reaches its capacity limit. This is when the file to be written replaces an already stored file to a different device or tier depending on the migration policy.
Possible algorithms have been discussed in detail in chapter 2.2.2. At the moment, random and LRU are the implemented replacement strategies.
For a start, randomly choosing a device is used as a migration policy as well as moving the data to be replaced at the tier below. In a heterogeneous tier, the data is moved from faster devices to the slower ones.

System Configuration

Similar to the configuration file for the FFS library there also is a file specifying system characteristics. The replacement strategy and the migration policy are determined. Also, information such as the file to store the data base containing the mapping of external to internal paths are set.
Apart from that, the possibility is offered to state whether the system holding the actual physical files should be stored for longer than the run of the simulation. This enables either its reusing or cleaning of the surrounding system if it is supposed to be deleted when finalising the simulation.

Summary

In this chapter the essential decisions regarding the design of the two libraries have been presented and justified. The seperate components and their desired functionality has been discussed.

5. FFS - Simulating HSM & FS

In this chapter the internal design of the FFS library is presented. Furthermore, the requirements for simulating tiering and the requirements for the necessary HSM model are discussed. The key components and their interaction as well as their functionality is explained. Finally, a detailed insight into the implementation is given.

5.1. Internal Design

The FFS library, as pictured in figure 5.1, is located on top of a file system. It realises a flexible storage system model. From tape systems to SSDs all device types can be represented. The capabilities to simulate faster devices depends on the used file system below. Ideally, a RAM-FS such as tmpfs is used. Additionally, FFS provides file handling functionality as a basis for further data migration.

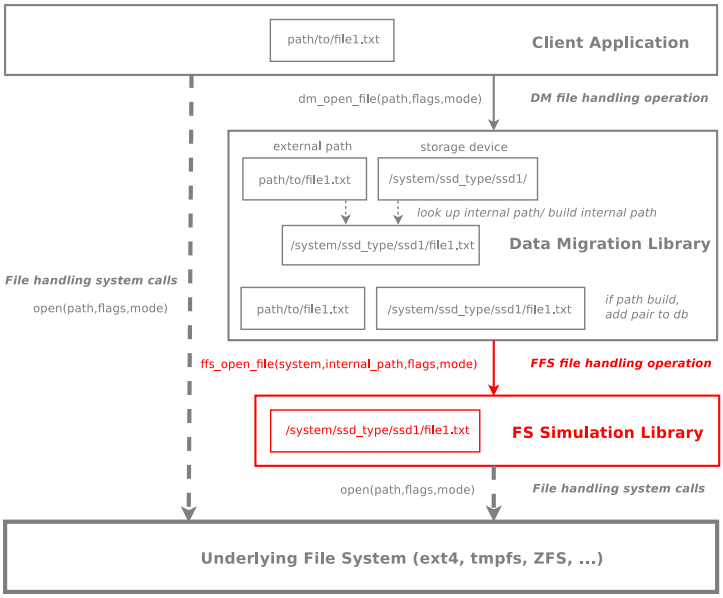

Figure 5.1.: The FFS library and its position in the library stack are highlighted in red.

5.1.1. Directory Tree Representation

FFS is developed as a behaviour simulation, an emulation. As mentioned before, this means it is able to actually provide the functionality to run the applications on it and to store the related files. The data can even be kept when the corresponding system simulation finished. This is possible by using the following concept.
The hierarchical storage system, explained in section 4.2, is realised by a directory hierarchy held by the underlying operating system's file system at the mount point declared in the configuration file. Every FFS system has a unique mount point which represents the root node of the directory tree as illustrated in figure 5.2. This directory structuring determines the internal path format which allows encapsulating information that is required inside the library but of no interest to an external user. By layering the device type as well as the device instance and including them in the path the simulation is able to perform the calculations defining the behaviour. The internal path format is specified as follows:
"mountpoint_of_system/device_type/device_instance/the_actual_file.txt"
Thereby, the library can look up the characteristics of the passed device type and the device instance without adding an extra parameter to its I/O interface. The translation of external file paths into internal paths is done by the DML. Hence, the application layer is not assumed to have sufficient knowledge of the internals to create a feasible path. It is able to call the DM with exactly the same parameters which are passed to the related system call.

In figure 5.2 the possibility is depicted how to use the internal path for identifying files inside the system but at the same the limitations of this approach. The resulting internal path is only a sensible identifier when files are not moved from one device to another respectively from one directory to another. The external path "/path/to/f1.txt" can be mapped to *"mountpoint/Seagate_SSD/SSD1/f1.txt"* but also to *"mountpoint/WD_HDD/HDD3/f1.txt"* which results in a conflict if the latter is moved to the SSD. Even *"path/to/a/different/f1.txt"* may be assigned to *"mountpoint/Seagate_SSD/SSD1/f1.txt"* as the proceeding directories in the external path are not considered for building the internal path. Simply adding the external directory structure to the internal path as in *"mountpoint/WD_HDD/HDD3/path/to/different/f1.txt"* makes the situation more complex and also reduces the reachable simulation performance as a potential deeply nested directory hierarchy needs to be created and moved around by the data migration policy. This problem can be solved by adding a unique identifier such as an increasing integer to the file name or omitting the file name completely and just work with the identifier. However, this requires

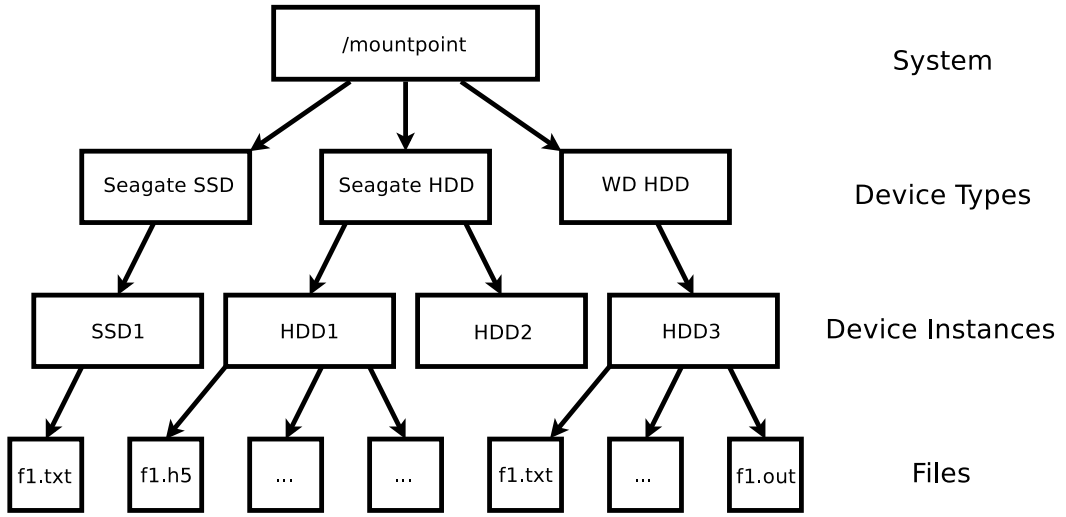

Figure 5.2.: Directory structure in the underlying file system realising the actual storing of the simulated system

storing the assignment of the identifier to the external path which increases the used memory especially for said nested files. As a starting point for a first implementation and in the explanations of the following sections, file names are the only way to address a file. Even so the representation of the external file system's structure does not contribute crucial functionality to evaluate tiering approaches or migration policies.

5.1.2. HSM Simulation

Below the decisions regarding the HSM design are presented.
An essential item is the representation of the different storage technologies. Summarised under the term *device type* are all those features applying to every device instance. Also, future hardware development needs to be captured in order to improve applications to perform well on prospective systems. Considered characteristics comprise the access latency, the read and write throughput and the capacity. Hardware failure and therefore fault tolerance are not part of the current design.

A *device* is an instance of a device type gathering information only applicable at a specific time because they are alterable. This includes the used capacity and identifier of the tier it belongs to. For a detailed model of storage hardware like HDDs or tape drives also the position of the last access is relevant to determine the behaviour when being accessed again. The moveable parts like the header and their positioning drastically affect the latency. So in consideration of

providing the possibility to distinguish contiguous from non-contiguous I/O operations the last access is important. However, to actually simulate non-contiguous accesses an additional model is required describing the way files are held internally by the device. At a representation level this fine-grained also optimisations such as hardware caches, burst buffers or the read ahead mechanism come into play. Such advisements alone would go beyond the scope of this thesis. Thus, there is no distinction whether an access is random or sequential.
The *connection type* is a further component for building a hierarchical storage structure. In contrast to the device types, there are no instances of a connection type. Individual links are not differentiated. They are characterised by a latency and their bandwidth. Neither network protocols nor network buffering is included in the model yet since they also impose a significant increase in complexity.

The last element for building an HSM system is the *tier*. Unlike the previous components, there is no concept of a type. Only tier instances are conceptualised consisting of a connection type representing the link coming in from the compute node. Tiers can either be homogeneous when composed of only one device type or heterogeneous if several device types are included. They have a list of all devices belonging to them.

The general design decisions are summarised. Every storage device is part of exactly one tier. Every tier has one type of incoming connection leading from compute node to the tier. Each tier can be composed of either one or more different device types.

5.1.3. FS-Functionality

In the following, the considerations are noted for building a file system simulation. FFS includes two different aspects of functionality. On the one hand, the imitation of the storage devices' behaviour and on the other hand the execution of the actual file related I/O calls. The supported functions are essential operations such as `open`/`close`, `read`/`write` and `create`/`delete`. Listing 5.1 states a general approach how to realise them. Acquiring a time stamp is requested at the beginning of each operation. Afterwards, the necessary validation of the input path is performed to fit the internal system design. This path in combination with the system's mountpoint is then used to construct the path for the underlying file system which is passed to the I/O system call. Thereby, the file system functionality is guaranteed. In order to mimic the hardware effects the time is computed it would take a specific device to accomplish the task. With the use of a second time stamp, the passed time is determined

as well as the remaining simulation time which the function still has to operate. At last the result from the system call is returned, e.g. a file handler or the number of written bytes. This procedure applies to all the supported file handling calls. The call in line 4 is therefore replaced with the relating system call. The calculations of the individual simulation times are presented below.

```
1  get timestamp t1;
2      check path format;
3      translate internal path to path for underlying fs;
4      I/O system call to this fs;
5      compute simulation_time;
6  get timestamp t2;
7      calculate delta_t = t2-t1;
8      calculate remaining_sim_time = sim_time - delta_t;
9  wait for remaining time;
10 return result from system call;
```

Listing 5.1: Basic procedure for the simulation of an I/O operation

The assumption made in the following formulas is that sending the issuing request to the storage hardware via the network does not have a notable size. Therefore, only the latency and the seek time of the device are relevant for computing the time it takes to open a file in this system (equation 5.1). The read simulation time is noted in equation 5.2. In contrast to opening a file, reading or writing a file does involve also the network bandwidth not only its latency. The sequence of the addends displays the data flow through the system which always includes the network latency as the first part. In order to support fast and possibly even more powerful hardware the time is measured in nano seconds.

$$T_{transfer} = \frac{blksize\ *\ count}{bandwidth};\ T_{read/write} = \frac{blksize\ *\ count}{throughput_{read/write}}$$

$$T_{opensimulation} = T_{latencx_{network}} + T_{seek} \tag{5.1}$$

$$T_{readsimulation} = T_{latency_{network}} + T_{seek} + max(T_{read}, T_{transfer}) \tag{5.2}$$

$$T_{writesimulation} = T_{latency_{network}} + T_{seek} + max(T_{transfer}, T_{write}) \tag{5.3}$$

The model above is very simple and does not consider more fine-grained details of the individual device types. Therefore, an I/O operation to a tape

does not yet include the time it takes to mount or unmount a tape nor the time spent waiting for the robot to receive a tape. Also, neither the parallel access enabled by providing several robots nor the occurring time penalty when trying to access a tape cartridge already mounted in a different tape drive are incorporated. A first improvement is to extend the model to comprise at least the mounting process which is shown in equation 5.4.

$$T_{readsimulation} = T_{mount} + T_{seek} + T_{read} + T_{latency_{network}} + T_{transfer} \tag{5.4}$$

5.1.4. Data Migration Support

Besides the pure file handling functionality support for data migration and the concomitant distribution of files has to be provided by the FFS library. The most important feature is to enable movements of files from device to another. Otherwise the layer above needs to open the file, read it into a buffer, create a file on the target device and write the buffer into the new file. Compared to a simple movement operation this induces unnecessary overhead which would distort any evaluation of migration policies. In order to simplify the system initialisation and finalisation, import and export functions are supplied. With their use, external files can be copied into the simulated system or can be stored outside the system.

5.2. Implementation

In the following, the approaches to realise the discussed design are elucidated. Detailed insights to the implementation are given as well as arguments justifying the choices. The library is implemented in C using the Glib library to provide additional data structures. As most of the applications that may be run on the simulated system are written in C it was an obvious choice to offer an interface which highly resembles the standard system call API. This way the necessary modification to the application are as small as possible. In addition, the underlying file systems are also implemented in C since it allows explicit memory management, induces only a small overhead, and therefore facilitates efficient code.

5.2.1. Component Interaction

In figure 5.3 the overall structure of the FFS implementation is depicted. The diagram is similar to a UML class diagram highlighting the interrelation between

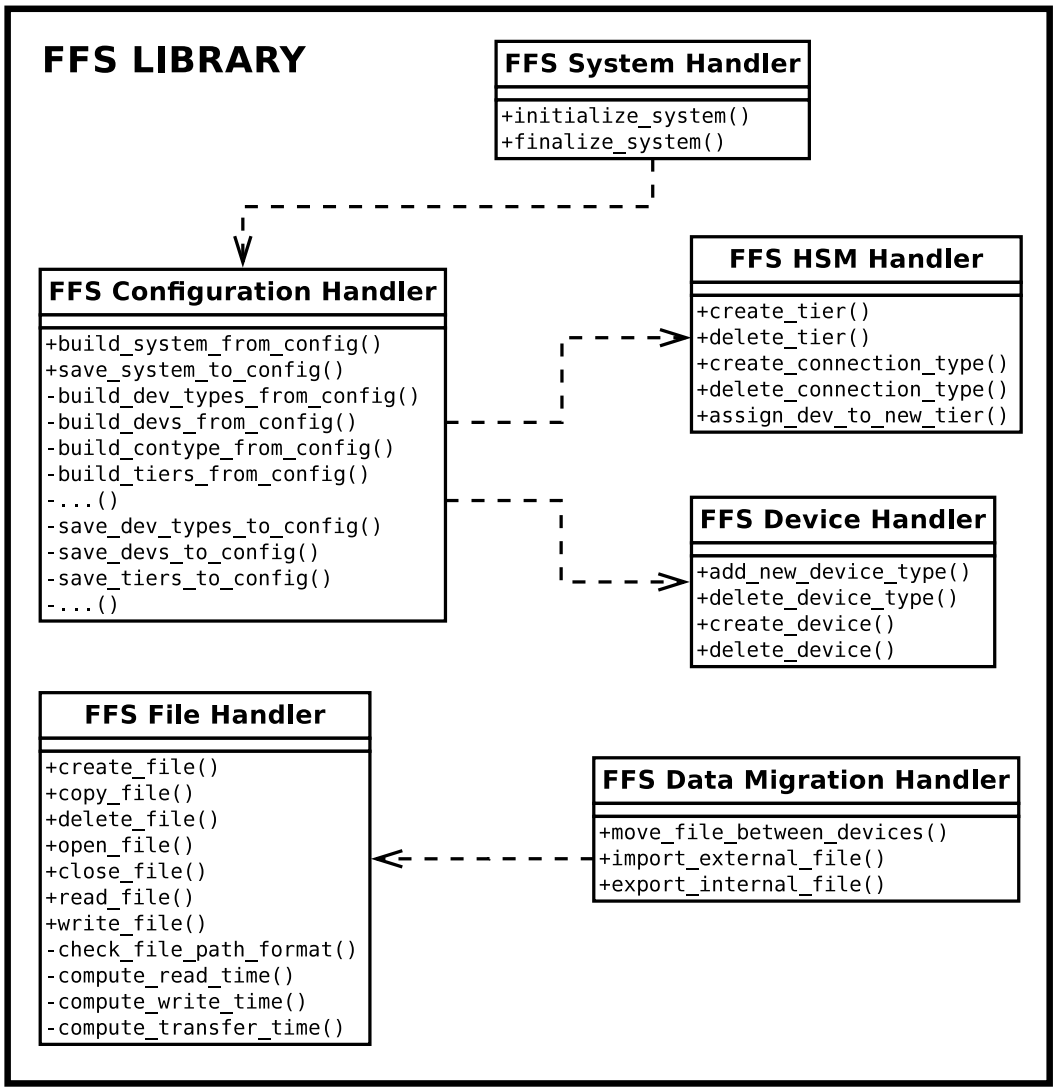

Figure 5.3.: Diagram showing the interaction between the different components of the FFS library

the individual components of the library. Modularity is achieved by separating the functionality in subparts. The final design present in the diagram is the result of several steps of redesigning and extending the former model. In the beginning, only the system handler as well as the I/O operations were present. Their simulation lead to a lot of code duplication containing the information about the device characteristics. These parts were gathered and formed the device handler. As a result, the file handling had to be remodelled as well and the management of internal file descriptors became necessary. At this point the functionality was split in the storage emulation and the file system aspect. The

former was then extended to include the hierarchical structuring of devices into separate tiers. The latter was enhanced to support data migration mechanisms atop the FFS library. The configuration handler was introduced to provide a comfortable solution to build and save system architectures.

In the following, the interaction between the different components is outlined. First, there is the system handler realising the initialisation and finalisation of the surrounding system. Then, based on a configuration file the configuration handler issues the creation of the particular device type as well as the related device instances. Additionally, the hierarchical layout is built by gathering these devices into tiers interconnected via connection types using the HSM handler. The file handling component comes into play when an application, usually the DM library, invokes an I/O operation. In the next sections, only a part of the functions is discussed in detail, e.g. the creation is explained while the related deletion is not shown. The whole FFS API is available in the appendix B.1.

5.2.2. System Handler

As mentioned above, the system handler is responsible for creating the basic system, e.g. allocating the necessary memory. The interface to the system handler consists of the following two operations. The initialisation only requires the path to the directory the simulated system be mounted. Since the simulation is highly dependant on the performance of the file system below it is recommended to run it in RAM which is pointed out by naming the parameter "mntpnt_tmpfs". The returned pointer to the system struct is used to identify the system throughout the simulation.

- *system_struct** **ffs_initialize_system**(const char *mntpnt_tmpfs)
- *void* **ffs_finalize_system**(system_struct *system)

In listing 5.2 the system struct is presented. The most crucial information is the mountpoint of the system as all calls to the file system below require the relating external path format which is build based on the specified mountpoint (see 5.2.4). `devs` holds all device instances which are described by the `dev_struct` explained in section 5.2.4. The creation of a device takes time because it is necessary to iterate over the whole array of devices and check if either the according device type is unknown or if a device of this type and name is already existing. As it is not possible to create two directories having the same path this case is permitted. The overhead for creating a device is acceptable since it is done only once while accessing a device is the far more relevant scenario which is fast as the array index functions as the unique device ID. The same

applies to the list of tiers `tiers`. In order to enable the deletion and re-usage of array entries, the highest ID is saved in `max_dev_id`. When an entry is deleted an empty space emerges and because the number of used devices `dev_cnt` is reduced the highest ID may not equal the device count. So, the purpose of `max_dev_id` is to minimize the number of elements any iterating operation needs to access while still considering all assigned IDs. Otherwise, the whole array, which can easily contain several thousand of entries when simulating an HPC system, has to be looked at.
Apart from that, the system struct does not only keep the overall capacity but also the currently used capacity to simplify the writing simulation. Thereby, verifying if the remaining capacity is sufficient to write a file to any device does not involve accessing every single device struct or even worse moving all files from one device to another infinitely. By including both capacities in the `tier_struct`, defined in Listing 5.9, also the validation for writing to a specific tier is more efficient. The `dev_struct` specifies the characteristics of a device that do not change over time, e.g. the capacity. These structs are managed by a hash-table identifying the entries by their name. Hash-tables are a data structure suitable when dealing with a lot of elements. They provide efficient and fast access especially when their maximum size is known. `dev_types` is accessed using the device name as the key while the device structs are the stored values. The connection types are organised in the same way.
Though the goal is to ultimately enable the simulation of large HPC systems adaptations can be made to allow the execution on consumer hardware. These options are stored in the `system_defines` struct. An excerpt of the settings is listed in 5.3. As the file system below realises the storing its limitations must be considered. The maximum name and path length are often crucial. Additionally, the memory requirements of the simulation are reducible by decreasing the maximum numbers of possible storage components, e.g. devices or device types. In order to avoid reallocation and therefore non-contiguous memory, the allocation is done in the initialisation process. So, to keep the memory consumption as low as possible the system specifications are customizable depending on the use case. Furthermore, the ordering of the elements in all the structs is an attempt to reduce the size of the used memory while increasing the cache locality [Ray17]. This is considered because of the high number of possible `dev_struct` instances. However, the implementation is a trade off between optimisation and readability which is increased by grouping related members.
The `fd_dev_mapper` structure is necessary for supporting the file system functionality of both the FFS and the DM library. How the file descriptors assigned by the operating systems' FS are mapped to internal file descriptors is explained more detailed in section 6.3.

```
1  typedef struct {
2  system_defines *sys_defines; /* system specifics */
3  dev_struct *devs;   /* list of all device instances*/
4  tier_struct *tiers; /* list of all tier instances */
5
6  GHashTable *dev_types;  /* Hashtable of device types*/
7  GHashTable *con_types;  /* -"- of connection types*/
8
9  char *mntpnt;       /* mount point of the tmpfs*/
10 fd_dev_mapper *map; /* map kernel fds to dev ids */
11
12 uint64_t used_capacity; /* used capacity in B */
13 uint64_t sum_capacity;  /* overall capacity in B*/
14
15 int tier_cnt;       /* number of used tiers*/
16 int dev_cnt;        /* number of used devices*/
17 int fd_cnt;         /* number of used fds*/
18 int max_tier_id;    /* highest tier id used */
19 int max_dev_id;     /* highest device id used */
20 } system_struct;
```

Listing 5.2: Struct for the system handler representing the general structure of the system

In Listing 5.3 a few of the possible specifications are shown. The default values represent the middle ground between small private storage setups and the HPC systems. The managing of these setting options is performed by the configuration handler described in the following section.

```
1  typedef struct{
2      int MAX_NR_DEVS_PER_TIER;   //1000
3      int MAX_NR_TIERS;           //50
4      int MAX_NR_DEV_TYPES;       //50
5      int MAX_NR_DEVS;            //5000
6      int MAX_NAME_LENGTH;        //256
7      ...
8  } system_defines;
```

Listing 5.3: Structure for the detailed system definitions with the default values annotated

5.2.3. Configuration Handler

The storage system to be simulated can either be created by an application using the operations provided by the HSM and device handler (see 5.2.4) or by importing a prepared setup from a configuration file. In Listing 5.4 the creation of a system is demonstrated without using an existing configuration file.
Knowledge of any path format is not required when utilising the individual operations. The necessary path transformations are encapsulated in the FFS library. Therefore, users only need to supply names. As the operations partly depend on already present structures the following orders needs to be maintained. In the beginning, the system structure is initialised at the passed mountpoint. Afterwards, the device type "DT1" and the connection type "CT1" are added to the hashtables `dev_types` and `con_types` respectively. Only then, it is possible to create a tier. Finally, the device is created based on the passed data type and tier ID. At this point, the directory "/home/user/example/" contains a directory named "DT1" which in turn includes the subdirectory for the device "DV1".

```
1  int t_id,dev_id;
2  system_struct *sys;
3
4  sys = ffs_initialize_system(``/home/user/example/'');
5
6  ffs_add_device_type(sys,``DT1'',3,5,8,13,21);
7  ffs_add_conncection_type(sys,``CT1'',42,101);
8
9  t_id = ffs_create_tier(sys, ``TR1'', ``DT1'',
     ↪ ``CT1'');
10 dev_id = ffs_create_device(sys, ``DV1'', ``DT1'',
     ↪ t_id);
```

Listing 5.4: Creating a system without the configuration handler illustrating the order they need to be executed in.

To avoid creating the system manually the configuration handling component of the FFS library offers two functions;first the possibility to save an existing system `*system` to a file specified by the passed path and second a way of building a saved system from a configuration file.

- *int* **ffs_save_system_to_config_file**(system_struct *system, const char *path_for_new_file)
- *int* **ffs_build_system_from_config_file**(system_struct *system, const char *path_to_config_file)

To implement the configuration files the key files of the Glib is used. Such a key file contains key value pairs which can be organised in groups. In Listing 5.5 the shortened first part of an example configuration file is presented.

```
1  # === SYSTEM SPECIFICATIONS ===
2
3  [system]
4  Current local time and date=Mon Jul 17 15:57:21 2017
5  # --- System defines ---
6  MAX_NR_DEV_TYPES=1000           // int
7  ...
8  # --- System characteristics ---
9  mntpnt=/home/user/testsystem/   // string
10 dev_type_cnt=7                  // int
11 sum_capacity=2000000000000      // uint64 [B]
12 # list of device ids
13 dev_ids=0;1;2;3;...;999         // int;int;int
14 ...
15 # --- Following section lists all device types ---
16 [dt_i]
17 dt_name=SSD_BARRACUDA           // string
18 r_ltncy=8500000                 // uint64 [nsec]
19 r_thrghpt=156000000             // uint64 [B/s]
20 capacity=2000000000             // uint64 [B]
21 ...
```

Listing 5.5: This listing shows an extract of a configuration file where '//' mark annotations that are not present in an actual configuration file.

Comments inside the key file are denoted as a hash and ignored by the parser. Square brackets indicate a group which can contain one or more key-value pairs. Additional remarks specify the type of the value to facilitate the usage. This is also the reason for storing the creation time and date making the association between the configuration file and a certain simulation run easier.
While it is unlikely to exceed the integer value range with counting the number of devices both the device type and system capacity require a larger data type. As the latency of newer technologies such as NVRAM and NVMe now ranges between 1 to 10 microsecond the representation for device latencies has a nanosecond resolution. At the beginning of the key file, the system specifications are managed including a list of all assigned device IDs. Afterwards, the device and connection types are itemised followed by tier definitions along with all

device instances. Thereby, the entire system architecture can be rebuild. In the appendix B.1 an example of a complete configuration file is given.

5.2.4. Device Handler

The management of all the information required by the file handler to simulate the I/O operations is performed by the device handling component. One important concern was to keep the amount of data stored in a device as small as possible as the long-term goal is to enable a simulation of HPC systems which often contain thousands of devices. Therefore, the information associated with a certain device is classified as either time dependent or non-dependent. The time independent device features like the capacity or the writing throughput are collected in the `dev_type` structure illustrated in listing 5.6. Those characteristics apply to all instances of a certain storage device for example an SSD such as the Barracuda by Seagate. Consequently, it is sufficient to store them once and make them accessible throughout the library. For this reason the `system_struct` includes a hashtable to provide access via the name of the device type. A new type can be added to the system by using the function `ffs_add_device_type`.

```
1 typedef struct{
2 char *dt_name;     /* name of device type */
3 uint64_t r_ltncy; /* read latency in nano seconds */
4 uint64_t w_ltncy; /* write latency in nano seconds */
5 uint64_t r_thrghpt; /* read throughput in B/s*/
6 uint64_t w_thrghpt; /* write throughput in B/s*/
7 uint64_t capacity;  /* storage capacity*/
8 } dev_type;
```

Listing 5.6: Device Type structure containing the device information which does not change over time

The individual values are passed as a parameter. They are either read from the specified configuration file or provided by the application above. In order to represent the wide variety of device types from tape drives to NVRAM that are possibly present in an HPC system with one generic structure the data types had to be chosen carefully. The largest value an unsigned integer 64 bytes long can store is 18,446,744,073,709,551,615 or approximately 18 x 10^{18}. This is sufficient to describe capacities from one byte to 18 Exabyte or latencies and time intervals from one nanosecond to roughly 213 days. Thereby, current system and at least near future systems can be properly modelled. An approach

how the enormous RAM requirements of such a simulation can be reduced is discussed in chapter 8.
However, there are several assumptions in this model that prevent a fine-grained simulation. As an example, when simulating tape systems the effects need to be considered that arise by mounting and unmounting drives via a robot as well as those which originate in the behaviour of the related tape library. A first improvement could be made by either appending additional features to the structure such as an average mounting time or by extending the calculation to incorporate the simulation model discussed in chapter 2.2.2.

- *int* **ffs_add_device_type**(system_struct *system,
  const char *new_type, uint64_t r_ltncy, uint64_t w_ltncy,
  uint64_t r_thrghpt, uint64_t w_thrghpt, uint64_t capacity)

- *dev_id* **ffs_create_device**(system_struct *system,
  const char *name, char *type, tier_id t_id)

The parameter list of `ffs_create_device` indicates an essential premise. Every device requires a type and is integrated in exactly one tier. It is not possible to create independent devices as FFS is designed to represent tiered storage hierarchies.

In the `dev_struct`, detailed in Listing 5.7,the information is held regarding the present state of a device. The current implementation only includes the used capacity and the ID of the tier. As previously mentioned, the differentiation between contiguous and non-contiguous accesses does not only increase the accuracy of the simulation but also the complexity of the model.

```
1 typedef struct {
2 char *d_name;            /* device name */
3 char *d_type;            /* device type*/
4
5 uint64_t used_capacity; /* used capacity in B*/
6 int tier_id;             /* device part of this tier */
7 } dev_struct;
```

Listing 5.7: Device Struct representing the time-dependant state of a device

5.2.5. HSM Handler

Modelling storage hierarchies does not only involve device but also their interconnection. In contrast to the previous component there is no distincion between

individual links. Connections exist only as a type as the current model does not include any information differing for specific link. The `con_type` consists of the name to identify the type, the latency and the bandwidth.

- *int* **ffs_add_connection_type**(system_struct *system, const char *new_type, uint64_t latency, uint64_t bandwidth)

```
1 /* struct representing connection types*/
2 typedef struct {
3 char *ct_name;  /* name of connection type */
4
5 uint64_t ltncy; /* latency in nano seconds */
6 uint64_t bw;    /* bandwidth in Bit/s */
7 } con_type;
```

Listing 5.8: Connection Type Struct

The hierarchy of the storage devices is realised by grouping devices into tiers. As discussed in chapter 3, the existing simulation tools do not support heterogeneous tiers. However, using different device types inside one tier and linking them with a faster and therefore more expensive connection is a scenario which will become more prominent. The reason lies in the evolution of RAM based technologies like NVRAM. Although, this non-volatile memory fills the gap between the permanent storage of SSDs and the high throughput and low latency of RAM its price makes it not yet deployable for large scale tiers.
Therefore, the concept of heterogeneous tiers is suggested for example by the DAOS developers of Intel (see also 7.2). The idea is to gain the advantages of the reduced latency while providing larger capacity utilising a cheaper technology. This combination is backed by capable connections to enable fast data migration between the devices in one tier.
In FFS, tiers are implemented by the structure presented in Listing 5.9 consisting of a connection type and the device types of included devices. A flag indicates whether more than one device type is present or not. In case of a homogeneous grouping, a tier is created by passing the related system pointer, the name, the device type and the connection type of the incoming link. The according struct is added to the global tier list of the system handler. In order to establish a heterogeneous arrangement, `NULL` is passed as a device type to `ffs_create_tier`. Hence, later on there is neither a limitation concerning the supported types nor the number of different types. The necessary information to simulate a device access is supplied indirectly through the list of contained device IDs which are uniquely assigned throughout the system.

- *tier_id* **ffs_create_tier**(system_struct *system, const char *name, char *dev_type, char *con_type)

- *int* **ffs_assign_dev_to_new_tier**(system_struct *system, tier_id old_t_id, tier_id new_t_id, dev_id d_id)

```
1  typedef struct{
2  char *t_name;       /* name of specific tier*/
3
4  dev_type d_type;    /* device type of the tier */
5  con_type c_type;    /* connection leading to tier */
6  dev_id dev_ids[XY];         /* list of dev ids*/
7
8  uint64_t t_used_capacity; /* used capacity in B*/
9  uint64_t t_sum_capacity;  /* sum capacity in B*/
10
11 int dev_in_tier_cnt;     /* number of devs in tier */
12 int is_homogenous;       /* only one device type? */
13 } tier_struct;
```

Listing 5.9: Tier Struct

The summarised capacity of all the contained devices simplifies the verification of whether further data can be stored. Despite the diverse possibilities to adapt a tier, a few requirements need to be fulfilled. A tier may consist of any number of devices, limited only by the maximum set in the configuration file, and may include an arbitrary number of device types but there has to be exactly one type of connection coming in from the compute node. However, this does not impair the design too much as there is no reason to group elements into a tier if they do not share even one feature.
As the connection type imposes the most significant restrictions regarding the access possibilities it is more reasonable to organise devices with different linking into separate tiers. Nevertheless, the intra-tier interconnection is completely adaptable by specifying the link as a parameter of the `ffs_move_file` operation of the file handler which is elucidated in section 6.3. Furthermore, `ffs_assign_dev_to_new_tier` offers a way for reorganising the system structure of already existing elements without losing the contained files. Also, deleting a connection type is only permitted if it is not part of any tier. The removal of a tier, however, results in the deletion of all devices belonging to it. The according functions are listed in the complete FFS API in the appendix B.1.

5.2.6. File Handler

Implementing the file system functionality imposed several challenges. The most important decision was how to intercept the I/O calls and write the files to the simulated system. Instead of trying to actually interrupt system calls a simpler approach is to define a new API highly resembling the system call interface. This way the usage for application developers is easier and more intuitive than a complete redesign.

- `int ffs_open_file(system_struct *system, const char *path, int flags,mode_t mode)`
- `int ffs_close_file(system_struct *system, int fd)`
- `size_t ffs_read_file(system_struct *system, int fd, void *buf, size_t cnt, off_t offset)`
- `size_t ffs_write_file(system_struct *system, int fd, void *buf, size_t cnt, off_t offset)`

However, this introduces the requirement to assign internal file descriptors otherwise all operations based on the passed fd can not be simulated.

As stated in Listing 5.2, the system handler holds an array of `fd_dev_mapper` structures. The array index represents the internal file descriptor while the array elements contain the file descriptor assigned by the underlying file system as well as the device ID on which the file is stored.

```
1 typedef struct {
2 int     system_fd;  /* kernel file descriptor */
3 dev_id  d_id;       /* index of global device list*/
4 }
```

Listing 5.10: Mapping structure

In Listing 5.11 the implementation of the open and the write call is presented as an example to illustrate the internal organisation of the file handler. When opening a file the path is the most important parameter. The file handling component of FFS retrieves the device ID related to the device specified inside the path. The individual path elements are the recombined with the mountpoint information to build the internal path which is passed to the `open` system call. The returned file descriptor is store into the mapper along with the device ID. The array index is then used as the internal file descriptor and returned to the application. The following I/O operations, e.g. `write`, take it as a parameter which is then replaced by the system fd for the according system

calls. Afterwards, the simulation is performed based on the device ID contained in the mapper. Finally, the return value of the system call is forwarded to the layer above.

```
1  open(path,...)
2  {
3      get_device_name_from_path(path);
4      dev_id = get_id_for_dev_name;
5
6      build external path;
7
8      sys_fd = open(ext_path,...); //system call
9
10     mapper[int_fd].system_fd = sys_fd;
11     mapper[int_fd].d_id = dev_id;
12     return int_fd;
13 }
14
15 write(int_fd,...){
16     sys_fd = mapper[int_fd].system_fd;
17     dev_id = mapper[int_fd].d_id;
18
19     cnt = write(sys_fd,...); //system call
20
21     get characteristics of devs[d_id].d_type;
22     simulate_writing(write_latency,write_throughput);
23     return cnt;
24 }
```

Listing 5.11: Implementation of file descriptor management and I/O operations

Apart from the usual I/O operations FFS also offers additional functionality to simplify data migration in the library atop. By importing and exporting files with `ffs_import_file` and `ffs_export_file` the transition between the surrounding file system and the simulation has become easier as no read or write simulation is performed. Calling `ffs_move_file` allows to move the file between different devices and tiers while the underlying file system moves the file as well. It is not copied which reduces the storage requirements and the overhead. Specifying a connection type enables to represent interconnections in the storage system besides the incoming link included in the `tier_struct`.

- *int* **ffs_import_file**(system_struct *system,
  const char *external_path, const char *internal_path)
- *int* **ffs_export_file**(system_struct *system,
  const char *internal_path, const char *external_path)
- *int* **ffs_move_file**(system_struct *system, const char *src_path,
  const char *dest_path, int src_dev_id,
  int dest_dev_id, const char *con_type)

Default Types

To simplify basic testing and benchmarking of the storage simulation several default types are included in a config file.

- NVRAM: Optane - Intel 3D X Point
- SSD: Nytro - Seagate
- HDD: Barracuda - Seagate
- Tape: LTO - capacity assumed 6 TB with throughput of 300 MB/s [Lü16]
- Ethernet 10 GBit
- Infiniband - EDR

Summary

This chapter gave detailed insights into the internal design of the FFS. The separate components and their responsibilities have been presented. Finally, the implementation and the occurred problems have been discussed.

6. Data Migration Library

This chapter focuses on the internal design of the DM library and its interaction with the application on the one hand and the FFS library on the other hand. The requirements to support data migration between different and inside one HSM tiers are discussed as well as the basic requirements to enable a variety of metrics. The decisions related to the aim to provide a user-friendly interface and a sufficient performance are outlined.

6.1. Overview

The file movement in complex typically layered storage systems is determined by data migration policies. They incorporate detailed information of the storage hierarchy in order to improve the utilisation. As the development of the storage technologies continually introduced new capabilities, it became increasingly difficult to identify key elements affecting the overall performance. Simulations aim to support the search for suitable solutions. The FFS library provides the possibility to emulate the behaviour of hierarchical storage systems. Atop of it, the DM library offers additional file system functionality to enable a variety of data migration strategies.

Just as the layer below, DML is designed to establish a generic foundation to be adaptable to individual requirements. In addition, the interface to the application layer above is oriented on an intuitive and simple usage. In figure 6.1 an implementation employing the opening call of the DML and its way through the libraries is illustrated. The partitioning of the functionality into separate layers simplifies the adjustments in one library by avoiding a complete redesign for minor changes.

6.2. Design

The internal structuring of the DML tries to suffice the design goal of a flexible and modularised library. However, as this is a challenging task a number of iterations of redesigning were necessary as was the case with the FFS library. This resulted in an extensive layout whose implementation is not yet fully

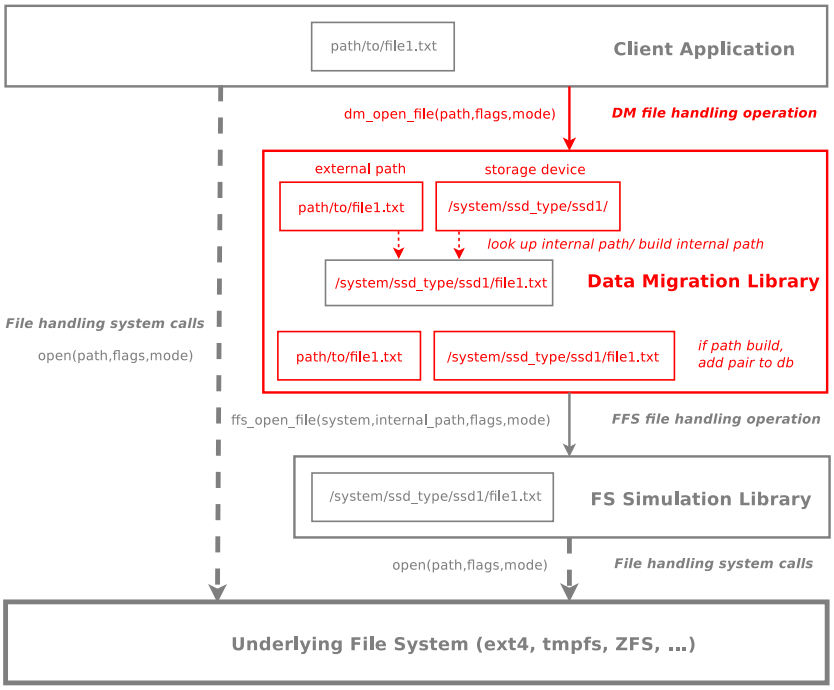

Figure 6.1.: Location of the Data Migration Library inside the Library Stack

realised. Nevertheless, its composition should support several different policy approaches. The DML consists of the following components.

- **System handler**: In order to keep the adaptation to the application as small as possible, the system initialisation is mainly handled in the DML itself. A configuration file is used to manage the customisation to specific requirements. Such an instance may be the request for persistent and recoverable storage of the whole system which is accomplished by combining the information of the FFS configuration file and the related physical directory structure. Thereby, the storage hierarchy, its simulated behaviour and the stored files are reusable. By integrating the mapping between the external file system structure and its internal equivalent the data migration and file system functionality can be restored as well.
- **DB Handler**: A key feature of the DM library is the management of the data distribution across the simulated devices. So as to achieve a consistent state, the file movement needs to be logged. This requires not only a highly reliable infrastructure but also the means to support large-scale HPC applications which access thousands of files possibly in a parallel manner. Although a parallel implementation is not viable within the limits of this thesis, the general layout needs to incorporate those

aspects to avoid completely redesigning the library several times. The concept of key-value stores seemed most suitable to meet these needs. This choice is discussed in section 6.2.

- **Replacement Handler**: When enough storage space of the favoured device type is available data migration is not necessary. This changes when the selected device to save a certain file has reached its capacity limits. While writing a file, its priority is usually higher than those of files which have not been accessed. However, determining which is the least valuable file in a device or a tier is intricate. Depending on the use case different factors are to be considered. Often, the file least recently or frequently used is evicted. Though this is a common scenario, there might be contexts in which a rarely used file is vital for an exceptional application behaviour and a fast access has to be guaranteed. This is a demand the library can neither derive from the access patterns nor from creation or modification times. So, in order to support this type of request the replacement strategy needs to offer more than just a heuristic based on past I/O profiles. Besides displacing files from a device, the replacement handler also manages the reloading of files. For example, when opening a file on a slow device the arising question is whether to move this file to a faster device again and if so in which situation.

- **Policy Handler**: As a policy is a general expression that sometimes includes the definition of one of more replacement strategies, in the following subclasses are described. In contrast to an eviction scheme, they do not determine the subject of the movement but the target location. Depending on the storage structure and its usage there may be several decisions to make. For a first approach, the placement of files across the complete system and the distribution inside one tier are considered. Should a file be located on the fastest tier in terms of device and network latency or on a device with a high throughput? Is an answer applicable to all kinds of files or it is more sensible to distinguish between data and metadata? Though these issues can be addressed by an analysing tool, it is still necessary to enable specifications by the user. At the same time, the complexity of the interface should not increase to a level where it takes more time to understand the internal design than to implement one's own solution.

- **File Handler**: Apart from the management of data movement, the DM has to supply an interface for I/O operations. Therein the mediation between the application and the FFS library is performed. The API is orientated on the system call interface and provides similar functionality such as `open`,`read` and `write`. They use not only the same parameters but

realise also the expected behaviour and return values, e.g. when opening a file the internal file descriptor is handed back.

In Figure 6.2 the concept for the internal structure of the DML is depictetd. Besides individual components also their dependencies are illustrated. The bold arrows highlight the interface to the application above. Only the system handler responsible for the initialisation and the file handler interact directly with the layer atop. The remaining elements are used for internal structuring and modularisation.

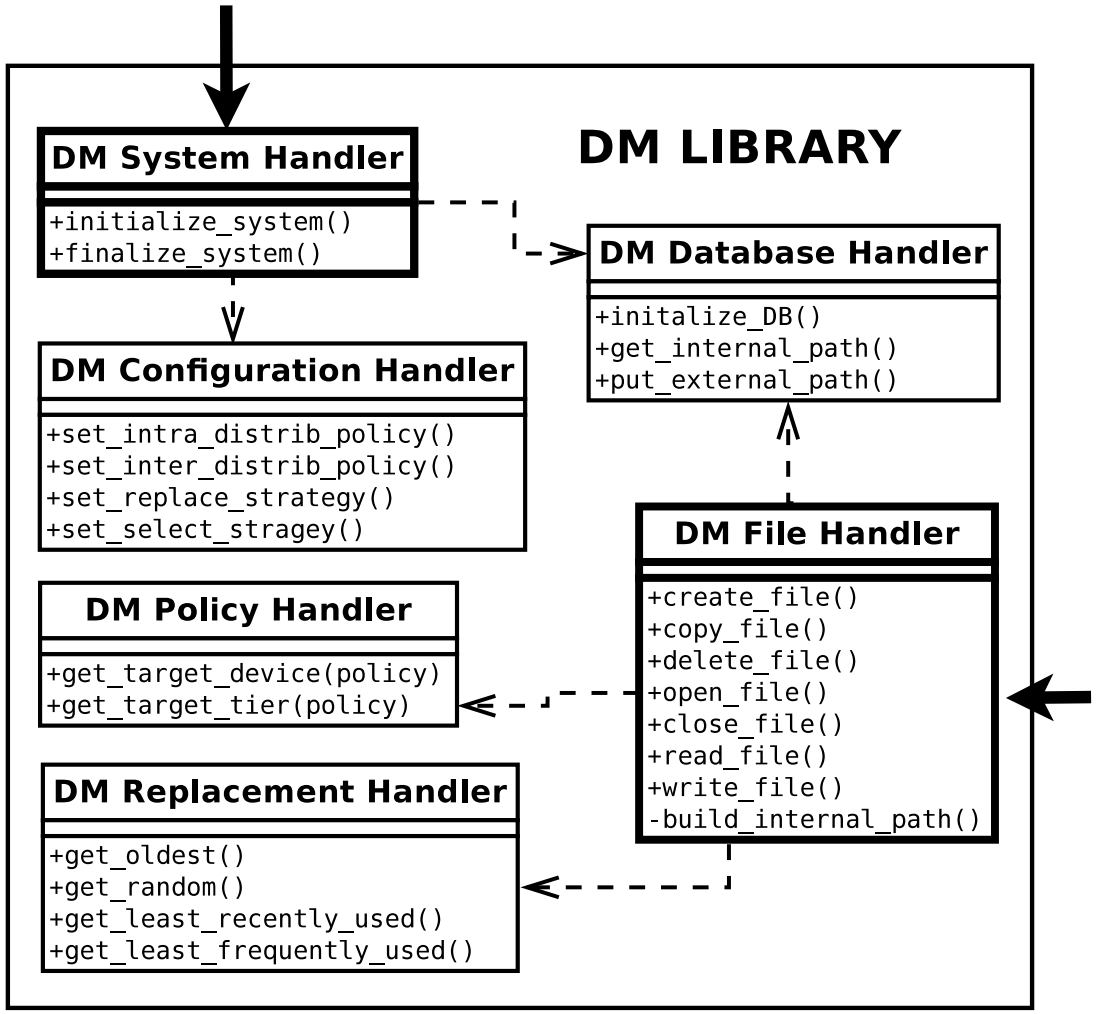

Figure 6.2.: Diagram illustrating the internal design of the DML. Printed in bold are the components directly providing an interface to the application.

DB Handler - Key Value Stores

The most important features to guarantee in a file system are the consistency and the availability of the stored files. In order to simulate data migration strategies the DM library needs to keep track of the movements. More specifically it has to save which external path belongs to which file inside the simulated system. A file system, especially in an HPC environment, is exposed to hundreds of thousands individual files. This induces special requirements for the resilience of the used data base. First, the number of records which are held at the same time needs to be extraordinary large. Additionally, as performance is crucial for HPC systems, the overhead of the data base should be as small as possible. Furthermore, the support of parallel access also important. Without the later the DM will only be able to simulate sequential I/O and while data migration strategies might be useful for such a system the core task is to represent parallel distributed systems. The amount of included elements and the demand for persistent storage rule out RAM based approaches. One of the most popular data base concept is the *relational model* of data. It includes a set of tables consisting of columns and rows. A row is identifiable via a unique key and also referred to as a record. While the relational model is suitable for a lot of use cases it is not sufficient as a foundation for a distributed file system. Its fixed data base scheme does not allow to include additional information just for a specific record. It requires a complete redesign.
A different way to store information is using an associative array which is an array type allowing non-numerical values as its unique keys. The database concept on top is called *key-value store*. It provides the possibility to store records of different structures. Thereby, flexible adaptations are enabled and the use of wildcards for missing values is avoided. Key-value stores also support large datasets and highly parallel access. A prominent example is Google's LevelDB. However, as examined by Pillai et al. it exposes several vulnerabilities on today's file systems such as ext4 and btrfs. It is prone to silent errors, data loss and failed reads and writes [PCA+14]. It is also stated that the implementation exhibiting the least weak spots and the best crash recovery is *Lightning Memory-mapped Database (*LMDB*)*. For this reason it was chosen as a basis for the DML. The LMDB is based on a binary tree and maps the entire database into the memory. Therefore, memory allocation is not required when fetching data and also no additional page caching is necessary. This allows for high performance and memory efficiency. The LMDB uses a transactional scheme and guarantees the ACID properties while supporting concurrent access. By relying on the copy-on-write mechanism data pages can not be overwritten while they are still in use.

6.3. Implementation

In Figure 6.3 the interplay of the two libraries and their relation to the application is highlighted. The user initialises the DML system handler which issues the following initialisations. First, the configuration handler sets the policies according to the passed file. Afterwards, the database handler issues the LMDB instantiation. At last, the DML system handler initialises the FFS instance providing the storage system emulation. The internal call structure of the FFS is described in subsection 5.2.1. Inside the application the standard I/O operations have been exchanged with the operations of the DML file handler which is responsible for the data movement between the separate tiers conforming to the selected migration policy and replacement strategy. In section 6.3 the opening of a file is explained in detail. Additionally, the procedure for writing a file is outlined. The replacement handler chooses the file which needs to be moved and passes this information to the data migration handler of the FFS library realising the actual movement operation. In case the file resides already on the right device the DM file handler directly calls the FFS file handler handing over the standard system call parameters.

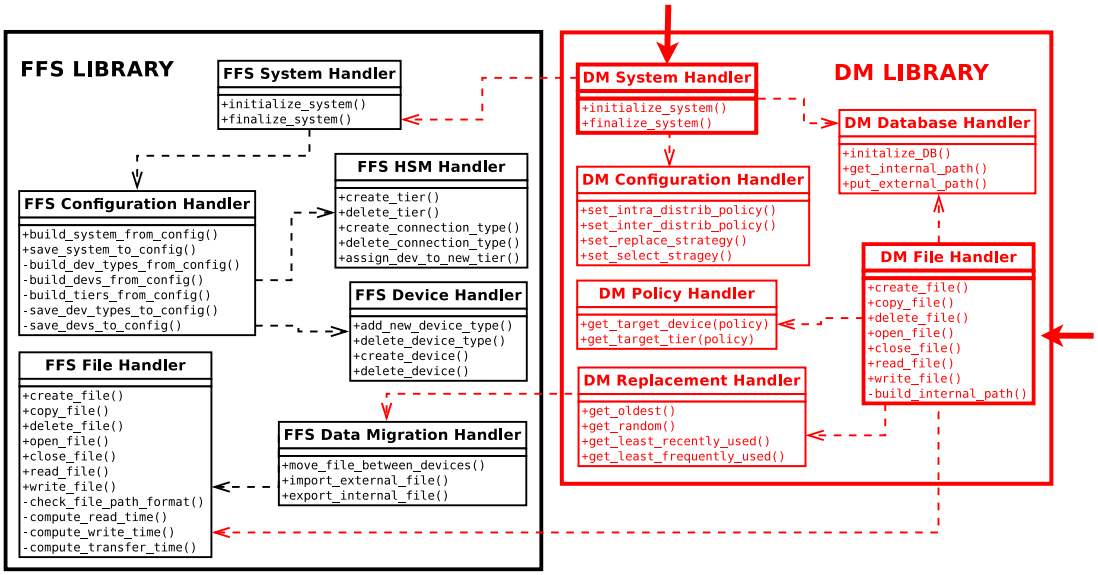

Figure 6.3.: Diagram of the interaction between the components of the DM library and the FFS library

System handler

The system handler manages the setup of the environment required for the data migration and also issues the initialisation of the FFS instances below. Besides the mountpoint specifying where the directory tree of the FFS is located, the path to the configuration file as well as an existing directory for the database are passed as parameters to `dm_initialize_system`. In contrast to `ffs_initialize_system`, it returns only an integer indicating whether the initialisation has been successful. The system structure presented in Listing 6.1 is exclusively used inside the DM library in order to minimise the interaction between the application and the library internals.

- *int* **dm_initialize_system**(const char *mntpnt_tmpfs, const char *config_file, const char *db_dir

The DM system is represented by the following structure. It contains the necessary information for the interaction with the LMDB as well as the FFS library. The FFS system structure, providing access to the emulated storage system, is explained in Listing 5.2. All interaction with the LMDB is performed with the use of the `db_struct` which is presented later on. The directory holding the database file and the related locking file is specified by a its path. Furthermore, the DM stores which policies are set for data migration between different tiers as well as inside a tier and which eviction and selections schemes are chosen.

```
1  typedef struct{
2      system_struct *ffs_system;
3
4      db_struct *db;
5      const char  *db_dir;
6
7      int          inter_tier_policy;
8      int          intra_tier_policy;
9      int          replacement_strategy;
10     int          selection_strategy;
11     int          distribution_strategy;
12 } dm_system;
```

Listing 6.1: DM system structure where the prefix MDB indicates the membership to the LMDB.

Database Handler

The database handler manages the interaction with the LMDB instance. The required pointers are gathered in the `db_struct` as shown in Listing 6.2. In order to use the database an `MDB_env` environment has to be created. When opening the environment the directory to save the database file and the according locking file has to be specified. As the LMDB is transactional the next step is to begin a transaction. Within the transaction a database instance `MDB_dbi` is created which is used for all following accesses. While this setup is sufficient to perform the adding, retrieving and deleting of a key-value pair more complex operations necessitate the concept of a cursor.

```
1 typedef struct{
2     MDB_dbi      *dbi;
3     MDB_env      *env;
4     MDB_cursor   *cursor;
5 } db_struct;
```

Listing 6.2: DM system structure where the prefix MDB indicates the membership to the LMDB.

At the moment, the database holds only strings as keys and values namely the external path and the internal path for a file. However, the usage of a key-value store enables flexible adaptations to the data entries. So additional information, e.g. the access frequency, which are vital for complex replacement strategies can be appended without changing anything else besides a specific entry.

Replacement Handler

The replacement handler comes into play when an I/O operation, e.g. a write call, cannot be executed because the target device has reached its maximum capacity. It is responsible for determining which file should be moved to provide free space to write the current file. As discussed in section 2.2.2, there are several strategies how to find the appropriate file to move which have a severe impact on the system's performance. Choosing the best replacing mechanism for a specific environment is a challenging task because it requires providing a suitable evaluation scheme for the value of a file. The optimal one evicts the file which will be accessed furthest in the future. This information, however, can only be acquired by performing test runs of a specific application on a given storage system which is not viable for most HPC applications. Therefore, the common approach is to use past access patterns as an indicator of future behaviour. Using the file age or the file size are rarely sensible features to

predict prospective I/O as evaluated by Arpaci-Dusseau [ADAD15]. The DML implementation currently contains the following strategies; selecting a random file and picking the file which is least recently used given the directory path of a certain device simulated by the FFS.

- `char *get_random_file(char *device_path)`
- `char *get_LRU_file(char *device_path)`

For realising `get_random_file` the number of files on the passed device is counted. Afterwards, `rand` is used to generate a random number which is then mapped to a value in the range between 1 and the current file count by applying the modulo operator. The implementation of `get_LRU_file` relies on the features of the underlying file system. The least recently used file is determined based on the time of the last access(`atime`) contained in the structure of `stat` system call. Whether the access time is updated regularly, depends on the actual file system making the current implementation of the DML prone to erroneous behaviour. This situation is highlighted in the documentation and will be changed in future versions. However, keeping track of every file access reimplements a lot of existing file system functionality and has therefore been omitted for now. Replacing the least frequently used (LFU) file results in a better performance, given it is realised efficiently, but imposes some additional tasks and hence is scheduled for near future. LFU requires the implementation of an access counter for every file. These counters are then managed in a heap structure to keep the overhead for searching and accessing a file as low as possible.

Policy Handler

While the replacement handler determines the object to move, the policy handler is among others responsible for defining the target a movement operation. Depending on the specified policy this can either be an adjacent device of the same type or a device residing inside a different tier. In case of heterogeneous tiers, it is also possible to move a file inside a tier from one device type to the other. Therefore, inter-tier and intra-tier movement is distinguished.

- `int get_target_tier(int policy)`
- `int get_device_in_tier(int tier_id, int policy)`

The *distribution policy* decides the which device of a tier is selected. At the moment, only a random distribution is supported, providing a simple

solution aiming for an even distribution but a round-robin scheme is envisaged as well. Besides managing the distribution inside a tier, the policy handler also administers the overall behaviour. An intuitive approach is to direct the I/O operations towards the fastest tier and migrate the evicted files to slower tiers. However, it may also be reasonable to separate metadata from data, storing the former on the tier with the lowest latency as metadata is usually small but accessed frequently. The data is then kept at the high-throughput devices. Another task of the policy handler is to handle accesses to files which are currently not at the fastest tier. It has to answer questions such as the following. Should a file be moved to a faster tier when it is accessed? If so, should this happen the first time it is accessed or only after a certain number of accesses in a specific time interval? To which tier should this data be moved? One can easily come up with a lot of different aspects to consider, further increasing the complexity of the movement strategies like pre-fetching schemes. Currently, the migration is based on the assumption that the faster tier is the favoured target but the possibility to prioritise different tiers for different data is in the works. The developer and the user of an application have more knowledge of a specific application and its behaviour than a simple design like the DML's one can incorporate at the beginning. Therefore, providing the possibility to allow adaptations to the policies is aimed for.

File Handler

Besides the system handler the file handler is the only component of the DM library directly providing an interface to the application layer. It offers an API for I/O operations which highly resemble the corresponding system call taking exactly the same parameters as input values while returning the expected values. These input parameters are then accordingly adapted to meet the requirements of the FFS library.

- `int dm_open_file(const char *path, int flags, mode_t mode)`
- `size_t dm_read_file(int fd, void *buf, size_t cnt, off_t offset)`
- `size_t dm_write_file(int fd, void *buf, size_t cnt, off_t offset)`

In Listing 6.3 the internal procedure of the `dm_open_file` function is detailed. The most essential distinction is whether the file does already exist in the FFS system or not. In case of the latter, the file creation is straightforward and can be simply executed on the appropriate device which is determined by the discussed policies. The assumption in the code listing is to use any device inside the fastest tier. After receiving the target device id the internal path is build

which is then passed to `ffs_open_file`.
If the file is present in the FFS system two scenarios are possible; either the file is already located on the appropriate device or it is stored elsewhere. The former is the easy case. When the file exists on a different device which is not part of the favoured tier or type, or in case of heterogeneous tiers both, a suitable target device needs to be selected to which the file is moved and opened on.

```
1  open(path,...){
2  if(path not in DB){
3      /* file is opened on fastest tier */
4      t_id = get_fastest_tier();
5      new_d_id =  get_device_in_tier(t_id,RANDOM);
6
7      /* build internal path */
8      build_i_path(ext_path,new_dev_id);
9  }else{
10     old_d_id = get_dev_id_from_path(ext_path);
11     t_id = get_fastest_tier();
12
13     /* check if file is stored inside fastest tier */
14     if(devs[old_d_id].t_id != t_id) {
15
16         /* move file to fastest tier*/
17         old_i_path = db.value;
18
19         new_d_id =  get_random_device_in_tier(t_id);
20         new_i_path = basename(mntpnt_tmp) + d_type
21                         + d_name + filename;
22         db.value = new_i_path;
23
24         con_type = get_fastest_connection_type();
25
26         ffs_move_file(sys, old_i_path, new_i_path,
27                     old_d_id, new_d_id, con_type);
28         }
29     }
30     fd = ffs_open_file(sys, new_i_path, flags, mode);
31     return fd;
32 }
```

Listing 6.3: Opening a file with a policy favouring the fastest tier

An important check, however, is not part of the listing as it leads to a significant increase in complexity; the situation where all devices fitting the policy criteria are filled with files that are currently opened.
One possibility is to move already opened files to a different tier. This approach raises several issues. When an opened file is moved, the mapping in the FFS library needs to be updated, to contain the new device id. The internal file descriptor is the index of a certain mapper element providing the access. However, the movement operation works on basis of the file path which does not supply any information about the corresponding file descriptor because this relation is managed by the kernel file system. Therefore, an additional structure handling the translation of internal path to the file descriptor is necessary to update the device id in the FFS map. Since a file can be opened several times leading to a number of different kernel file descriptors the structure needs to hold a list of all corresponding file descriptors. Besides the additional management requirements moving an opened file leads to a different behaviour for the following I/O operations than expected because they are performed on a device having different performance characteristics.
As the sudden change for sequent calls is not desirable the DML implementation does not move opened files. So, the opening procedure for the above scenario, where the favoured device type is not available, is performed on a different device type. In order to check, whether all files on the favoured are already opened the managing structure for internal paths and the corresponding file descriptors is required, nonetheless. When enough files on the target tier are closed they are moved to a slower tier so sufficient space is available to store the file which was to open. Currently, the mapping between internal paths and file descriptors does not work as it is supposed to, limiting the evaluation possibilities. Also the automatic migration is not yet fully realised.

After discussing the possibilities where to open a file the next question is when to open it there. At the moment, the DML does no support sophisticated selection schemes. When a file is accessed, it is moved to the fastest tier if possible. Even though just one single read operation might take place, this approach is easier to implement than to track the previous application I/O pattern and derive assumptions regarding the future accesses.

In Figure 6.4 the procedure of a write call is depicted highlighting the main decisions. The simplest situation is writing to a file descriptor corresponding to a file on a device with sufficient free space. If that is not the case a check is performed whether there is sufficient space left in the storage system to avoid endless attempts to migrate data between devices or tiers. When no free space

is left available in the system an error is returned to the application indicating the failing of the write operation. As files are currently stored contiguously on one device without supporting the distribution of files across several devices there might be fragmentation reducing the actually available capacity. If the tier is not full the target device is determined based on the distribution policy and the file is written to it. The complexity increases when the tier does not have enough space left free. Then one or more files must be selected which are moved to a different device either in the same or a different tier depending on whether the tier is heterogeneous or not. Finally, the file is written to the favoured device type inside the favoured tier.

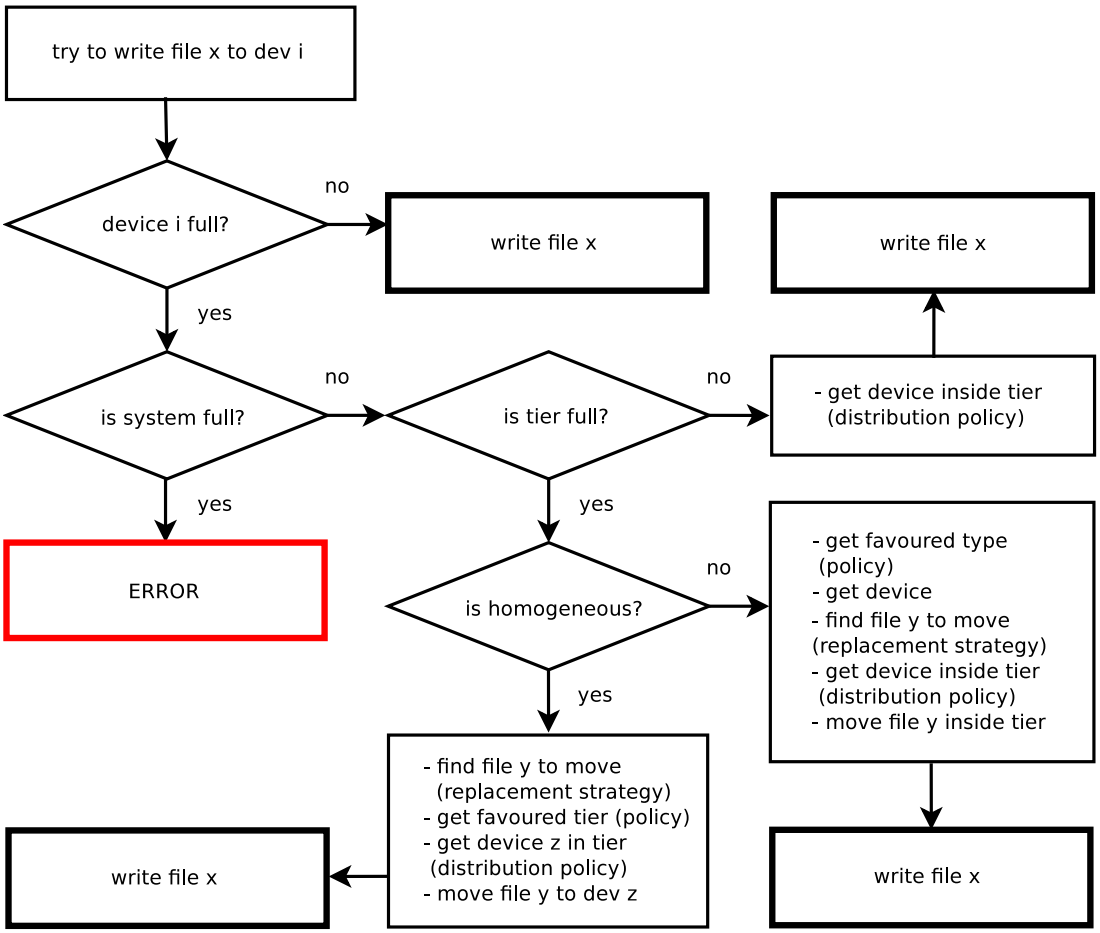

Figure 6.4.: Flowchart for a write call to the DM library

Summary

In this chapter the functionality working atop the FFS library has been presented. The requirements for a usable interface to the application are discussed. Also a reasonable preprocessing of the passed input values which are then forwarded the underlying simulation is shown. Furthermore, the problems occurring when building a model and finally an implementation for a complex system are exemplified.

7. Evaluation

This chapter covers the validation and the evaluation of the FFS and the DM library. First, the maximum performance is determined. Afterwards, a detailed analysis on the impact of the block size is presented as well as recommendations. Finally, the possibility to assess storage configurations is discussed.

7.1. Validation of FFS

To verify whether the libraries are sufficient for a real workload an existing I/O benchmark and a scientific application have been modified. In Listing 7.1 the changes made to the benchmark developed by Kuhn are shown [Kuh17b]. The sequence of I/O operations is simple; opening a file, writing to it and closing it. Afterwards, the file is opened again, read and finally closed. The block size as well as the number of blocks can be passed as a parameter. The code listing portrays how small the changes are from system calls to the usage of DML functions.

```
1  fd = open(path, O_RDWR | O_CREAT, 0600);
2  fd = dm_open_file(path,  O_RDWR | O_CREAT, 0600);
3
4  bytes = pwrite(fd, buf, opt_block_size,
     ↪ get_offset(thread_data, iteration + i));
5  bytes = dm_write_file(fd, buf, opt_block_size,
     ↪ get_offset(thread_data, iteration + i));
6
7  close(fd);
8  dm_close_file(fd);
9
10 bytes = dm_read_file(fd, buf, opt_block_size,
     ↪ get_offset(thread_data, iteration + i));
11 bytes = pread(fd, buf, opt_block_size,
     ↪ get_offset(thread_data, iteration + i));
```

Listing 7.1: I/O benchmark with adaptions to call the DML

In order to evaluate the FFS library the following benchmarks have been performed. As a first step the maximum performance generally possible was determined. This allows for a more educated decision which kind of further analysis is appropriate. When the performance is significantly lower than it should be the next step is to learn what causes this behaviour rather than evaluating several topology configurations. For measuring the maximum performance the device characteristics are set to a currently unreachable level so they do not interfere with the evaluation. The device and network latencies are defined as 1 nanosecond and the throughputs and bandwidth to 1 TB/s. Ideally, the resulting performance would reassembles the one of the hardware the benchmark and the simulation are running on. In the following, the test system is presented.

- Intel Core i5-4460 CPU @ 3.20GHz
- 16 GB DIMM DDR3 Synchronous 2133 MHz
- Samsung SSD 840: 250 GB with an average reading throughput of 530 MB/s, average writing throughput of 390 MB/s
- HDD WD 10EZRX-00D: 1 TB with an average reading/writing throughput of 150 MB/s

Figure 7.1 illustrates the performance capabilities of the used RAM evaluated with the unmodified benchmark and of the FFS simulation which was analysed using the adapted benchmark shown in Listing 7.1. The block size was varied from 2048 byte to 8.4 MB. The performance was evaluated for 14 separate block sizes for 1000 blocks. Per run 10 iterations were performed while 3 runs were measured per block size.

The results for the benchmark running in the RAM ranged from 3.6 to 7.7 GB/s for read and from 2 GB/s to 6.2 GB/s for write operations. The simulation starts with a significantly lower performance of 36 MB/s as the calculating overhead makes up a larger percentage of the execution time for small block sizes in contrast to the execution time of operations processing larger block sizes. The maximum throughput for read operations is 6.9 GB/s and 9.2 GB/s for write operations. While the RAM has a higher reading performance the simulation performs better at writing data reaching the RAM performance for a block size of 1 MB. For block sizes surpassing 1 MB the simulation of write operations is actually faster than the RAM and about 2 GB/s faster than the read simulation. This is an unexpected result as their implementation is basically the same despite the I/O system call.

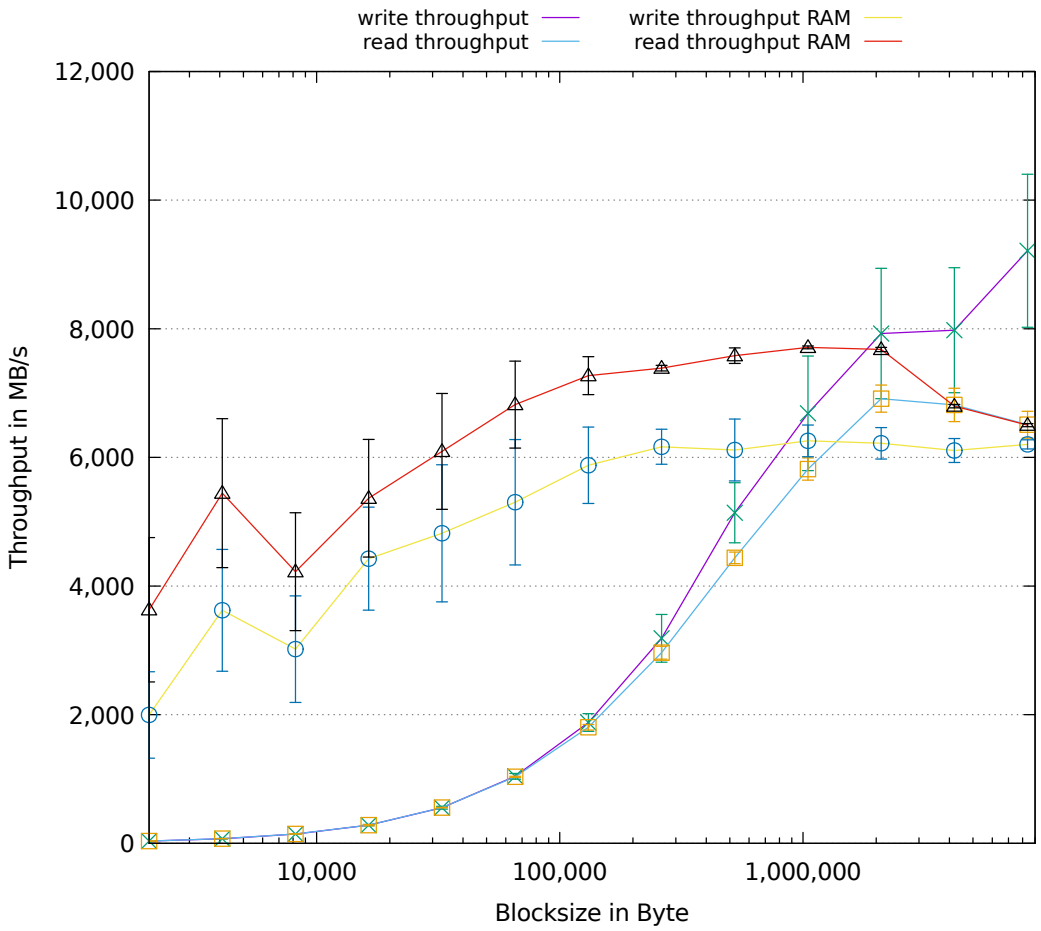

Figure 7.1.: Maximum performance of the FFS library compared to the performance of the unmodified benchmark running in the RAM

The performance for smaller block sizes is highlighted in Figure 7.2. Though the performance for a few kilobyte is low it reaches a useful range to simulate system for a block size of 100 kB and more. This is a viable result as file systems such as Lustre use a default stripe size of 1 MB.
After establishing the validity of the internal model in FFS further analysis is required to determine the resulting performance given a specific value in the configuration file.
Therefore, four different read and write throughput values have been set and evaluated with regard to several block sizes. The results are shown in Table 7.1. As also seen in Figure 7.1 and Figure 7.2, the size of data written contiguously to the simulated system is significantly impacting the reachable performance.

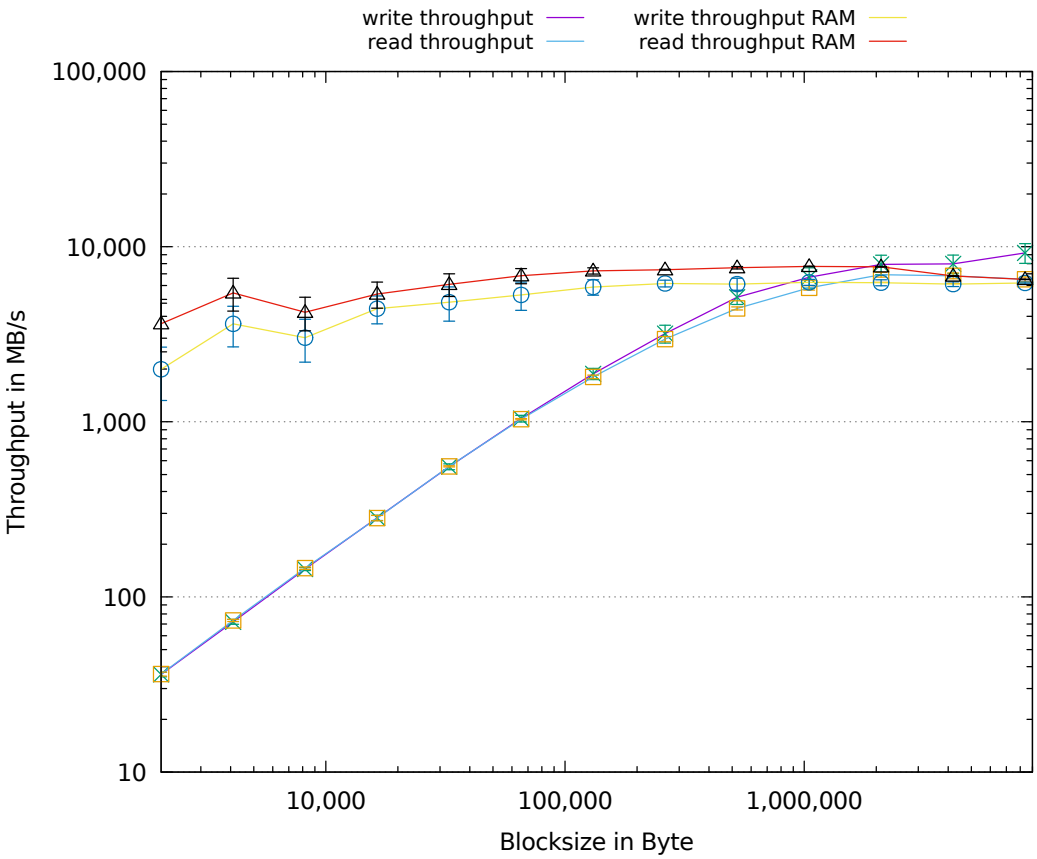

Figure 7.2.: Maximum performance of the FFS library compared to the performance of the unmodified benchmark running in the RAM with a focus on the results for smaller block sizes

In order to actually achieve a throughput as specified in the configuration file the block size has to be chosen according to the targeted value range. Reading and writing several MB per second is possible even for block sizes of 4096 byte. When the defined value is increased to 50 to 100 MB per second the block size needs to be set to several thousand byte. However, the resulting performance for higher throughput value is inside a ten percent deviation interval constituting a viable output for block sizes between 100 and 500 kB. This block size range is therefore recommended to gain the required performance.
The outliers notably surpassing the set value were not anticipated. A possible explanation might be increased caching performance when the block size is aligned to internals such as page size, cache lines or buffer sizes.

Blocksize	Write Mean	Read Mean	Defined value
4096	10.319	9.356	10
8192	11.411	10.171	
16384	9.600	9.598	
32768	9.846	9.847	
65536	9.999	10.001	
131072	10.068	10.067	
524288	10.122	10.123	
4096	32.657	29.225	50
8192	44.696	41.301	
16384	50.030	45.439	
32768	54.982	49.322	
65536	47.215	47.064	
131072	48.839	48.900	
524288	47.231	47.236	
4096	44.005	41.929	100
8192	65.253	58.533	
16384	85.948	79.445	
32768	97.196	88.251	
65536	108.923	97.478	
131072	94.481	93.966	
524288	99.171	99.123	
4096	66.610	66.094	500
8192	120.153	118.717	
16384	202.728	196.133	
32768	306.566	291.653	
65536	412.889	383.533	
131072	471.613	430.269	
524288	543.122	509.507	

Table 7.1.: Comparison of results of the I/O benchmark for the FFS simulation (left) with the throughput specified in the configuration file(right)
Write and read mean values in MB/s, 10 Iterations, 1000 blocks

7.2. Validation of DML

The definition for the overhead of the DML atop the FFS is performed by taking the first configuration setup of the FFS validation process. This way the results can be compared to one another which enables the determination of the

additional overhead introduced by the DML. In Figure 7.3 and in Figure 7.4 it is demonstrated that the performance penalty induced by the additional layer is negligible as the achieved performance is sufficient to simulate most of today's hardware appropriately. As for the FFS, the DML also performs significantly better with a block size above 100 MB which is again the recommended input range.

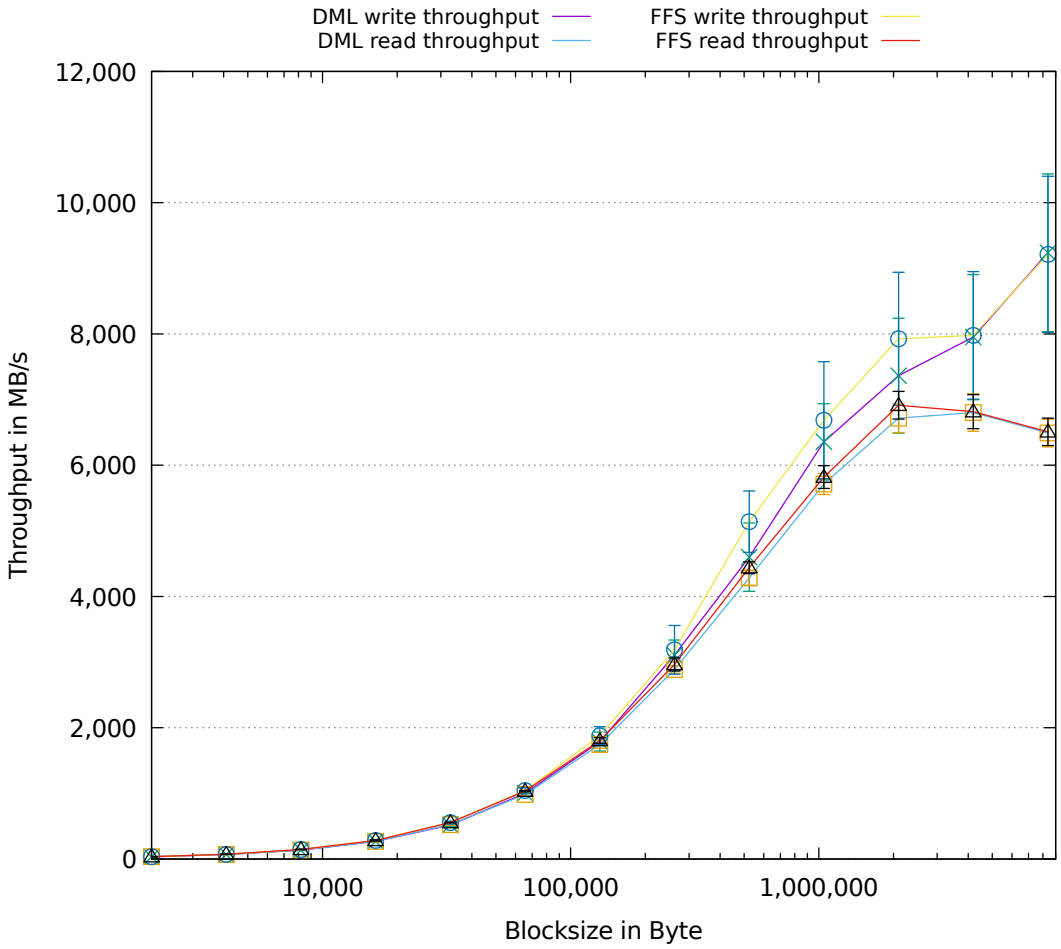

Figure 7.3.: Maximum performance of DML compared to FFS

System Configurations

Besides the I/O benchmark also the numerical application `partdiff` solving partial differential equations was modified to use the FFS library and a second version working on the DM library. This was done to evaluate the behaviour

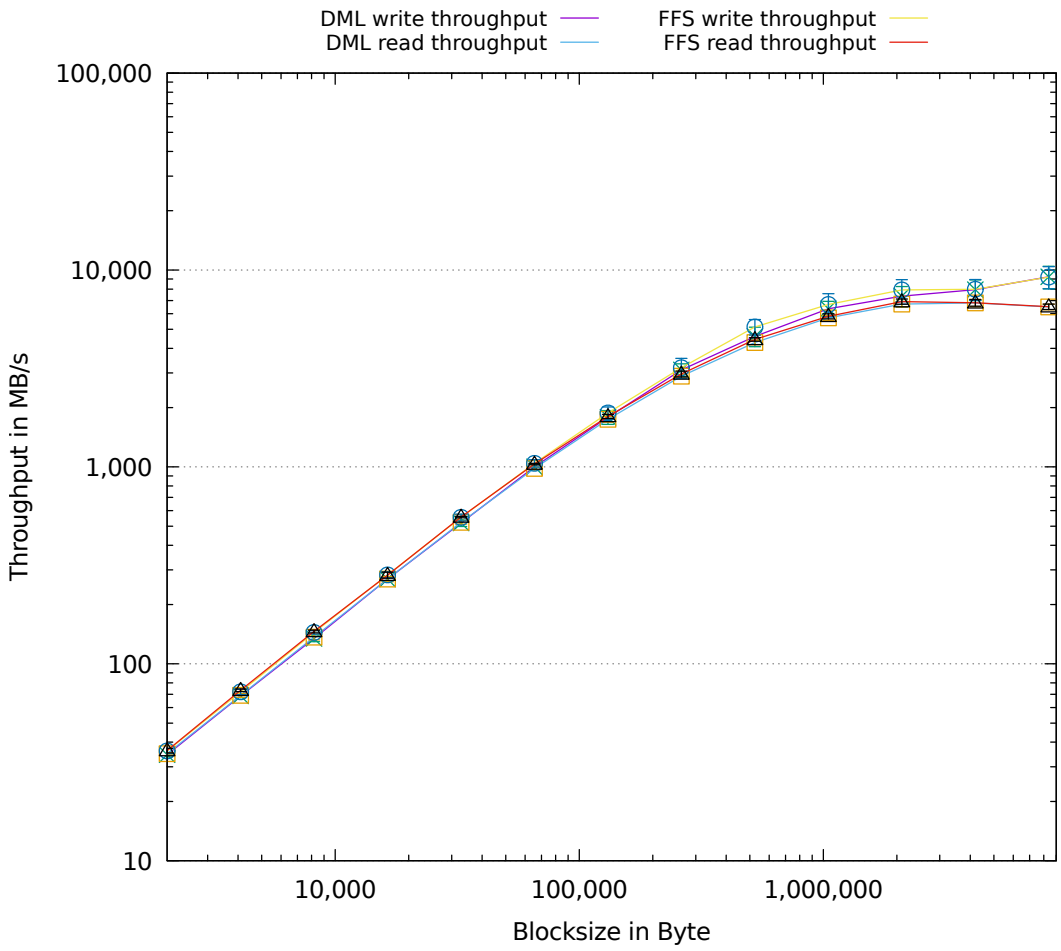

Figure 7.4.: Maximum performance of the DML compared to FFS with a focus on the results for smaller block sizes

not only for synthetic benchmarks which do not necessarily resemble real world usage. `partdiff` has been enhanced to write checkpoints after a certain number of iterations. When the application is terminated, e.g. due to a time limit on a cluster, it can be restarted reading the written checkpoint as an input file and resume calculation. The system configurations to be evaluated were configured according to the DKRZ systems as well as the proposal by the DAOS team. Depending on the used hardware the actual values were to be scaled down while keeping the ratio between the sizes of the hot and the warm storage tiers.

- DKRZ System:
  - RAM: 266 TB
  - HDD - Lustre: 54 PB

  - Infiniband FDR

- DAOS System:
  - RAM: 266 TB
  - Hot : 3 x RAM
  - Warm: 20 x RAM
  - Cold: 54 PB

These configuration have not been evaluated due to the problems in the DML implementation which were discussed in section 6.3. Also the behaviour of the random and the LRU replacement policies are not analysed for this reason. This remains an open task to complete in the future.

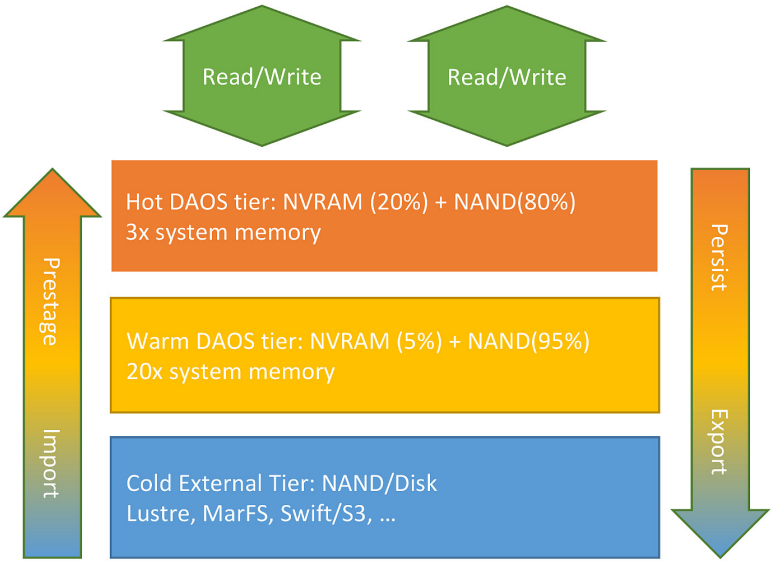

Figure 7.5.: Data migration between different heterogeneous tiers in DAOS
Taken from [Dil17]

Summary

This chapter provides the validation of the FFS and the DM library and their sufficient performance as well as further analysis on the impact of the used block size. The benchmark and its adaptations to support the libraries are discussed as well.

8. Conclusion and Future Work

This chapter recalls the initial situation and the requirements derived after reviewing state of the art solutions. After presenting the individual achievements, topics to analyse further are proposed.

With the ever-increasing gap between computational power and the speed of network and storage devices, solutions are required to lessen the I/O bottleneck as otherwise the system's performance is significantly reduced. Over the years a wide range of different storage technologies has been developed, each possessing its own set of advantages and disadvantages. In order to get the maximum performance for a limited budget, a complex storage hierarchy is a viable approach. This, however, leads to an increased complexity regarding the data management. In order to maximise the system's performance, the devices have to be used efficiently. Therefore, sensible data migration policies are vital. Since the appropriate systems usually execute the applications related to their intended purpose, e.g. climate simulations, they are not available for thorough testing and evaluation of data migrations schemes. Also, they are not accessible to a lot of researchers leaving them without an opportunity to work on this domain. In chapter 3 several existing proposals have been presented. While they expose interesting and promising functionality they also lack a number of important functions. Therefore, this thesis focussed on providing a new approach including the missing behaviour specifications. The design of an adequate model and the faced challenges have been shown in chapter 4. An idea which turned out to be quite helpful was the partition of tasks into two separate libraries. This way modifications to one of them were possible without the need to redesign the other as well. As the design and implementation went through various iterations this would have resulted in even more work. The FFS library at its current state is able to emulate arbitrary device types and devices regardless of whether they actually exist. As long as their properties are specified within the discussed value ranging up to a scale of $1,8x10^{19}$ they can be represented. Also a flexible concept of interconnection is presented to build complex hierarchies. The lacking feature of heterogeneous tiers has been realised as well.
The design of the data migration library displayed even more challenges. As

the system they are executed on can vary noticeable, finding a general model to incorporate diverse use cases is even more demanding than designing the FFS library. This is because the generic requirements of the DML have to be realised on a already complex model for the emulation of storage hierarchies. To the time of this writing, the DML supports the several replacement and distribution concepts. They are backed by the highly capable LMDB key-value store to manage the path mediation. Also, the interface is very user-friendly as it does not involve the change of any of the parameters for an I/O operation. The maximum performance of both the FFS and DML lies between 7 to 9 GB/s, making them a viable solution to simulate hierarchical storage systems and data migration schemes atop.

Future Work

Despite the widespread functionality of the libraries there are a lot of properties which would drastically extend their capabilities.

- Optimizing the memory usage increases the maximum size of the emulated system.
- Supporting large files which are distributed across several devices enhanced the applicability for HPC systems. Adding an efficient page cache lookup using a radix tree would also increase the overall performance
- Adapting the FFS model to a be more realistic by:
  - Modelling fault tolerance respectively failing of components
  - Emulating a detailed network including a model for the behaviour of network protocols
  - Enhancing the model to contain a representation for I/O nodes and burst buffers
  - Emulating delays imposed by the software stack
- Supporting parallel access makes the evaluation far more helpful to advise the management in HPC systems as a sequential model does not contain the most difficult decisions regarding the migration policies.
- Integrating a statistical approach and combining its behaviour with the emulator in order to support the emulation of large-scale HPC systems on ordinary hardware.

- Extend the namespace by considering arbitrary paths inside a simulated device not only the filename
- Implementing more complex data migration policies such as pre-fetching and automated migration to lower tiers after a certain time window passed. Additionally, the usage of different policies for data and meta data would allow to store the former on devices with high throughput while the latter is held at devices with low latency.

Besides extending the libraries another interesting point would be, to compare the performance of the FFS and DML against other implementations. Therefore, traces could be generated from the benchmark and the numerical application discussed in chapter 7. These can then be forwarded as input files to OGSSim or StorageSim. Furthermore, including the FFS and DML into the implementation of self describing data formats, e.g. HDF5, offers a valuable field of future analysis. As a file of such a format is basically an independent file system additional possibilities to evaluate a more fine-grained data migration schemes would be feasible.

Summary

This chapter provides the recap of the requirements of this thesis and the separate achievements which have been made. Additionally, several proposals for future work are presented.

Bibliography

[Ada17] Adam Armstrong. Intel 545s SSD review. `http://www.storagereview.com/intel_545s_ssd_review`, June 2017. last accessed: 17.09.2017.

[ADAD15] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. *Operating systems: Three easy pieces*. Arpaci-Dusseau Books, 2015.

[arc] IBM Archives - IBM 350 disk storage unit. `http://www-03.ibm.com/ibm/history/exhibits/storage/storage_350.html`. last accessed: 15.08.2017.

[BC05] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

[Bel66] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[BNS69] Laszlo A Belady, Robert A Nelson, and Gerald S Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *Communications of the ACM*, 12(6):349–353, 1969.

[BSSG08] John S Bucy, Jiri Schindler, Steven W Schlosser, and Gregory R Ganger. The disksim simulation environment version 4.0 reference manual (cmu-pdl-08-101). *Parallel Data Laboratory*, page 26, 2008.

[BZ02] Peter J Braam and Rumi Zahir. Lustre: A scalable, high performance file system. *Cluster File Systems, Inc*, 2002.

[Che05] Ying Chen. Information valuation for information lifecycle management. In *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pages 135–146. IEEE, 2005.

[Com] Computer History Museum. Memory & Storage: Different Tasks, Different Technologies. `http://www.computerhistory.org/revolution/memory-storage/8/249`. last accessed: 16.08.2017.

[DBW+16] B. Dong, S. Byna, K. Wu, Prabhat, H. Johansen, J. N. Johnson, and N. Keen. Data Elevator: Low-Contention Data Movement in Hierarchical Storage System. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 152–161, December 2016.

[Dil17] Dilger, Andreas. DAOS: Scale-out Object Storage for NVRAM. `http://materials.dagstuhl.de/files/17/17202/17202.AndreasDilger.Slides.pdf`, May 2017. last accessed: 26.08.2017.

[Duw14] Kira Isabel Duwe. *Comparison of kernel and user space file systems*. Universität Hamburg, 2014. Published: Online http://edoc.sub.uni-hamburg.de/informatik/volltexte/2015/210/pdf/bac_duwe.pdf.

[EDD13] Esther Spanjer, Dan Lee, and David Hiatt. SNIA - Total Cost of Solid State Storage Ownership, October 2013.

[JXW08] Hai Jin, Muzhou Xiong, and Song Wu. Information value evaluation model for ILM. In *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2008. SNPD'08. Ninth ACIS International Conference on*, pages 543–548. IEEE, 2008.

[Kuh15] Michael Kuhn. *Dynamically Adaptable I/O Semantics for High Performance Computing*. PhD Thesis, Universität Hamburg, 2015. Published: Online http://ediss.sub.uni-hamburg.de/volltexte/2015/7302/pdf/Dissertation.pdf.

[Kuh17a] Michael Kuhn. High performance i/o. `https://wr.informatik.uni-hamburg.de/_media/research/talks/2017/2017-06-08-high_performance_i_o.pdf`, June 2017. last accessed: 01.09.2017.

[Kuh17b] Michael Kuhn. Hochleistungs-ein-/ausgabe - i/o benchmark. `https://wr.informatik.uni-hamburg.de/teaching/sommersemester_2017/hochleistungs-ein_ausgabe`, June 2017. last accessed: 02.08.2017.

[Kuh17c] Michael Kuhn. Zukünftige Entwicklungen - Hochleistungs-Ein-/Ausgabe. `https://wr.informatik.uni-hamburg.de/_media/teaching/sommersemester_2017/hea-17-zukuenftige_entwicklungen.pdf`, June 2017. last accessed: 15.08.2017.

[KWI+16] K. R. Krish, B. Wadhwa, M. S. Iqbal, M. M. Rafique, and A. R. Butt. On Efficient Hierarchical Storage for Big Data Processing. In

2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pages 403–408, May 2016.

[Las07] Lash, Dan. Computer Memory Hierarchy. `https://commons.wikimedia.org/wiki/File:ComputerMemoryHierarchy.png`, September 2007. last accessed: 11.06.2017.

[Lü16] Jakob Lüttgau. *Modeling and Simulation of Tape Libraries for Hierarchical Storage Management Systems*. Master's Thesis, Universität Hamburg, 2016. Published: Online https://wr.informatik.uni-hamburg.de/_media/research:theses:jakob_l___ttgau_modeling_-and_simulation_of_tape_libraries_for_hierarchical_storage_management_systems.pdf.

[Mel17] Mellanox. LinkX™ InfiniBand Active Optical Cables . `http://www.mellanox.com/products/interconnect/infiniband-active-optical-cables.php`, 2017. last accessed: 16.09.2017.

[MGST70] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[Mä94] Christian Märtin. *Rechnerarchitektur: Struktur, Organisation, Implementierungstechnik*. Hanser, 1994.

[NTD+09] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating server storage to SSDs: analysis of tradeoffs. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 145–158. ACM, 2009.

[PCA+14] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *OSDI*, pages 433–448, 2014.

[Pet06] Michael Peterson. Ilm and tiered storage. *Storage Networking Industry Association*, 2006.

[Pet17] Peter Bright. Intel's first Optane SSD: 375GB that you can also use as RAM. `https://arstechnica.com/information-technology/2017/03/`

intels-first-optane-ssd-375gb-that-you-can-also-use-as-ram, Mar 2017. last accessed: 17.09.2017.

[Poo00] Ralph Spencer Poore. Valuing information assets for security risk management. *Information Systems Security, Auerbach Publications*, 9(4), 2000.

[PS04] Peterson, Michael and St. Pierre, Edgar. Information Lifecycle Management Roadmap, October 2004.

[Ray17] Raymond, Eric S. The Lost Art of C Structure Packing. http://www.catb.org/esr/structure-packing/, June 2017. last accessed: 25.08.2017.

[RG10] Aditya Rajgarhia and Ashish Gehani. Performance and extension of user space file systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 206–213. ACM, 2010.

[Ryz17] Ryzen Review. AMD Ryzen Review and Benchmarks – 2-channel DDR4 Cache and Memory Performance. http://www.sisoftware.eu/2017/04/05/amd-ryzen-review-and-benchmarks-cache-and-memory/, April 2017. last accessed: 17.09.2017.

[Sam16] Samsung. Samsung 960 Pro M2 NVMe SSD Review. http://www.storagereview.com/samsung_960_pro_m2_nvme_ssd_review/, October 2016. last accessed: 17.09.2017.

[Sea16] Seagate. Enterprise Capacity 2.5 HDD - Datenblatt. http://www.seagate.com/www-content/product-content/enterprise-hdd-fam/enterprise-capacity-2-5-hdd/en-us/docs/ent-capacity-2-5-hdd-ds1719-8-1602de.pdf, 2016. last accessed: 15.08.2017.

[SH02] Frank B Schmuck and Roger L Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *FAST*, volume 2, 2002.

[SMM10] Ketan Shah, Anirban Mitra, and Dhruv Matani. An O (1) algorithm for implementing the LFU cache eviction scheme. *dhruvbird. com/lfu. pdf*, pages 1–8, 2010.

[SPBW10] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending SSD Lifetimes with Disk-Based Write Caches. In *FAST*, volume 10, pages 101–114, 2010.

[Ste09] Stephen Lawson. Two rival supercomputers duke it out for top spot. http://news.idg.no/cw/art.cfm?id=FB70C2C5-1A64-6A71-CEEA6C17D51B1E3C, Nov 2009. last accessed:.

[SVSA06] Gauri Shah, Kaladhar Voruganti, Piyush Shivam, and Maria Alvarez. Ace: Classification for information lifecycle management. *NASA Mass Storage Systems and Technologies*, 2006.

[TFLS08] Lars Arne Turczyk, Christian Frei, Nicolas Liebau, and Ralf Steinmetz. Eine Methode zur Wertzuweisung von Dateien in ILM. In *Multikonferenz Wirtschaftsinformatik*, 2008.

[THBS06] Lars Arne Turczyk, Oliver Heckmann, Rainer Berbner, and Ralf Steinmetz. A formal approach to Information Lifecycle Management. In *Proceedings of 17th Annual IRMA International Conference, Washington DC*, 2006.

[Ton10] Tony Pearson. The Correct Use of the term Nearline. https://www.ibm.com/developerworks/community/blogs/InsideSystemStorage/entry/the_correct_use_of_the_term_nearline2?lang=en, October 2010. last accessed: 16.08.2017.

[TOP17] TOP500. TOP500 List. https://www.top500.org/statistics/list/, July 2017. last accessed: 15.08.2017.

[WA10] Fons Wijnhoven and Chintan Amrit. Evaluating the Applicability of a Use Value-Based File Retention Method. In *Proceedings of SIGSVC Workshop*, pages 10–118, 2010.

[Wik17a] Wikipedia. 3D XPoint. https://de.wikipedia.org/wiki/3D_XPoint, 2017. last accessed: 17.09.2017.

[Wik17b] Wikipedia. Ethernet. https://de.wikipedia.org/wiki/Ethernet, Aug 2017. last accessed: 15.09.2017.

[Wik17c] Wikipedia. Flash Speicher. https://de.wikipedia.org/wiki/Flash-Speicher#Anzahl_der_L.C3.B6schzyklen, September 2017. last accessed: 17.09.2017.

[Wik17d] Wikipedia. InfiniBand. https://de.wikipedia.org/wiki/InfiniBand, Jul 2017. last accessed: 17.09.2017.

[Wik17e] Wikipedia. Random Access Memory. https://de.wikipedia.org/wiki/Random-Access_Memory, 2017. last accessed: 16.09.2017.

[WLC+14] Lipeng Wan, Zheng Lu, Qing Cao, Feiyi Wang, Sarp Oral, and Bradley Settlemyer. SSD-optimized workload placement with adaptive learning and classification in HPC environments. In *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on*, pages 1–6. IEEE, 2014.

[YWH+13] Jinsoo Yoo, Youjip Won, Joongwoo Hwang, Sooyong Kang, Jongmoo Choil, Sungroh Yoon, and Jaehyuk Cha. Vssim: Virtual machine based ssd simulator. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pages 1–14. IEEE, 2013.

[ZCD+10] Gong Zhang, Lawrence Chiu, Clem Dickey, Ling Liu, Paul Muench, and Sangeetha Seshadri. Automated lookahead data migration in SSD-enabled multi-tiered storage systems. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–6. IEEE, 2010.

[ZOS+04] Erez Zadok, Jeffrey Osborn, Ariye Shater, Charles P Wright, Kiran-Kumar Muniswamy-Reddy, and Jason Nieh. Reducing Storage Management Costs via Informed User-Based Policies. In *MSST*, pages 193–197, 2004.

Appendices

A. Measurements

Maximum Performance FFS

Blocksize	Write Mean	Write Stddev	Read Mean	Read Stddev
2048	36.088	1.123	36.270	0.965
4096	72.073	2.260	73.390	1.132
8192	144.324	2.812	145.967	2.393
16384	282.813	9.828	282.115	9.970
32768	553.883	21.726	556.562	4.455
65536	1042.738	44.765	1034.728	8.665
131072	1879.144	137.301	1804.310	43.362
262144	3187.942	370.653	2964.240	99.895
524288	5139.507	466.636	4438.622	89.431
1048576	6686.209	890.588	5819.562	173.233
2097152	7926.003	1013.281	6913.679	210.910
4194304	7977.956	972.291	6815.998	259.577
8380000	9212.609	1189.637	6508.697	208.138

Table A.1.: Maximum Performance of FFS system : write and read mean values and standard deviation in MB/s. 10 Iterations per run, 3 runs per blocksize, 1000 blocks

Blocksize	Write Mean	Write Stddev	Read Mean	Read Stddev
2048	1996.075	671.981	3631.743	1122.115
4096	3623.084	947.534	5444.835	1157.366
8192	3019.589	828.520	4223.573	916.543
16384	4426.516	800.021	5365.683	913.904
32768	4820.951	1065.860	6093.620	900.049
65536	5303.987	974.462	6821.442	675.665
131072	5877.755	593.770	7272.236	295.132
262144	6166.008	272.414	7388.007	43.758
524288	6115.949	480.432	7582.957	120.955
1048576	6257.520	246.122	7711.123	22.404
2097152	6219.798	245.156	7678.670	32.786
4194304	6107.757	186.597	6797.876	26.529
8380000	6202.635	71.733	6501.863	23.740

Table A.2.: Maximum Performance of RAM with unmodified benchmark
Write and read mean values and standard deviation in MB/s.
10 Iterations, 1000 blocks

Maximum Performance DML

Blocksize	Write Mean	Write Stddev	Read Mean	Read Stddev
2048	34.521	0.905	34.947	0.450
4096	68.690	1.381	69.120	1.551
8192	134.621	5.024	136.548	4.309
16384	269.369	7.270	269.341	8.399
32768	519.283	21.030	523.509	11.636
65536	1000.626	34.701	982.928	17.063
131072	1788.819	143.949	1745.243	57.625
262144	3107.953	229.253	2887.924	57.095
524288	4599.878	520.169	4282.770	110.901
1048576	6356.614	579.948	5714.211	160.864
2097152	7364.123	874.712	6717.032	218.934
4194304	7952.183	953.073	6802.950	285.738
8380000	9237.149	1201.675	6491.406	210.363

Table A.3.: Maximum Performance of DML with static device without network overhead
Write and read mean values and standard deviation in MB/s.
10 Iterations per run, 3 runs, 1000 blocks

B. Library APIs

B.1. FFS

API

SYSTEM HANDLER:

- *system_struct** **ffs_initialise_system**(const char *mntpnt_tmpfs)
- *void* **ffs_finalise_system**(system_struct *system)

CONFIG HANDLER:

- *int* **ffs_save_system_to_config_file**(system_struct *system, const char *path_for_new_file)
- *int* **ffs_build_system_from_config_file**(system_struct *system, const char *path_to_config_file)

HSM HANDLER:

- *tier_id* **ffs_create_tier**(system_struct *system, const char *name, char *dev_type, char *con_type)
- *int* **ffs_delete_tier**(system_struct *system, tier_id t_id)
- *int* **ffs_assign_dev_to_new_tier**(system_struct *system, tier_id old_t_id, tier_id new_t_id, dev_id d_id)
- *int* **ffs_delete_dev_from_tier**(system_struct *system, tier_id t_id, dev_id d_id)
- *int* **ffs_add_connection_type**(system_struct *system, const char *new_type, uint64_t latency, uint64_t bandwidth)
- *int* **ffs_delete_connection_type**(system_struct *system, const char *type)

DEVICE HANDLER:

- *int* **ffs_add_device_type**(system_struct *system,
  const char *new_type, uint64_t r_ltncy, uint64_t w_ltncy,
  uint64_t r_thrghpt, uint64_t w_thrghpt, uint64_t capacity)
- *int* **ffs_delete_device_type**(system_struct *system,
  const char *type)
- *dev_id* **ffs_create_device**(system_struct *system,
  const char *name, char *type, tier_id t_id)
- *int* **ffs_delete_device**(system_struct *system, dev_id d_id)

FILE HANDLER:

- *int* **ffs_open_file**(system_struct *system, const char *path,
  int flags,mode_t mode)
- *int* **ffs_close_file**(system_struct *system, int fd)
- *int* **ffs_create_file**(system_struct *system, const char *path,
  mode_t mode)
- *int* **ffs_copy_file**(system_struct *system, const char *path,
  const char *path_for_copy)
- *int* **ffs_delete_file**(system_struct *system, const char *path)
- *size_t* **ffs_read_file**(system_struct *system, int fd, void *buf,
  size_t cnt, off_t offset)
- *size_t* **ffs_write_file**(system_struct *system, int fd, void *buf,
  size_t cnt, off_t offset)
- *int* **ffs_import_file**(system_struct *system,
  const char *external_path, const char *internal_path)
- *int* **ffs_export_file**(system_struct *system,
  const char *internal_path, const char *external_path)
- *int* **ffs_move_file**(system_struct *system, const char *src_path,
  const char *dest_path, int src_dev_id,
  int dest_dev_id, const char *con_type)

Configuration File

```
# === SYSTEM SPECIFICATIONS ===

[system]
Current local time and date=
# --- System defines ---
MAX_NR_DEV_TYPES=       // int
MAX_NR_CON_TYPES=       // int
MAX_NR_TIERS=           // int
MAX_NR_DEVS=            // int
MAX_NR_FD=              // int
MAX_PATH_LENGTH=        // int
MAX_NAME_LENGTH=        // int
MAX_NR_DEVS_PER_TIER=   // int
#
# --- System characteristics ---
mntpnt=                 // string
persistent=             // int
savepnt=                // string
dev_type_cnt=           // int
con_type_cnt=           // int
tier_cnt=               // int
dev_cnt=                // int
max_tier_id=            // int
max_dev_id=             // int
sum_capacity=           // uint64 [B]
# list of device ids
dev_ids=                // int;int;int
# list of tier ids
tier_ids=               // int;int;int

# --- Following section lists all existing device
  ↪ types ---
[dt_i]
dt_name=                // string
r_ltncy=                // uint64 [nsec]
w_ltncy=                // uint64 [nsec]
r_thrghpt=              // uint64 [B/s]
w_thrghpt=              // uint64 [B/s]
```

```
capacity=                // uint64 [B]

# --- Following section lists all existing devices ---
[d_id_j]
d_name=                  // string
dev_type=                // string
tier_id=                 // int

# --- Following section lists all existing tiers ---
[t_id_k]
t_name=                  // string
# list of device ids belonging to this tier
dev_ids=                 // int;int;int
dev_in_tier_cnt=         // int
is_homogenous=           // int [0/1]
d_type=                  // string
c_type=                  // string

# --- Following section lists all existing connection types ---
[ct_l]
ct_name=                 // string
ltncy=                   // uint64 [nsec]
bandwidth=               // uint64 [Bit/s]
```

Listing B.1: Config file structure

The following is a short example file with actual values.

```
# === SYSTEM SPECIFICATIONS ===

[system]
Current local time and date=Mon Jul 17 15:57:21 2017\n
# --- System defines ---
MAX_NR_DEV_TYPES=50
MAX_NR_CON_TYPES=50
MAX_NR_TIERS=20
MAX_NR_DEVS=40
MAX_NR_FD=200
MAX_PATH_LENGTH=256
MAX_NAME_LENGTH=256
MAX_NR_DEVS_PER_TIER=256
```

```
14 # --- System characteristics ---
15 mountpoint=/home/testmountpoint
16 dev_type_cnt=2
17 con_type_cnt=1
18 tier_cnt=2
19 dev_cnt=2
20 max_tier_id=1
21 max_dev_id=2
22 sum_capacity=5920000000000
23 # list of device ids
24 dev_ids=0;1;
25 # list of tier ids
26 tier_ids=0;1;
27
28 # --- Following section lists all existing device
     ↪ types ---
29 [dt_0]
30 dt_name=Seagate_SSD
31 r_ltncy=135000
32 w_ltncy=59000
33 r_thrghpt=560000000
34 w_thrghpt=430000000
35 capacity=1920000000000
36
37 [dt_1]
38 dt_name=Barracuda
39 r_ltncy=8500000
40 w_ltncy=9500000
41 r_thrghpt=156000000
42 w_thrghpt=156000000
43 capacity=4000000000000
44
45 # --- Following section lists all existing devices ---
46 [d_id_0]
47 d_name=device1
48 d_type=Seagate_SSD
49 tier_id=0
50
51 [d_id_1]
52 d_name=device2
```

```
53 d_type=Barracuda
54 tier_id=0
55
56 # --- Following section lists all existing tiers ---
57 [t_id_0]
58 t_name=test_tier
59 # list of device ids belonging to this tier
60 dev_ids=0;
61 dev_in_tier_cnt=1
62 is_homogenous=1
63 d_type=Seagate_SSD
64 c_type=Infiniband_EDR
65
66 [t_id_1]
67 t_name=test_tier2
68 # list of device ids belonging to this tier
69 dev_ids=1
70 dev_in_tier_cnt=1
71 is_homogenous=1
72 d_type=Barracuda
73 c_type=Infininband_EDR
74
75 # --- Following section lists all existing connection
     ↪ types ---
76 [ct_0]
77 ct_name=Infiniband_EDR
78 ltncy=500
79 bandwidth=300000000000
```

Listing B.2: Config file structure

B.2. DML

API

SYSTEM HANDLER:

- *int* **dm_initialize_system**(const char *mntpnt_tmpfs, const char *config_file, const char *db_dir
- *void* **dm_finalize_system**(const char *mntpnt_tmpfs)

DB HANDLER:

- *int* **initialize_db**(dm_system *system, const char *db_directory)
- *int* **finalize_db**(dm_system *system)
- *char * * **get_internal_path_from_db**(const char *extern_path, const char *intern_path)
- *char * * **put_external_path_to_db**(const char *extern_path, const char *intern_path)

FILE HANDLER:

- *int* **dm_open_file**(const char *path, int flags, mode_t mode)
- *int* **dm_close_file**(int fd);
- *int* **dm_create_file**(const char *path, mode_t mode)
- *int* **dm_copy_file**(const char *path,const char *path_for_copy)
- *int* **dm_delete_file**(const char *path)
- *size_t* **dm_read_file**(int fd, void *buf, size_t cnt, off_t offset)
- *size_t* **dm_write_file**(int fd, void *buf, size_t cnt, off_t offset)

REPLACEMENT HANDLER:

- *char* ***get_random_file**(char *device_path)
- *char* ***get_LRU_file**(char *device_path)

POLICY HANDLER:

- *int* **get_device_in_tier**(int tier_id, int policy)
- *int* **get_target_tier**(int policy)

Acronyms

AMAT Average Memory Access Time. 13

API Application Programming Interface. 7, 36, 46

Btrfs B-tree file system. 8

CPU Central Processing Unit. 9, 10

CSMA/CD Carrier Sense Multiple Access Collision Detection. 20

DRAM Dynamic RAM. 17

FDDI Fiber Distributed Data Interface. 20

FIFO First In, First Out. 14–16

FS File System. 7

FUSE File System in Userspace. 7

GB Giga Byte (1 000 000 000 Bytes). 10

GPFS General Parallel File System. 8

HDD Hard Disk Drive. 10, 12, 18, 37, 43

HPC High Performance Computing. 8

HSM Hierarchical Storage Management. 5, 22, 36, 37, 43, 44

I/O Input and Output. 1, 8, 11

IEEE Institute of Electrical and Electronics Engineers. 20

ILM Information Lifecycle Management. 22

JBOD Just a Bunch Of Disks. 29

kB Kilo Byte (1 000 Bytes). 10

LFU Least Frequently Used. 4, 14, 15

LMDB Lightning Memory-mapped Database. 64

LRU Least Recently Used. 4, 14–16, 30

MAID Massive Array of Idle Drives. 12

MB Mega Byte (1 000 000 Bytes). 9, 10

NFS Nework File System. 8

NRU Not Recently Used. 14

NVRAM Non Volatile Random Access Memory. 1, 10

OIVE Output of Information Value Evaluation. 23, 24

OS Operating System. 13

PB Peta Byte (1 000 000 000 000 000 Bytes). 9

PFS disk-based Parallel File System. 3

POSIX Portable Operating System Interface. 7

RAID Redundant Array of Independent Disks. 29

RAM Random Access Memory. 2, 10, 16

RPM Rotations Per Minute. 18

SNIA Storage Networking Industry Association. 23

SNIA-DMF Storage Networking Industry Association Data Management Forum. 22

SRAM Static RAM. 16

SSD Solid State Drive. 10

TB Tera Byte (1 000 000 000 000 Bytes). 10, 11

VFS Virtual File System. 7, 8

WORO Write Once, Read Occasionally. 12

ZeroMQ Zero Message Queue. 29

ZFS Zettabyte File System. 8

List of Figures

List of Listings

List of Tables

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Master Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ort, Datum	Unterschrift

Veröffentlichung

Ich bin damit einverstanden, dass meine Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik eingestellt wird.

Ort, Datum	Unterschrift