

# Design, Implementation, and Evaluation of a Low-Level Extent-Based Object Store

— Masterarbeit —

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

Vorgelegt von:	Sandra Schröder
E-Mail-Adresse:	8schroed@informatik.uni-hamburg.de
Matrikelnummer:	6060939
Studiengang:	Informatik
Erstgutachter:	Prof. Dr. Thomas Ludwig
Zweitgutachter:	Prof. Dr.-Ing. Norbert Ritter
Betreuer:	Michael Kuhn

Hamburg, Dezember 2013



# Abstract

An object store is a low-level abstraction of storage. Instead of providing a block-level view of a storage device, an object store allows to access it by a more abstract way, namely via objects. Being on top of a storage device, it is responsible for storage management. It can be used as a stand-alone light-weight file system when only basic storage management is necessary. Moreover it can act as a supporting layer for full-featured file systems in order to lighten their management overhead. Only a few object store solutions exist. These object stores are, however, not suitable for these use cases. For example no user interface is provided or it is too difficult to use. The development of some object stores has ceased, so that the code of the implementation is not available anymore. That is why a new object store is needed facing those problems.

In this thesis a low-level and extent-based object store is designed and implemented. It is able to perform fully-functional storage management. For this, appropriate data structures are designed, for example, so-called inodes and extents. These are file system concepts that are adapted to the object store design and implementation. The object store uses the memory mapping technique for memory management. This technique maps a device into the virtual address space of a process, promising efficient access to the data. An application programming interface is designed allowing easy use and integration of the object store. This interface provides two features, namely synchronization and transactions. Transactions allow to batch several input/output requests into one operation. Synchronization ensures that data is written immediately after the write request. The object store implementation is object-oriented. Each data structure constitutes a programming unit consisting of a set of data types and methods.

The performance of the object store is evaluated and compared with well-known file systems. It shows excellent performance results, although it constitutes a prototype. The transaction feature is found to be efficient. It increases the write performance by a factor of 50 when synchronization of data is activated. It especially outperforms the other file systems concerning metadata performance. A high metadata performance is a crucial criterion when the object store is used as a supporting storage layer in the context of parallel file systems.



# Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
1.1	Motivation . . . . .	7
1.2	Scope and Goals of the Thesis . . . . .	8
1.3	Outline . . . . .	9
<b>2</b>	<b>File Systems and Object Stores</b>	<b>11</b>
2.1	Object Stores: State of the Art . . . . .	11
2.2	File System Concepts . . . . .	13
2.3	Memory Mapping . . . . .	18
<b>3</b>	<b>Object Store Design</b>	<b>21</b>
3.1	General Design Decisions . . . . .	21
3.2	Data Structures and On-Disk Layout . . . . .	24
3.3	The LEXOS Application Programming Interface . . . . .	27
3.4	Features . . . . .	29
3.5	Related Work . . . . .	31
<b>4</b>	<b>Technical Realization of the Object Store</b>	<b>35</b>
4.1	Software Design and Environment . . . . .	36
4.2	Bitmaps . . . . .	36
4.3	Extent Allocation . . . . .	38
4.4	Object Store Operations . . . . .	40
4.5	Object and Inode Operations . . . . .	44
4.6	Transactions . . . . .	49
<b>5</b>	<b>Evaluation</b>	<b>53</b>
5.1	Small-Access Benchmark . . . . .	53
5.2	Metadata Performance . . . . .	67
<b>6</b>	<b>Conclusion and Future Work</b>	<b>71</b>
6.1	Summary and Conclusion . . . . .	71
6.2	Future Work . . . . .	72
	<b>Bibliography</b>	<b>75</b>
	<b>List of Figures</b>	<b>79</b>

<b>List of Listings</b>	<b>81</b>
<b>Appendices</b>	<b>83</b>
<b>A Setup and Configuration of LEXOS</b>	<b>85</b>
A.1 mklexos - Command Line Tool . . . . .	85
<b>B LEXOS API</b>	<b>87</b>
B.1 Object Store Operations . . . . .	87
B.2 Object Operations . . . . .	88
B.3 Transactions . . . . .	90
<b>C API Usage Example</b>	<b>91</b>
<b>D Metadata Performance</b>	<b>95</b>

# Chapter 1.

## Introduction

### 1.1. Motivation

Today's data management is faced with an increasing amount of digital data produced in various fields of application. This constant growth of data, recently known as *Big Data*, raises several challenges in terms of scalability and reliability of modern data management. Much of the data growth is driven by the recent innovations in *Cloud Computing* and *High Performance Computing*. Both fields of application are based on parallel systems, that is, groups of computational nodes and storage nodes connected by a network. The storage nodes store application data persistently and serve data to the rest of the parallel system. In order to handle this data, efficient parallel file systems build the basis. Those file systems have to deal with a lot of tasks and challenges. For example, they have to handle extremely large data sets and a lot of requests from numerous clients querying data concurrently. These requests have to be performed as efficiently as possible in order to provide immediate access to the data. In order to provide this, the data has to be managed on the storage devices. That means in addition to serving the data and metadata to the clients, a file system has to manage the disk locations, so that data can be retrieved when it is required. These are difficult tasks, often representing a bottleneck for the performance of parallel file systems [AEHH<sup>+</sup>11], since such file systems often have to deal with amounts of data in a petabyte scale [ADD<sup>+</sup>08]. Moreover, these file systems maintain millions of files [XXSM09]; for example the file systems of the German Climate Computing Center [Ger] store approximately 200 million files in total. That is why a supporting layer that applies storage management in an efficient way and that lightens the workload of parallel file systems would be advantageous. This is where the *object store* concept comes into play.

A storage paradigm, the so-called object store [GNA<sup>+</sup>97], addresses the challenge of efficiently managing massive volumes of data in parallel applications. Object stores abstract data in *objects* often identified by a unique number and hiding the physical locations of the data they are related with. Being on top of the storage device, object stores allow the file system to interact with the device via those objects. The main task of an object store is storage management, so that it can be used as a replacement for the storage management normally being performed by the file system itself. In this way parallel file systems can concentrate on their high-level tasks and invoke the object store

to manage the data on the storage device.

Although a few object store solutions exist, most of them are not applicable due to several reasons:

**Specific-purpose object store** The object store is mainly designed and implemented for a specific file system. It is adapted to the architecture of the file system and is an integral part of it. The object store can not be used for another file system, since there is no user interface available.

**Enormous integration effort** Although there is the possibility to integrate the object store into another file system, it can only be done with the utmost effort.

**Un-documented and hard-to-use user interfaces** Although there is a user interface available, it is not clear how to use it due to a lack of documentation.

**Un-availability of object store code** The implementation of the object store in form of code can not be used as it is not provided by the developers, because the development and maintenance of the object store implementation has been stopped.

In addition a lot of parallel file systems use another file system instead of an object store for storage management for single storage devices. These file systems often have an enormous workload that could decrease the performance of the parallel file system. This is especially true for metadata performance. Consequently, a new object store solution is necessary meeting all these challenges described above. In the next section, the goals and the associated tasks faced in this thesis are motivated.

## 1.2. Scope and Goals of the Thesis

The previous section made clear why a new object store is required. The goal of this thesis is the design and implementation of a thread-safe library allowing the creation of the object store on an arbitrary storage device. The object store should be able to do basic storage management on a device. It should either act as a stand-alone and light-weight file system or as a low-level abstraction of a storage device providing an object-based view to a file system. In the second use case, the object storage applies the storage management for the file system. Since the object store may be used in a parallel file system, it should provide a high metadata performance. In this thesis a fully-functional object store prototype is developed that builds the basis for extensions and optimizations in future use.

The development environment is focused on Linux operating systems, since they are the most present operating systems in high performance computing. The realization of the object store library includes the design of appropriate data structures needed for storage management and forming the on-disk layout of the object store.

In combination with the object store, a new application programming interface is designed. The interface should provide basic operations for the object store and objects and works as a connecting layer between an application program and the object store.



Moreover, it should be possible to pass additional information to the object store via the interface. With this possibility, specific optimizations can be applied internally by the object store, so that data can be handled more efficiently.

The further focus of this thesis is the performance evaluation of the object store concerning the read-write speed and the metadata performance. This will give an impression about the quality of the store concerning Input/Output (I/O) and metadata performance. The results will be compared with the performance of other file systems.

## 1.3. Outline

The remainder of this thesis is organized as follows:

**Chapter 2: File Systems and Object Stores** The object store concept was presented shortly in this introduction. In the following chapter, it will be described in more detail and state of the art object store solutions are presented. The concepts of file systems and object stores are related to some extent. The main concepts of file systems are presented in this chapter that can be used later for the design and realization of the object store.

**Chapter 3: Object Store Design** Based on the state of the art on file systems and object stores presented in the previous chapter the design of the object store is described. This includes the data structures needed for basic storage management as well as an appropriate user interface in order to interact with the object store and the objects. The design is finally compared with related work on object stores in order to clearly state why there is a need for a new object store solution.

**Chapter 4: Realization** This chapter focuses on the realization of the design described in the previous chapter and emphasizes algorithm and implementation fundamentals and highlights.

**Chapter 5: Evaluation** In order to evaluate the quality of the designed object store, many benchmarks are applied. The benchmarks measure read and write speed and metadata performance. The results are compared with the performance of other file systems.

**Chapter 6: Conclusion and Future Work** The final chapter gives a summary on the results of the thesis. Furthermore, possible directions for future work are discussed.



## Chapter 2.

# File Systems and Object Stores

Hard disk devices (HDD) are the most represented non-volatile storage devices in modern computer systems, regardless of whether in personal or in high performance computers. They are able to store a huge amount of data over a long, indefinite period of time in contrast to main memory, being of volatile nature and not able to save information after power is turned off. On a low system level, HDDs can be viewed as a collection of storage blocks which can be accessed sequentially or randomly. However, this block abstraction level is still too detailed for many applications, so that it becomes necessary to organize blocks in a more convenient way in order to increase access performance and reduce maintenance effort. This is the intended purpose of *file systems* [Tan08] and *object stores*.

In this chapter the motivation for object stores will be emphasized including an introduction to state-of-the-art object stores. Moreover a detailed understanding of the object store concept will be conveyed in order to form a basis for the following design of the object store. The main task of a file system is managing data on a storage device. For this, appropriate data structures are required in order to keep track of the data. Common data structures for storage management are presented. Two of them, the so-called inodes and extents, can be adapted for use in the object store and are described in detail. The memory management of the object store designed in this thesis, is based on memory mapping. In the last section of the chapter, the basics of this technique are explained.

## 2.1. Object Stores: State of the Art

As described above, disk access on block-level is not well-suited for most applications. Object stores present a more practicable way to access the storage device. The object store acts as an additional layer between the user and the storage device, hiding the details about how data is stored on it. Instead of working directly on the device, that is, retrieving single blocks from it, the storage is accessed via pieces of storage, namely the *objects*.

The concept of object stores was first applied as a research project in 1996 at the Carnegie Mellon University in the context of network-attached storage [GNA<sup>+</sup>97]. The aim of using object stores was to develop an appropriate interface for reducing the

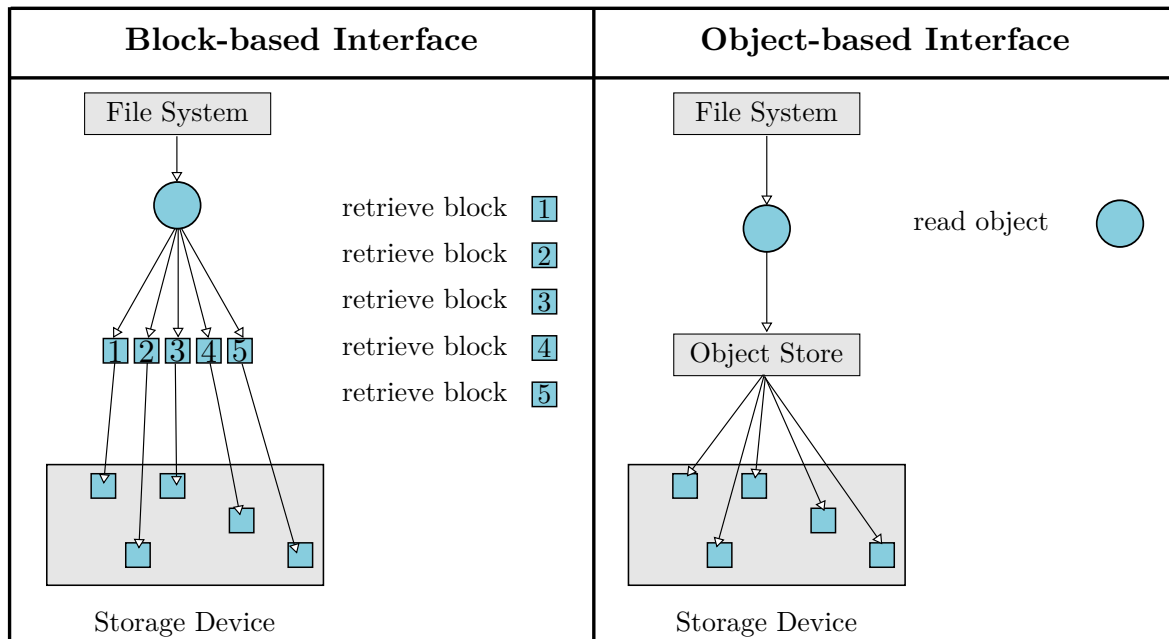
client-storage interactions that are relayed through the file manager, for example like name space manipulation. These tasks were separated from operations that go directly to the disk, like file read and write operations. These operations were executed on abstract objects. This approach was expected to promise a better scalability and higher performance in network-attached secure disks.

From then, the object store concept was developed over the years and can nowadays be found in high-performance distributed file systems. For example, the parallel file system Lustre [Sch03] contains so-called *object storage targets (OST)*. This is an additional software layer on top of the storage device providing an interface used by the clients in order to access file data with one or more objects. This software layer is, however, often a file system and not an object store. Another file system maintaining an object store is ZFS, offloading storage management to this unit. The object store of ZFS [BAH<sup>+</sup>03] is called the Data Management Unit and will be compared with the design of the object store developed in this thesis (see Section 3.5).

Object stores are also prominent in cloud computing where most cloud storage solutions apply object-based architectures providing access to objects by internet protocols like REST (Representational State Transfer), SOAP (Simple Object Access Protocol) and BitTorrent. Well-known examples in this field are Amazon S3 [Ama13] and Google Cloud Storage [Goo]. Both are on-line file-storage web-services providing storage for any arbitrary web service that needs to store any amount of data. They organize objects in so-called buckets and are assigned to a specific region (for example, Europe, USA etc.). They allow objects only to be present in a specific region, so that they cannot be stored in another region. Each bucket can store an arbitrary, unlimited number of objects.

As described, object stores are mainly present in high-performance parallel file systems and cloud computing. Both areas apply the concept of object stores differently. In this thesis, object stores are seen in the context of parallel file systems:

An object store is a collection of objects. Each object represents a data construct consisting of two parts: data and metadata. An object store defines a global name space for these objects. Objects can be related to some extent or be completely independent of each other. The object store concept as it is understood in this work is illustrated in Figure 2.1 by comparison with the traditional block access used by file systems. An object store provides an object-based interface allowing the communication between the end user and the storage device via objects. While block-based interfaces are restricted to operations retrieving single blocks from a device, object-based interfaces allow access on a more abstract way, namely by this object. The object store is optimized for high-performance storage access and replaces the block-oriented methods of the file system. The command interface of an object store includes operations for creating, deleting, opening and closing objects. In order to allow the end user reading from and writing to the object in form of byte ranges, read and write operations are provided. As an object also holds metadata, set and get functions for the metadata allow to query and modify the metadata attribute values. These functions constitute the basis for an object store interface, but the command interface can be easily extended by additional operations as required for a specific purpose.



**Figure 2.1.:** The difference between the block-oriented access of storage (left) and an object-based interface (right).

Object stores can be viewed as a light version of a file system. While a file system organizes data in a hierarchy of files, subdirectories and directories, the object store implements a so-called *flat file system* having no subdirectories. All objects are held on the same level of hierarchy. Using a flat name space eases the abstraction of how objects are addressed on physical storage. In contrast to hierarchical file systems, no path lookup is necessary and objects can be accessed very efficiently. Object stores are versatile in terms of storage management, either as a stand-alone file system or as an additional layer between a file system and a storage device. An object store is responsible for basic storage management. It performs storage allocation and manages free disk space transparently for the end user. In case only a basic functionality in storage management is required and file systems provide too many services for the intended purpose, an object store can be used as a stand-alone file system acting as a low-overhead replacement for file systems. When used as an additional layer, then the file system accesses the storage device by objects and the object store takes over the task of storage management. This separates the functionality of storage management and other file system tasks.

## 2.2. File System Concepts

File systems provide an appropriate abstraction of data in mainly two data structures the user can easily interact with: files and directories. Files are a logical unit of information modeling a part of a storage device, where this information is located. Since there can exist thousands or even millions of files, it is adequate to group related files together, so

that a specific file can be found easily. That is why the file system supports a container for files, namely directories. These directories can contain other directories, which results in a tree-based hierarchy. Files are identified by a path of this tree. Paths are resolved by traversing the name space tree, starting with the first component of the pathname until the file has been found. This traversal is called *path lookup* and is a very frequent operation in file systems. File systems are directly stored on the storage device and are defined by a specific layout holding data structures that are necessary to keep track of the internal state of files and directories associated with this file system. The layout varies from file system to file system, but often it will contain some of the following main components:

**Superblock or Header** Key parameters about the file system that are readout when the computer system starts or the file system is mounted for the first time. This information is kept in memory as long as the file system exists in the computer system. These parameters are, for example, the concrete type of the file system, the number of blocks in the file system, the size of the underlying disk and so on.

**Free Space Management** Using appropriate data structures, for example bitmaps or a list of pointers to the block locations, this part of the layout keeps track of the available blocks in the file system.

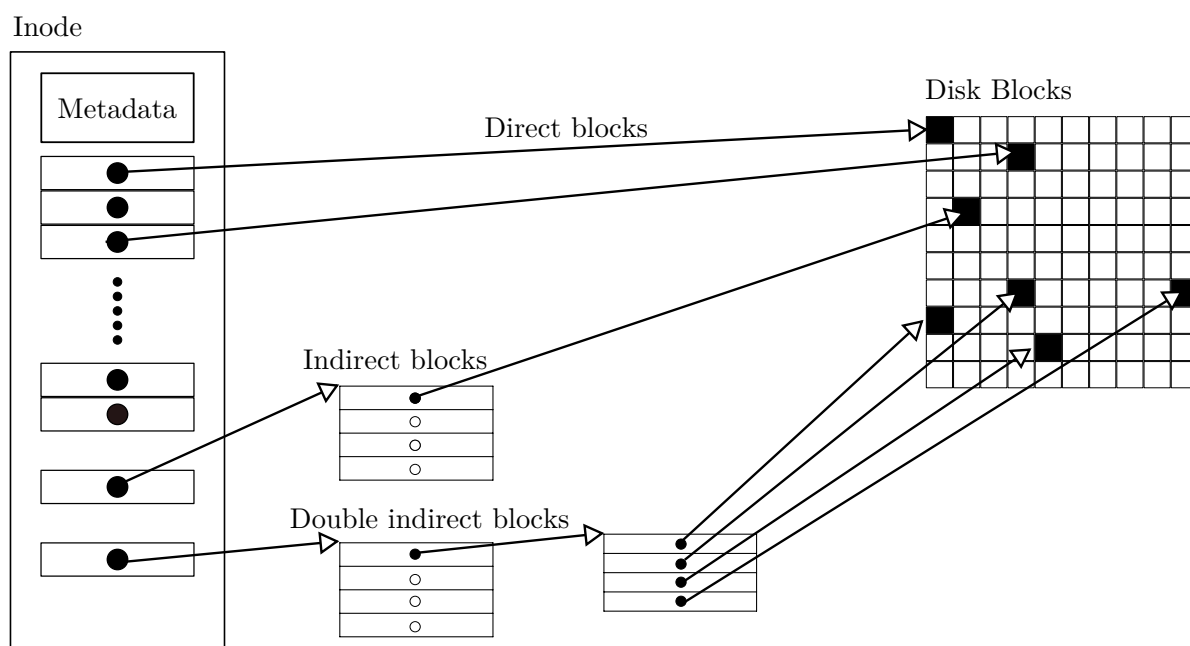
**Inode Management** Data structures holding metadata about files and directories. Each file is associated with an inode. For further details see Section 2.2.1.

**Data Block Area** The actual area of the storage device where files and directories are stored. The free space management component of the file system is responsible to keep track of the blocks of this area.

### 2.2.1. Inodes

A file consists of two parts: the actual data and metadata describing the content of the file. The metadata of a file is stored in an inode, a comprehensive descriptor for every file and directory. This data structure is described in the following. An inode maintains information like the size of its corresponding file, timestamps storing the time of creation and last modification, credentials about who is allowed to modify this file and so on. The only information that is usually not contained in the inode structure is the file name. Instead of maintaining the file name, which is a string of characters and only a convenient way for the user to deal with files, each inode is associated with a unique number, called the *inode number*. When accessing a file via its name, the file system translates this string to the corresponding number and uses it to find and to obtain the inode. The reason for using the file's inode number instead of its name is that the file can be available under different names.

Inodes are usually maintained in an inode table somewhere on the disk. The table is created together with the file system. The number of inodes the file system is able to hold may be fixed with the creation of the file system or can be managed dynamically



**Figure 2.2.:** The inode pointer structure implemented by the ext file system [CCDSP01].

during the lifetime of the file system. The inode number is used as an index of the table, so that the inode and its corresponding meta information can be retrieved directly from the table. From the user's view, a file appears to be a continuous stream of bytes, but the file and its blocks may not be contiguous on disk. For this, specific links or pointers to the data blocks related with a file have to be stored in the inode beside the other metadata attributes. In the next section an example implementation of such a link is presented.

### 2.2.2. The Inode Pointer Structure of the ext File System

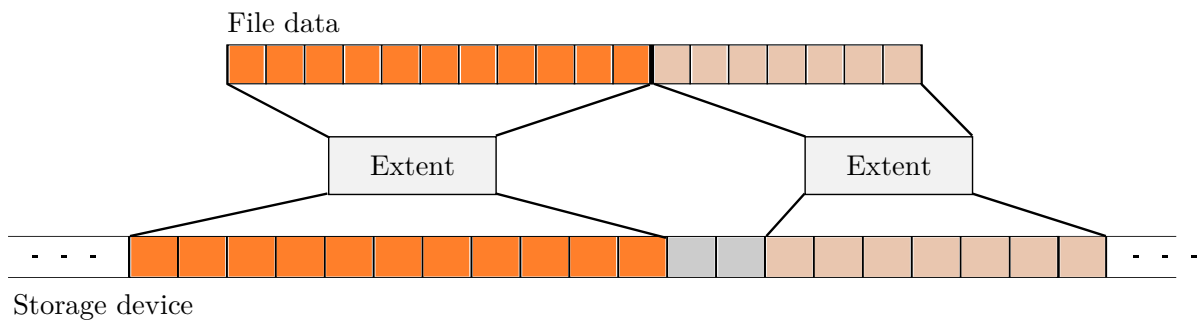
The well-known file system *ext* [CCDSP01] implements a pointer structure for storing the block locations of file data. This structure is a list of addresses pointing to a location on the disk where the file's content is stored. The list typically holds twelve pointers where each pointer addresses a block containing the data of the file. This technique is based on the assumption that all blocks have the same fixed block size.

In case of large files more pointers will be required, but allocating fixed space for more direct pointers in advance would also increase the size of the inode. Assuming a fixed block size of 4096 Bytes, the inode is able to store a file up to a size of  $12 \times 4096$  Bytes = 48 KByte. In order to overcome this space limitation, the pointer structure contains two additional pointers storing the addresses of other lists stored outside the inode structure. These lists are only created as more block pointers are required. Figure 2.2 demonstrates this technique. The first additional pointer is an indirect pointer to a list holding block addresses, whereas the second additional pointer points to a list of addresses of further block address lists (double indirect blocks). When using 8 Bytes

addresses on a 64-Bit file system and again a file system block size of 4096 Bytes, the indirect pointer is able to hold  $4096/8 = 512$  addresses to blocks that contain file data. This sums up to  $512 \times 4096$  Bytes = 2 MB of data for one file. The double indirect pointer, pointing to lists having the same size as the list the indirect pointer refers to, allows to store  $512 \times 512 = 262144$  block addresses and thereby 1 GB of data. If this is still not a reasonable amount of data, also triple-indirect blocks can be used.

### 2.2.3. Extents

Up to here, user data was assumed to be stored in single blocks that can be scattered non-continuously over the disk space. Another approach to store file data is to use so-called *extents* of blocks. The main difference to the direct and indirect block mapping scheme is that a range of consecutive blocks is addressed with one pointer, instead of a single block. The pointer is given as a tuple containing the starting address of the first block and the size of the extent, given as the number of consecutive blocks. Using extents instead of managing pointers to single blocks decreases the overhead of managing block locations. Since an extent is characterized by a start address and the length, it is not necessary to keep track of every single block. This is particularly advantageous for large files that may even require triple indirect pointers with a single block addressing scheme. Figure 2.3 illustrates this method. In this example the file has two extents, having a size of eleven and seven data blocks, respectively. While the two extents cohere in the file, this is not the case on the storage-device level. Data not belonging to the file (gray) separates the two extents. Applying the extent allocation method can also handle *file fragmentation*, an unsuitable distribution of small chunks of the blocks belonging to a file. This may cause an inefficient use of computer storage and consequently a lower I/O rate [Mcv91].

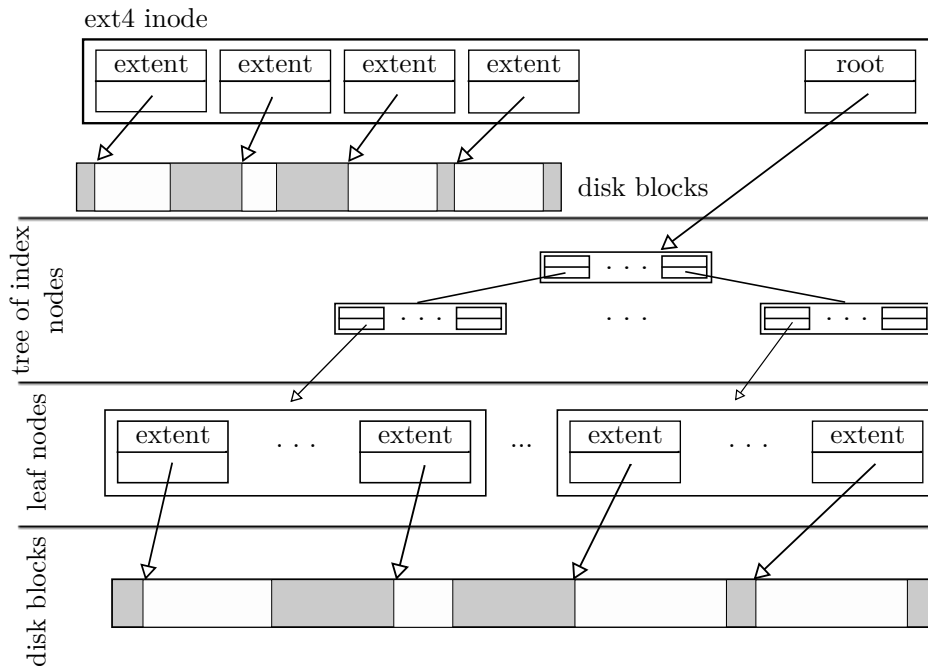


**Figure 2.3.:** Extent mapping between the data of a file and the disk blocks.

### 2.2.4. ext4 - An extent-based file system

A representative example for an extent-based file system is the *fourth extended file system* (ext4), especially when focusing on performance issues and on handling storage size constraints. Whereas its predecessor, the ext3 file system, only supports a volume size





**Figure 2.4.:** The extent tree of ext4 [MCB<sup>+</sup>07]. Once the first four extents are exceeded, a tree is spanned having additional extents as leaf nodes.

up to 16 TB, ext4 can handle volumes having a maximal capacity of 1 exabyte (EiB) [MCB<sup>+</sup>07]. ext4 uses 48-Bit numbers - in contrast to ext3, using 32 Bit - for block addressing, so that 4 billion files can be theoretically held.

Four extents are stored directly within the inode, where each extent is limited up to 128 MB assuming a block size of 4 KB. When dealing with larger or small and highly fragmented files, more extents are required to represent the file. For this purpose an constant depth extent tree is created, keeping the additional extents. This tree contains the following components (as illustrated in Figure 2.4):

**ext4 Inode** An ext4-specific data structure to hold file corresponding metadata. It contains links pointing directly to four extents of disk blocks. In addition it contains a link pointing to index nodes. The latter makes up the root node of the tree.

**Index Node** Internal node of the tree, means neither the root node nor a leaf node. It contains a data structure *extent index*, pointing to the next children node, either another index node or a leaf node. It does not point to locations on the disk drive.

**Leaf Node** A node containing the extent data structures, pointing directly to the extents of disk blocks.

Each node of the tree begins with a header, containing information about the valid entries in the node following the header (index or extent data structure), the maximum

number of these valid entries and the (current) depth at which a node is placed. This information is important concerning the construction of the extent tree.

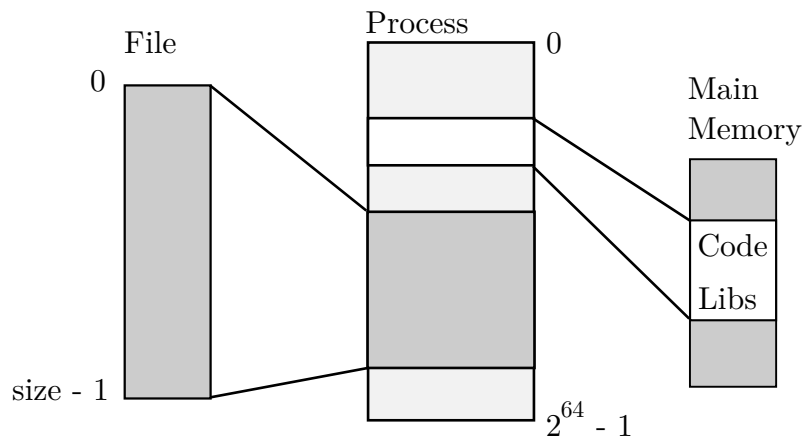
For improving file system performance of the extended file system series, a lot of optimizations have been developed concerning the block allocation methods. These optimization techniques work well together with the extent mapping scheme. Two of them are important to be mentioned here: multi-block and delayed allocation. Instead of allocating a single block at one time, multi-block allocation searches free groups of blocks, where data can be allocated contiguously on disk. Delayed allocation means that blocks are not allocated immediately when the file system recognizes a write request. The pending data is buffered in memory and then flushed to disk when memory space is getting low. Only then the free-space bitmap will be updated. Combined together with multi-block allocation, delayed allocation promises a decrease of file system fragmentation and, consequently, a higher throughput.

## 2.3. Memory Mapping

The object store designed and implemented in this thesis is based on memory mapping. The concept of memory mapping will be motivated in the following:

When a process has been started, a portion of memory is allocated that can hold the data needed to run the program associated with the process. This portion of memory is defined by two address spaces: first, the logical address space, also called *virtual memory*, seen by the program and secondly, the physical address space constituting the actual space given by the real system memory. Virtual memory appears to be contiguous to the process, while the corresponding physical memory does not need to be contiguous. Virtual memory is divided into fix-sized blocks, the so-called *pages*, a contiguous range of addresses. The physical address space is also divided fix-sized blocks, called *page frames*. Because of this division, it is possible to map parts of the process address space into secondary storage as long as they are not currently needed to run the program. In order to determine which virtual address corresponds to a physical address, each process manages its own page table that maps virtual pages onto page frames. When a part of the logical address space is referenced that is not located in the main memory, a page fault occurs and the operating system has to load the corresponding pages from secondary storage. This kind of management of virtual memory is called *paging*.

The paging method is implemented by memory mapping, more precisely it implements demand paging. Demand paging does not immediately load all parts of a program into memory when the process has been started. Similar to this, memory mapping only loads the metadata of a file and not the whole content of the file into main memory. Consequently, if a page is read or written, this produces a page fault, the page is loaded then into main memory and the page table is modified not to fault for this page at a later time.



**Figure 2.5.:** A file being loaded into memory of a process by memory mapping.

Memory mapping maps a file or a portion of it directly into the virtual address space of a process, so that the file can be directly accessed as if it was a byte array like it is illustrated in Figure 2.5. In this example a 64 Bit architecture is given, so that the virtual address space of a process ranges from 0 to  $2^{64} - 1$ . The file being mapped into the virtual memory of the process has an arbitrary size. The virtual memory additionally holds code and libraries (libs) in order to execute a specific program.

The operating system uses a kernel page cache in order to handle the pages automatically and efficiently. That means it directly manages which pages have to be loaded into memory and which can be written back to the storage device. Consequently, read and write system calls become unnecessary in order to access data on the disk.

Linux provides the system call `mmap`. This function uses the kernel page cache of the operating system. That is why no cache has to be implemented when using `mmap`.

## Chapter Summary

Object stores can mainly be found in parallel file systems and cloud computing systems. In this thesis, object stores are regarded in the field of parallel file systems namely as a low-level abstraction of storage. User applications and file systems access storage via objects rather than single blocks.

A file system is mainly responsible for organizing data on storage devices in a hierarchy of files and directories. Files consist of data and metadata (“data about data”). Metadata of a file is kept in an inode data structure. Moreover the data locations are stored in these data structures. These are often stored as pointers, either to single blocks or to a range of consecutive blocks. The latter is called an *extent*.

Memory mapping provides a useful way to access files. This technique maps a file into the virtual address space of a process and allows array-like access to the file.



# Chapter 3.

## Object Store Design

As stated in the introduction, the goal of this thesis is to design an object store that is able to perform basic storage management either as a light-weight file system or as a supporting layer for another file system. In order to design an object store, several requirements and design aspects have to be considered.

The chapter starts with a description of general design decisions. This defines the technical and conceptual principles used for the further realization of the object store. An important aspect is the concrete on-disk layout of the object store, that is, the data structures needed to provide information about the object store and how they will be placed on the underlying storage device. Objects constitute the storage abstraction of the object store. In this chapter their internal representation as well as their high-level abstraction are designed. An Application Programming Interface (API) is the fundamental part of the object store in order to allow the communication with the objects and the object store. It will be described in more detail what kind of operations are required. Furthermore, two features are presented that are added to the basic API, namely synchronization and transactions. The chapter concludes with a description of related work about other object stores and compares them with the presented design of the low-level and extent-based object store. Moreover, it emphasizes the need for a new object store.

### 3.1. General Design Decisions

Overall, five basic design decisions are of special interest. The first two refer to conceptual design of the object store:

**Low-Level** The object store is designed to be *low-level*: It should have direct access to the raw block abstraction of a storage device, meaning that it manages how data is stored on the disk and has no additional layer (for example, another file system) between it and the device. This direct low-level access method offers more possibilities for custom improvements and modifications, for example, in data structures and allocation algorithms.

**Extents** Objects are associated with data spread over the entire disk. That is why the different block locations have to be managed. In this object store design the extent

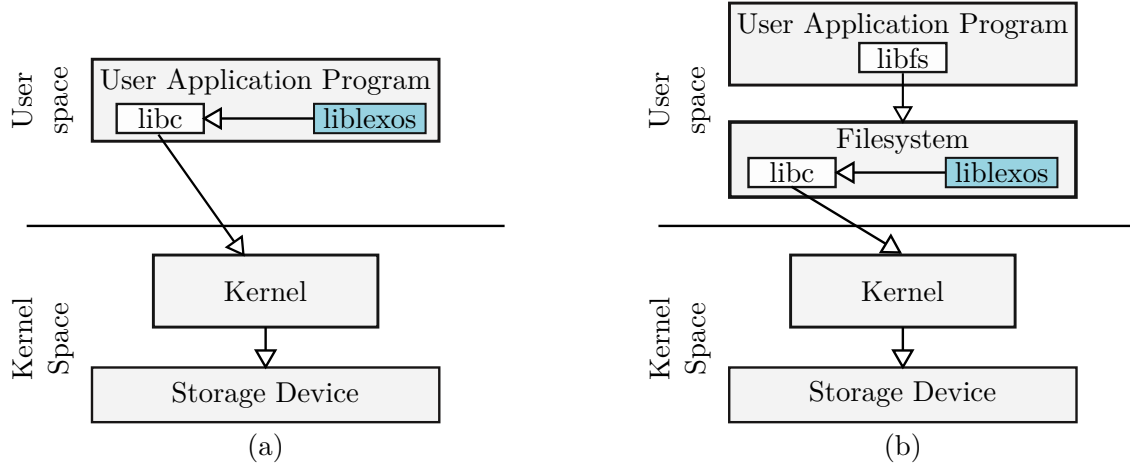
mapping scheme will be used. This mapping scheme promises advantages in terms of a lower fragmentation rate and a lower block management overhead (see Section 2.2.3). It is necessary, to design an appropriate extent managing data structure holding all extents belonging to one object (see Section 3.2.2).

In summary, a low-level and extent-based object store is designed in this thesis. From now, the object store is referred to as *LEXOS* (an acronym created from **low-level extent-based object store**). Three further design decisions are of interest:

**Memory Mapping** Another design decision is to use memory mapping instead of the I/O system calls like `read` and `write` for several reasons: The device is directly mapped into the virtual address space of the process, so that the part that has been mapped can be accessed easily. Other system calls like `read` and `write` require a lot of communication between the several layers located in user and kernel space, while memory mapping works directly with the page cache in the kernel. That is why fewer communications between the layers are necessary.

When using memory mapping, no dynamic allocation of main memory for each data structure is required (for example with the C-function `malloc`). The whole device and all the data structures located on it can be simply accessed like a big byte array of memory. If the device would not be mapped into main memory, the data structures first had to be read from disk, put into the allocated space of memory and are then written back to disk. After this, the allocated memory would have to be freed again. As it becomes clear, a lot of disk accesses would be necessary. Using memory mapping, the data structures can be directly stored on the mapping and are automatically written to the device when the mapping is synchronized with the disk. Since no dynamic memory allocation is necessary, the chance for memory leaks in the library is reduced. Memory leaks occur due to the incorrect handling of memory that has been allocated and not been released when the program terminates which may lead to a crash of the program.

**User Space** System memory in Linux-based operating systems can be divided into the kernel space and the user space. The kernel space executes programs and services of the operating system, while in user space application programs are executed. A file system or object store can be implemented for one of the two spaces. Consequently, it has to be decided for which space the object store should be designed for. In kernel space, I/O operations are implemented as so-called kernel modules. That raises several challenges. The kernel does not provide a stable programming interface. The kernel interface can change with a new kernel release, so that the object store implementation based on the interface of the previous kernel version is not compatible anymore. Consequently, the object store has to be adapted to the new kernel version, implying a great effort in maintenance and a restriction in flexibility of the object store implementation. Moreover, an interface to the object store has to be provided, conforming the abstract interface of the Virtual File System (VFS) [BHB99] and by eventually implementing new system calls. This also raises a great effort in the implementation of the object store.



**Figure 3.1.:** Usage of LEXOS in the user space. (a) A user application program linked with the object store library *liblexos*. (b) A user application program linked with a file system library *libfs*. The file system library is linked with the object store library.

All in all, programming in kernel space is a slow and difficult process [SDA<sup>+</sup>10]. Programming in user space eliminates the issues described above. For this reason, the object store is supposed to be designed for user space. In order to communicate with the underlying storage device the system level I/O interface provided by the kernel can be used. Under Unix-like operating systems this is the Portable Operating System Interface (POSIX) [Ope], a standard for system call interfaces. This interface is supposed to not change for different kernel versions.

Figure 3.1 illustrates how the object store library (depicted as *liblexos*) can be used in the user space. The object store library accesses the storage device via function calls provided by the library *libc* [GNU13]. This library calls the corresponding system operations in the kernel. Figure 3.1 (a) illustrates the object store as a library linked in a user application program. In this use case the object store is executed as a stand-alone light-weight file system. In Figure 3.1 (b) the object store is linked in another file system that uses the object store as a storage management unit.

Because of this decision, the object store can then only be used by file systems implemented in user space. More and more file systems are decided to be designed user space, what is particular motivated by the recent developments of the framework FUSE (File system in User space) [RG10]. That is why the object store can be used by a range of file systems that are implemented in user space.

**Thread Safety** Execution by multiple threads at the same time should be supported. As the memory mapped device acts like shared memory, critical sections in the code have to be locked appropriately. Multiple threads can share the same object or have their private object.

## 3.2. Data Structures and On-Disk Layout

This section focuses on the specific on-disk layout of the object store, that is the data structures required to successfully manage the underlying storage device.

The storage device the object store will reside on, the storage device is logically divided into fixed-size blocks. These blocks are consecutively numbered, where each number represents a unique block address. The first block is addressed by zero. The amount of addressable blocks depends on the size of the storage device and the given block size. The desired data can easily be accessed by an offset calculated by a multiplication of the block number and the block size. In order that the offset calculation stays consistent, it requires the block size to be fixed. After the creation of the object store the block size is supposed to be never changed as long as the object store exists. The task of the object store is to organize those fixed-size blocks. In order to do that, the concept of objects is used. That is why the object store mainly has to manage two instances: the objects, internally represented as inodes (see Section 3.2.1) and, secondly, the data blocks belonging to a particular object. As it will become clear in the following sections, four data structures are necessary to successfully manage blocks and inodes:

**Header** Contains general meta information about blocks and inodes (number of available blocks/inode, next available block/inode, block size etc.).

**Inode Table** Follows the header and stores the inodes metadata and links to the data blocks.

**Inode Bitmap** Data structure tracking the allocation state of the inodes.

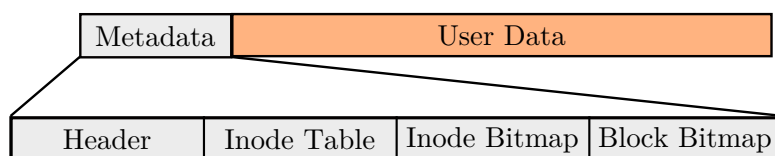
**Block Bitmap** Data structure tracking the allocation state of the blocks.

These data structures are directly stored on the storage device forming the on-disk layout and storing meta information relevant for keeping track of the current state of the object store. Once the object store is created on the underlying storage device the layout is written and can not be changed anymore. That is why the layout has to be considered thoroughly in advance. This includes the design about how the address space of the underlying storage is divided and which parts of the device hold metadata and actual user data. In this work, only one storage device is used. Its address space is divided in a straight-forward way by letting the metadata start with the first address on the disk. The ordering of the chosen data structures is depicted in Figure 3.2. The remaining data blocks that can be used to store actual user data are placed directly next to the block bitmap. In the following sections, it will be motivated in more detail why these data structures are important to allow a successful management of inodes and blocks.

### 3.2.1. Objects and Inodes

The object store allows storage to be accessed by *objects* hiding the raw block representation of the device and internally handling and managing the data they are related with. This requires an appropriate data structure containing metadata attributes relevant for





**Figure 3.2.:** The object store layout like it is installed when the store is created on a device.

both the user and the internal workings. Another important point is the management of the data blocks associated with this object. Since the user does not need to care about where and how data is stored, these links to the blocks are required internally, so that data can be found on the disk. For this purpose, the inode is a suitable data structure. As described in Section 2.2.1, inodes are used to represent the meta information of a file and its corresponding data by a pointer structure. Hence, an object is internally represented by an inode in the object store.

An inode associated with the low-level and extent-based object store should contain the following metadata attributes:

**ID** Inode identification number.

**Extent Information** Links to the extents belonging to this inode.

**Allocated Size** Resulting size of the inode/object summing up the blocks of all extents of one object.

**Object Size** Amount of extent space that is already associated with data.

In order to access each available inode, they are stored in a array, the *inode table*. By indexing the table, the corresponding inode can be accessed and the object-related metadata is extracted. For this purpose, each inode is identified by a unique number additionally used as the index of the inode table where the inode is stored.

Inodes have to be allocated before they can be used, and if they are not required anymore, they are freed and set as unused. That is why a data structure is needed that can track the allocation state of inodes. A convenient and easy way to track the allocation state is to use *bitmaps*. A bitmap is a list holding several bits, each being associated with exactly one inode. An inode can either be allocated or unallocated. These states can be easily tracked by a bit: An allocated inode is encoded by a bit value of one, an unused inode by a zero value, respectively. The inode ID corresponds the position of the bit associated with this inode in the bitmap.

An object can be allocated with a specific size in advance, even though this amount of data is not required at the moment, but probably in the future. This size is correlated with the *allocated size*, while the size of actual data corresponds to the *object size*. The object size can be smaller than or at most equal to the allocated object size since it is the resulting size when extents are filled with data.

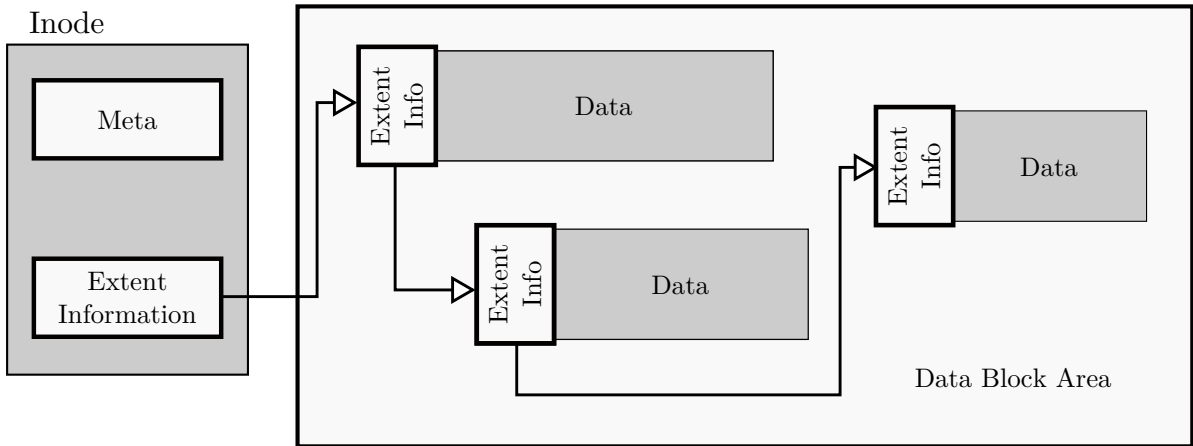
In order to store the extent information associated with an inode, another complex data structure is needed. This will be explained in the following section.

### 3.2.2. Extent Representation

The object store is designed to store related data in extents, a range of consecutive blocks. There are a lot of methods and data structures that can be used to store extents. As described in Section 2.2.3, extents of a file can be organized in a tree structure having the root node stored directly inside the corresponding inode. This approach is a promising solution concerning the performance, because finding an item in a balanced tree only has a worst case time complexity of  $\mathcal{O}(\log(n))$ , where  $n$  is the number of nodes in the tree. However, a balanced tree like the rather complex H-tree data structure has expensive operations - deletion and insertion - due to the steady maintenance of tree balance. This overhead in management is unsuitable for a first solution.

For the object store prototype implemented in this work the extent data structure is to be kept simple. The basic approach used is the singly-linked linear list. There can exist several extents for an inode having a specific order and additionally the extents are spread over the whole storage device. A list data structure is a suitable way to represent the relations between the extents.

The extent data structure is illustrated in Figure 3.3. The extent information of an inode holds a pointer to the first extent of the object. This extent consists of the data associated with the extent and metadata again called extent information (extent info in Figure 3.3). The extent information of one extent includes a pointer to the next extent of the object, so that the extents combine to a singly-linked linear list.



**Figure 3.3.:** The extent list data structure. It constitutes the internal representation of object-related data.

The information about the consecutive extent is always stored directly inside the preceding extent. Consequently, the first bytes of each extent are reserved to hold the start address and the size information about the next extent.

A new extent is always added at the end of the list. Without knowing the start block address of the last extent, the worst case time complexity is  $\mathcal{O}(n)$  - with  $n$  being the length of the extent list - because the whole list has to be scanned until the last extent is found. Hence it is reasonable to store the start block address of the last extent to reduce

the search time when an extent is going to be added. The start block address of the last extent can simply be stored inside the inode as an additional internal attribute. So when the index of the last list element is known, the worst case time complexity decreases to  $\mathcal{O}(1)$ , since inserting an element into a list is a constant time operation.

However, finding a specific offset within an object is still a time-consuming operation, because list traversal is necessary. In order to decrease the search time, further improvements on this data structure are necessary that are discussed in Section 6.2.

### 3.3. The LEXOS Application Programming Interface

An *Application Programming Interface* (API) is provided by a library to abstract from its internal implementation and to give the user the possibility to interact with a component of the library. It is a well-defined interface in form of a collection of functions, variables, constants and data structures that are necessary for a communication between the user and the library.

There are two instances associated with the object store the user can interact with: The object store itself and the objects that are embedded in the object store. Consequently, the library has to provide an API for these two instances, so that a communication with the objects and the object store is possible. The interface has to be designed, so that the necessary tasks related with the object store can be performed. This raises two questions:

1. Which basic functionalities have to be provided by the API and
2. which attributes of the objects and the object store should be accessible by the user?

In the following these two questions are discussed by analyzing the requirements concerning the object store and objects:

**Objects:** An object can be *created* within an existing object store. It can not exist outside an object store and only in exactly one object store. In case the user knows roughly in advance what size the object is supposed to have, a *preallocation* of a given size can be invoked with the object creation. If an object is not needed any more it can be *deleted*, so that the corresponding inode is released and can be used for other objects. A similar operation *closes* the object, but without freeing the inode and only releasing temporary allocated resources. An object can be *opened* again if it was closed before. For this it is necessary to specify the object's ID and the object store in which the object is located. Two I/O operations are necessary, that retrieve data from the object and fill the object with data. An object can be filled with data by executing a *write* operation on it. In order to specify the operation precisely, the following parameters are essential: the object to which the data is supposed to be written, the offset to specify the location from where the writing starts, the size of the data and a buffer containing the data. In case the stored data is needed again, a *read* operation retrieves data from the object. The read operation needs the same parameters as the write operation, whereby the

data buffer allocates memory for the data that will be readout from the object. Since an object is internally represented by an inode (Section 3.2.1), it inherits all its metadata attributes. But not all of them are interesting for the user and therefore are hidden. In order to access the object, the ID is provided. Additionally, it is reasonable to store in which object store the object is located. The user should be able to query the real size of the object. Since it is possible to resize the object, it is required to query the allocated size of the object.

**Object Store:** Three use cases will be considered: creating, opening and closing an objectstore. So that an object store can be used, it has to be *created* on a storage device in advance. When an object store has been created, it defines a name space for objects that will be embedded in it. With the creation the number of inodes and the block size is set. One object store can only exist on one storage device or on a partition of a device. As an object store is created it should be possible to *close* it when it is no longer needed in order to release temporary allocated resources. A delete operation is not necessary, since the object store can be removed from the storage device by formatting it with another file system. Obviously, it should be possible to *open* an object store that has been closed before, so that it can be used again. Once an object store has been opened, it cannot be opened in parallel and, similarly, a closed object store cannot be closed simultaneously.

An object store holds a number of objects given by the number of inodes that has been set during the creation of the object store. For the user it is important to know, how many objects are available in total (in use or not in use) and how many objects can still be created. Since a user program can create and open several object stores in parallel, it can be helpful to query the path of the storage device where each object store has been created.

An object store uses a reference counter in order to track the number of objects that exist within the object store. An object that has been created or opened establishes a reference to the object store where it has been created. The object's delete and close operations remove the reference of the object to the object store. Only when all objects have been deleted or closed, so that no references to the object store exist anymore, the object store can be closed.

In summary, the use cases related with object and object store lead to the following API interface:

- **Object store**
  - `create(path, number of inodes, block size)`
  - `close(object store)`
  - `open(path)`
- **Object**
  - `create(object store, size)`
  - `resize(object, size)`

- `delete(object)`
- `open(object store, object id)`
- `close(object)`
- `write(object, offset, count, buffer)`
- `read(object, offset, count, buffer)`
- `status(object)`

## 3.4. Features

The API developed so far already provides the basic functionality described above. However, it is not necessarily restricted to the available operations. For instance, two features have already been developed, that can be easily added to the existing API. The first feature is the synchronization of data used to ensure that it will be written on disk once the synchronization is invoked. This is similar to a **flush** operation writing the content from the I/O buffer to a file. Another feature is the batching of multiple read or write requests concerning one object into one operation called *transaction*.

### 3.4.1. Synchronization

Memory mapping does not define when data is written to disk, that is why it is uncertain if data is available on the disk immediately after it has been written to the mapping. In case of a program or system failure, the data could be lost and the data is left in an inconsistent state. So it becomes necessary to synchronize data, meaning that it is written to disk after a specific operation. Synchronization can be established in two operations:

**write** Synchronize after copying a chunk of data.

**close** When an object is closed, all extents are written to disk.

Synchronization is supposed to be an optional operation when writing or closing an object, so that an additional parameter has to be provided for the write and close operation:

- `write(object,offset,count,buffer,synchronization mode)`
- `close(object,synchronization mode)`

The synchronization mode is a flag either allowing or disabling synchronization in these operations.

### 3.4.2. Transactions

The main I/O operations designed for the object store are the read and write operations that can be executed any number of times and in an arbitrary order. However, considering the scenario, where the user wants to execute a lot of I/O operations consecutively, the individual execution of the operations could be very inefficient, for example due to a repeated traversal of the extent list when a lot of small chunks of data are copied although they are placed next to each other. Therefore, it should be possible to merge the read and write requests into a transaction. A transaction is performed as follows:

- The user invokes the *start* of a transaction.
- The user lists all operations that can be possibly merged in one operation. As the start of a transaction was invoked before, the operations are not performed immediately but stored in a waiting queue, pending for further processing.
- Having listed all the necessary operations, the user invokes the *execution* of the transaction which starts the processing of the pending operations. The object store analyzes which operations can be merged in one operation and performs the transactions.

For this it is necessary to analyze under which conditions two requests can be merged. Assuming  $n$  write or read requests  $r_i$  ( $i = 0, \dots, n - 1$ ), where each request is defined by a type (read or write), a synchronization mode, a count  $c_i$  of bytes that should be processed and an offset  $o_i$  pointing to the location from where the data is written or read. Then two requests  $r_i$  and  $r_{i+1}$  can be merged under three conditions:

- C1** The offset  $o_{i+1}$  of request  $r_{i+1}$  equals the sum of  $o_i + c_i$  (offset and count of request  $r_i$ ) and
- C2** the two requests have the same type and
- C3** the two requests have the same synchronization flag.

In order to enable the user to invoke a transaction, it follows from the description above that two further API operations are necessary, namely:

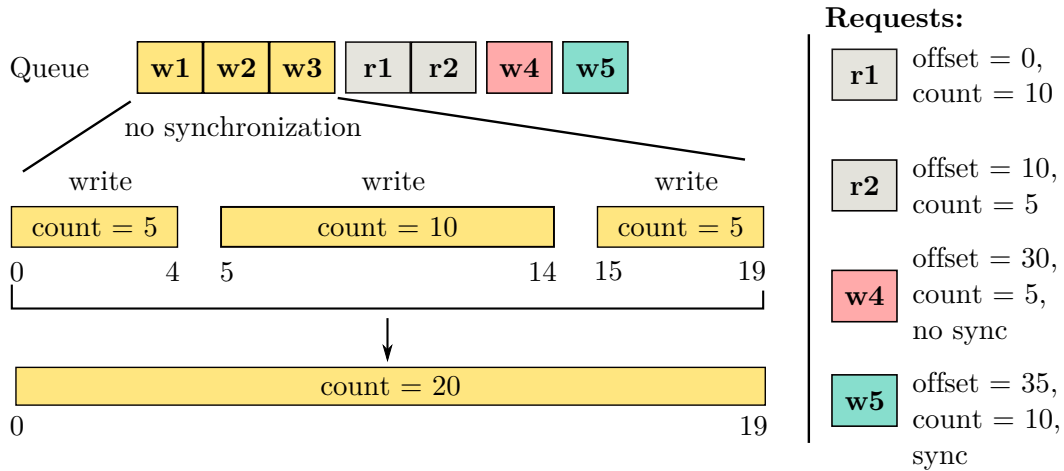
**transaction\_start(object)** Start a transaction for a specific object. The incoming operations are collected in a queue and no data is written/read to/from the disk immediately.

**transaction\_execute(object)** Execute the transaction for this object after providing the desired operations.

Having stated the three conditions for merging operations and designed two additional API operations to invoke a transaction, the internal working of the transaction process can now be considered. Given that the queue contains all operations the user wants to execute, the queue can now be processed. Taking and simultaneously removing the first

element from the queue defines a main request holding the offset  $o_i$ , the count  $c_i$ , the buffer with data to be written, the type of the request  $r_i$  and the synchronization mode. This information is necessary for checking if the three conditions are met. If so, the count of the main request is updated by summing up the counts of both requests. Additionally, the data buffers are merged into one held by the main request. The resulting request is then used in order to be compared with the next pending request in the queue after the processed request has been removed from the queue. This is repeated until a request has been found that does not meet the conditions and the resulting request can now be executed by calling the corresponding operation defined by the type given in the main request. After this the next main request is picked from the queue and the process is started all over again. If the queue is empty, meaning that all operations have been processed, the execution of the transaction is finished.

Figure 3.4 gives an example for a transaction execution and illustrates how these operations are stored and processed in a queue. The example shows read (r) and write (w) operations that can be merged. This is motivated as follows: The first three write requests (w1-w3) can be merged, since all three conditions (C1-C3) are met. The same holds for the following two read requests r1 and r2. The last two write requests w4 and w5 cannot be merged as they do not have the same synchronization mode. Consequently, the last two write operations stay individual operations.



**Figure 3.4.:** Merging of the first three write request into one operation with offset 0 and count 20.

### 3.5. Related Work

Only a few object store solutions exist, each using different design approaches. These object stores have in common that they are all designed for a specific file system. Three

file systems (Ceph, ZFS and hFAD) and their corresponding object store solution are presented in this section. Aspects that are of particular interest with regard to the comparison with the LEXOS object store design are emphasized. This especially concerns the representation of data (single block or extent mapping scheme), block or extent allocation techniques and the object storing data structures.

### 3.5.1. EBOFS

The object store that has come closest to the design of LEXOS described in this chapter is EBOFS, an extent and b-tree based object file system originally developed for Ceph, a scalable, high-performance distributed file system. Ceph was designed for a maximum separation between data and metadata management. Similar to LEXOS, EBOFS interacts directly with a raw block device and provides its own non-standard low-level object storage interface [WBM<sup>+</sup>06]. EBOFS is developed for user space to avoid cumbersome interaction with the LINUX VFS and page cache [WBM<sup>+</sup>06]. This agrees with the design decision of LEXOS.

EBOFS also uses extents as a mapping scheme. However, free block extents are organized in another way. Instead of using a bitmap where each block is encoded by a single bit, free block extents on disk are binned by size and sorted by location, so that free space near the write position can be quickly located and long-term fragmentation can be limited [WBM<sup>+</sup>06]. This was also considered for LEXOS, but because of the additional management overhead (updating size of bins and sorting) this feature is left for future work and further improvements.

The central design decision of EBOFS are b-trees for locating object-related data on disk. As already justified for LEXOS, the data structure that manages the extents should be kept simple and easy to manage. That is why, a tree-based data structure was omitted for the first design.

Another feature of EBOFS is that it aggressively performs copy-on-write operations, meaning that data is always written to unallocated regions of the disk. This helps data to be always in a consistent state in case of a system failure.

Ceph now works with another object store implementation (Reliable Autonomic Distributed Object Store) internally managing storage devices using a file system as an object store [WLBM07]. So, the object store of Ceph is no longer a low-level object store. The development of EBOFS has ceased and the source code is not provided any more. That is why EBOFS can not be used and there is a need for a new low-level object store.

### 3.5.2. ZFS

Another file system that uses the object store concept is ZFS. ZFS implements a pooled storage architecture allowing several file systems sharing one collection of multiple storage devices. Using the pooled storage architecture should ease adding a new file system without assigning it to a specific storage device. Additionally, when a new storage device is added to the pool, its blocks can be used immediately by the file systems mounted on the pool. Each file system can use as much storage as it needs.



ZFS especially focuses on an efficient storage management. This is handled by extending the regular file system architecture by mainly three components: the storage pool allocator (SPA), the data management unit (DMU) and the ZFS POSIX Layer (ZPL). The ZPL is the highest layer, implementing a POSIX file system. The ZPL communicates with the DMU. Roughly speaking, the DMU is comparable to the object store concept. It provides the ZPL an object-based interface. An object in ZFS is simply a piece of storage provided by the SPA which is the layer directly accessing the raw block storage devices and responsible for allocating new storage devices to the existing pool, so that file systems can access it. The SPA also constitutes the layer managing block allocation and deallocation when requested by the DMU.

Data in a pool is organized in a tree, having a root node, called the *uberblock*, indirect nodes pointing to data block locations and leaf nodes containing the actual data blocks. On-disk data is kept consistent at all times by using copy-on-write. When a data block is modified, a new block is allocated somewhere in the pool and the content of the modified block is written into that new block. Since the location of the old block has changed, the indirect nodes pointing to that old block also have to be updated.

Storage is logically divided in blocks of variable size instead of blocks having all the same size. The design of ZFS refuses the use of extents, since they are not compatible with the copy-on-write techniques used by ZFS [BAH<sup>+</sup>03]. Moreover this design decision expects the same performance benefits as it could be achieved from the extent-based approach. The DMU offers an interface that allows other file systems to integrate the DMU in their environment, so that the object-based communication with the underlying storage device can be applied. However, the interface is still designed to fit best with ZFS, rather than to be a general-purpose interface. The integration of the DMU into other file systems was already tested. But the ZFS library turned out to be not applicable for other file systems, especially in user space. Moreover, the DMU interface is sparsely undocumented what still more complicates its application. Therefore, ZFS is neither a well-suited object store solution.

### 3.5.3. hFAD

The hFAD file system (abbreviation for “hierarchical File Systems Are Dead”) implements a completely new approach of how a file system can be viewed. Instead of using the concept of the hierarchical name space, the architecture is adapted to user expectations moving away from organized data to search-based data [SM09]. This new approach also uses an object storage managing the underlying storage that the user can access via objects. As designed for LEXOS, objects are identified by a unique ID.

Like LEXOS, object-related data is allocated and stored in extents. In order to be able to locate objects and their data, they are stored in a database (Berkeley DB) by using B-trees storing associated keys and data pairs. The keys are file offsets where the extents begin and the data items are disk addresses and lengths corresponding to this offset [SM09]. A NULL key value stores object-related metadata in the B-tree.

Block allocation is performed by the lowest layer of the object store. This layer implements a so-called buddy storage allocator, a memory algorithm dividing storage into several

chunks and trying to satisfy a storage allocation request as fast and suitably as possible. As it became clear, hFAD divides the functionality of its object store into several layers. For example, the storage management is completely organized by the buddy allocator. As opposed to this, LEXOS implements one component responsible for all tasks related with storage management. This reduces the chance of incompatibility between several software components involved in the object store implementation.

## **Chapter Summary**

The design of the object store involves several aspects. As it is responsible for storage management, appropriate data structures have to be designed. Some of them are adapted from the file system concept described in the previous chapter. In order to represent objects, inodes are used. Moreover it has to be tracked which inodes and data blocks are available in the object store. For this, bitmaps are applied. Stored on the disk, these data structures form the on-disk layout of the object store. The object store uses the extent mapping scheme in order to store and track object data. Additionally, an application programming interface (API) is designed. With this interface an object store can be created, opened and closed. Moreover it allows the manipulation of objects. The API provides the transaction feature, allowing several write or read operations to be batched in one operation. Moreover it provides synchronization. Data is written to disk immediately after the write command when the synchronization feature is activated. The comparison to other object store implementations has shown that LEXOS is a promising approach that deserves implementation.

# Chapter 4.

## Technical Realization of the Object Store

The previous chapter described what kind of data structures and operations are needed in order to provide a basic functionality of the object store library. These are the data structures permanently stored on the storage device. From now, these data structures are referred to the term *on-disk data structures*. In addition to this, data structures are necessary that are held in memory during the runtime of the object store, referred to the term *memory data structures*. In summary the following data structures are necessary and can be classified by these two categories:

### On-disk Data Structures

**Inode** The data structure constituting the internal representation of an object.

**Extent** The information about a specific extent. An extent is part of the data of an object and can be allocated and freed.

**Bitmap** Tracking the allocation state of inodes and blocks of an object store. Bitmaps can be scanned for available bits and the bit values can be manipulated by bit masking operations.

**Header** A data structure holding information concerning the object store itself.

### Memory Data Structures

**Object Store** The temporary representation of the object store during runtime as a data structure. It holds information necessary to perform object store related tasks and gives the user a high-level abstraction of the object store.

**Object** Data structure being the high-level abstraction of storage.

**Request** This concerns the transaction feature. A request is created when a transaction is invoked.

The design suggests an object-oriented implementation. The object-oriented paradigm describes a program as a collection of interacting instances that have specific attributes and a set of methods that are used to invoke a specific behavior of these instances. Each of these components stated above constitutes a separate programming unit. All these

components are related with a set of operations that can either be used by other units or are only for internal use.

This chapter starts with general issues concerning the implementation concept and the software environment. Then the implementation of the library is described by focusing on algorithms and functions that are crucial regarding the functionality of the object store. This chapter is organized bottom-up: Bitmap algorithms and the extent allocation procedure are explained first and are used as a basis for further explanations on object and object store operations. Details on the implementation of the transaction feature conclude the chapter.

## 4.1. Software Design and Environment

The whole library is implemented in the C programming language. The C programming language does not directly support the object-oriented paradigm, like for example the programming language JAVA [GJS<sup>+</sup>13] does. The available features of C can be used, so that an object-oriented implementation is reached to some extent anyways. C allows the definition of user-defined so-called *structs*. These are compound data types defined by a collection of basic data types or other structs and additionally a set of methods in order to access the members of such a struct. Structs are similar to classes in an object-oriented language like JAVA, since a class can be seen as a data type defined by a collection of other data types. The interface for each component is provided in a *header* file (.h) listing all functions and data types that can be used by other components. The implementation of the functions and data types is kept in a separate source code file (.c) that cannot be accessed and seen by other components. In order to hide the technical realization of the data structures from its users, it can be defined as a so-called *opaque data type*. The values of an opaque data type can only be accessed and manipulated by specific functions given by the interface. The opaque data type is defined in the header file (using the `typedef` identifier) and its concrete implementation is given in the source code file.

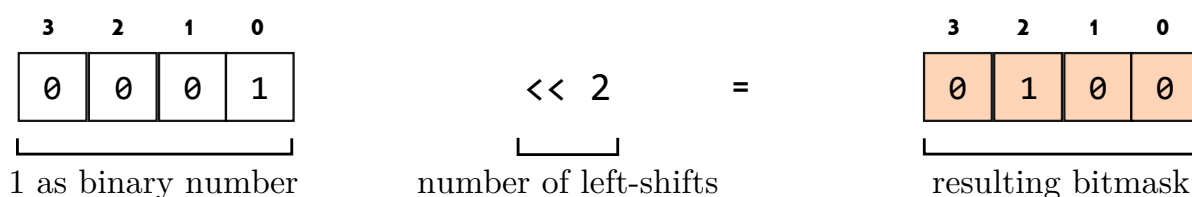
As a supporting layer for the implementation, the *GLib 2.0* [GLi] library was used. It provides a set of advanced data structures, cross-platform data types, a test environment and an error reporting system. For instance, the double-ended queue data structure (*GQueue*) provides a well-suited basis for the transaction implementation. In order to realize thread safety, the *pthread* [POS13] mutex interface is used.

## 4.2. Bitmaps

Bitmaps are used to track the allocation state of blocks and inodes. A bitmap is an array of type `guint64` (an unsigned 64 Bit integer), so that each array field holds 64 Bits and consequently, encodes 64 blocks or inodes. It is possible that more bits are available than blocks or inodes. The remaining bits are ignored.

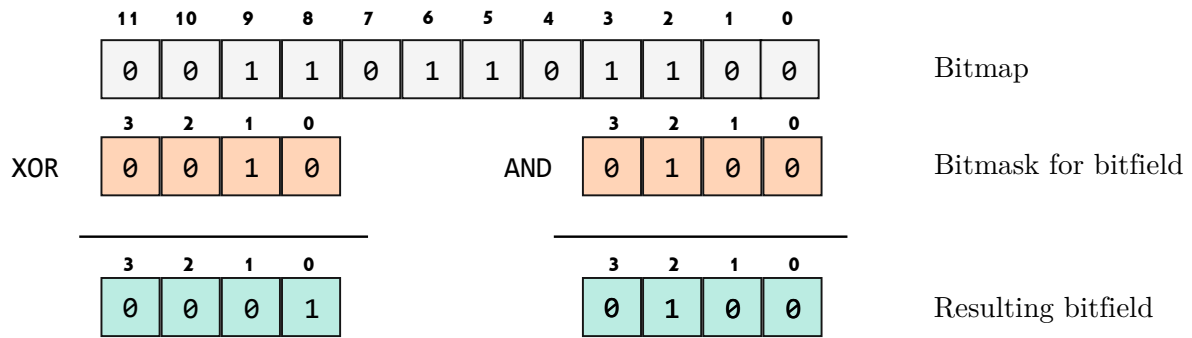
Single bits are manipulated using bitwise operations by creating bit masks having the same length as the bit field they are applied to. Given the position of the bit that will be queried or changed, the mask is created by a left-shift of the number 1 to this bit position. Bit masks are always applied to exactly one array field and not to the whole bitmap. That is why the bit position is modified, so that shift length refers to the local position in the array field. This is achieved by shifting the number 1 to the remainder of the division of the bit position and the number of bits contained in one array field; in this case 64 bits, since the array is of type `guint64`. If the data type of the bitmap is modified due to a change in the implementation, this function has to be adapted according to the new data type.

Figure 4.1 shows an example of creating a bit mask. In this example, a bit field has four bits. The number 1 is left-shifted by two bit positions. The bit field on the right side of the figure shows the resulting bit mask. Having created such a mask, it is used to query the state of a given bit or to invert its value, so-called *toggling*. Both operations are illustrated in Figure 4.2. The example bitmap contains 12 bits and each array field contains four bits. In addition to this, the indexes of each bit are depicted for both the bitmap, the bit mask and the resulting bit fields.



**Figure 4.1.:** *Creating a bit mask for a specific bit field.*

Bits are inverted by using the XOR-operator. If the bit has the value 1, it will toggle to 0 ( $1 \text{ XOR } 1 = 0$ ) and if it is 0, it is inverted to 1 ( $0 \text{ XOR } 1 = 1$ ). XOR is bit-safe, meaning that it will not affect unmasked bits, since  $X \text{ XOR } 0 = X$  where  $X$  is a bit value of either 0 or 1. This is depicted in Figure 4.2. Querying a bit corresponds to applying the bitwise AND-operator to the array field where the bit of interest is located. Using the bitwise AND and a mask only having a bit of value 1 at the position causes all other bits not being on this position to be equal 0. When the bit of interest is a bit of value 1, the resulting bit field equals a power of two. In Figure 4.2 the value of the resulting bit field equals 4, since the bit of interest is of value 1 and the index of its position in the bit field equals 2. If it is a bit of value 0, the value of the resulting bit field also equals zero. Consequently, the function either returns 0 or a value being a power of two. In order to find free blocks or inodes, the query operation can now be used to find zero bits. The start bit position is given as a parameter to the procedure. Beginning at this bit position and iterating over the following bits, the status of each bit is queried until a bit has been found having the value 0. As already mentioned, there can be more bits available than inodes or blocks. That is why an upper limit is set, specified by the maximum number of blocks or inodes. When no bit of value zero has been found until this index is reached, the start position is reset to zero, and the bitmap is scanned from its first bit. It is possible to scan the block bitmap for a number of consecutive bits in order to



**Figure 4.2.:** *Toggling (XOR) and querying (AND) specific bits in a bitmap.*

find extents of a given size from a specific bit position in the bitmap. This is reached by combining the functionality of the query and toggle procedures. The bitmap can be scanned for bits having a value of 0 or 1. As long as bits are found with the specified value (each bit status is queried with the corresponding procedure), the number of found bits is incremented and the bits are immediately toggled. For this, the mask created by the query function can be used and does not have to be created again when the bit is toggled. Since it can happen that less bits than the desired number is found, the number of found bits should be returned to the calling procedure for information. The procedure for finding consecutive bits can then be called again in order to find the desired number of bits.

### 4.3. Extent Allocation

The extent information is defined as a C-struct holding two data types as depicted in Listing 4.1. The first one stores the size of the extent in a 16 Bit unsigned integer number. From this it follows, that at most  $2^{16} = 65536$  blocks can be addressed with one extent information. The size of an extent in Bytes depends on the block size. For example, if a block size of 4 KB is used, the extent can have a maximum size of 256 MB. This number of blocks is stored as `EXTENT_SIZE`. If the implementation was going to be changed so that the data type of the extent size is modified, this number has also to be changed corresponding to the size of the new data type. The second data type stores the index of the start block of the next extent.

**Listing 4.1:** *Extent Information*

```

1 struct ExtentInfo{
2
3     guint16 size; //size in blocks
4     guint64 start_block_index; //pointer to the next extent
5 };

```

In order to avoid conflicts when several threads are involved, the whole allocation process is locked with a global mutex. The mutex is stored in the object store data structure. The mutex is initialized when the object store is created or opened; and it is destroyed

when the object store is closed.

The functionality of the extent allocation procedure is divided into two functionalities:

**F1** Allocating a completely new extent,

**F2** adding blocks to the last extent.

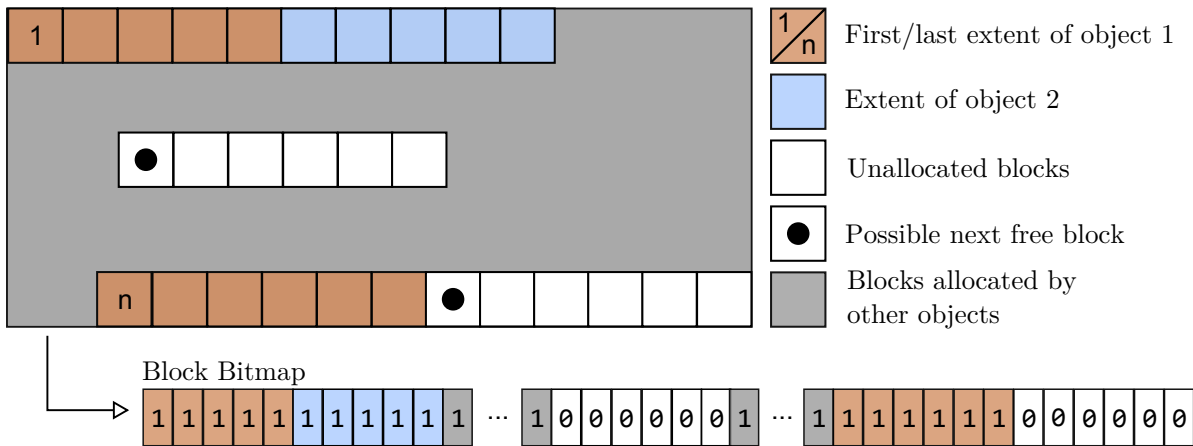
Expanding the last extent of an inode by further blocks (if possible) instead of allocating a completely new extent can reduce the number of extents held by an inode. Consequently, fewer list elements have to be traversed when searching for a specific element. Only the last extent can be filled with further blocks. Adding blocks to other extents in the list would change the offsets in the object.

The allocation procedure iterates in a loop as long as the desired number of blocks is not reached. This loop is necessary due to the maximum extent size constraint. The maximum extent size has to be considered in three situations:

1. Blocks can be added to the last extent, but adding blocks would exceed the maximum extent size,
2. the inode has extents, but no blocks can be added to the last extent and the number of required blocks is too big to be held in a new extent,
3. the inode has no extents and the number of blocks is too big to be held in a new extent.

Concerning the first case, the last extent is padded with blocks until either the maximum extent size is reached or further adding is not possible due to an allocated block. The remaining number of blocks has to be allocated in a new extent. Again in this case it has to be checked if the remaining amount exceeds the maximum extent size. If the maximum extent size is not exceeded the last extent is filled with blocks until the required number of blocks is reached. If this number was not reached, a new extent has to be allocated. In the first and second case, extents are simply allocated with the maximum extent size and since the required number of blocks is not achieved yet, the while loop is iterated again.

The block allocation procedure (F1) is executed as follows: Starting from the index of the next free block taken from the header, the allocation state of all the following bits is checked. If the blocks are unused, the bits are toggled immediately to one bits. In case that not enough blocks could be found, another extent has to be allocated. For the extent that has been allocated up to now, the extent information is updated. This extent information is stored at the start block address of the last extent currently pointing to no element. As the new extent now constitutes the end of the list, a new extent information is created at its start block address. Moreover its start block address is set as the last extent index in the inode. The number of allocated blocks is finally updated in the inode and in the header. Since the current next free block is now in use, the bitmap is scanned for the next zero bit, beginning from the index of the last block of the new allocated



**Figure 4.3.:** Two possibilities of extent allocation. Further blocks can be added to the last extent of object 1, if the next free block is right beside it. The extent of object 2 cannot be extended with further blocks. A new extent can be created for it at one of the two possible next free blocks.

extent. The whole procedure is repeated, until the number of blocks requested has been reached.

In order to decide if the last extent can be expanded by a number of blocks (F2), it is checked if the index of the next available block is right beside the last block index of this extent. Adding blocks to an extent is similar to the block allocation procedure. The bitmap is scanned for consecutive blocks beginning at the index of the next free block, which is directly beside the index of the last block of the extent. The number of found blocks may be less then the desired number of consecutive blocks. In contrast to the block allocation procedure, the adding of further blocks is not repeated if less blocks were found. Allocating the remaining blocks is handled outside this function by calling the block allocation procedure.

Figure 4.3 illustrates the two possibilities of extent allocation. It shows the data block area containing blocks allocated by objects. The corresponding bitmap depicts which blocks are free or allocated and further emphasizes the bits corresponding to blocks belonging to an object. The extents of two objects are emphasized. Two possible blocks are shown that can be available. If the next free block is right beside the last extent of object 1, this extent can be extended by six additional blocks. Given the other next free block (in the “middle” of the data block area) a new extent has to be allocated.

## 4.4. Object Store Operations

As already mentioned, an object data structure is needed during the runtime so that relevant information about it can be stored and retrieved. Listing 4.2 depicts attributes that are of special interest. It stores the addresses of the data structures that are mapped on the device. When the object store is created or opened the pointer variables are initialized with these addresses.



*Listing 4.2: Object store data structure.*

```

1 struct ObjectStore
2 {
3     Header* header;
4     Inode* inode_table;
5     guint64* inode_bitmap;
6     guint64* block_bitmap;
7
8     gpointer mapping;
9     gint ref_count;
10
11     //metadata sizes
12     //device path and file descriptor
13     //mutexes
14 };

```

Memory allocation for data structures like the header, bitmaps and the inode table is based on memory mapping (see Section 2.3). When an object store is created or opened the whole device it resides on is mapped into the virtual address space of the process. Listing 4.3 shows the corresponding function call. In order to be able to write to the mapping and retrieve data from it, the protection flags `PROT_READ` and `PROT_WRITE` are set.

*Listing 4.3: Creation of the objectstore mapping*

```

1 objectstore->mapping =
    ↪ mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

```

The address of the mapping is stored in the variable `mapping` inside the objectstore data structure. This guarantees permanent access to the mapping as long as the object store is opened. The mapping defines the common memory area for all the data structures that will be created (see Section 4.4.1). In order to create a data structure, it is simply assigned to the mapping at a specific offset. The offset defines the location, where the data structure is placed on the mapping and on the device, respectively. Furthermore, the object store implements so-called *reference counting*. If an object is created or opened in the object store it establishes a reference to it. Similarly, if it is closed or deleted, the reference is released. The object store can only be closed if no objects are opened in it. This ensures that the object store is kept open as long as objects are opened and consequently avoiding memory leaks. The number of references, the so-called reference counter is stored inside the object store data structure (`ref_count` in Listing 4.2). If no objects are opened the reference count equals 1 since the application using the object store has a reference to it. The operations that increment and decrease the reference counter have to be atomic in order to ensure thread-safe reference counting. Thus, the atomic operations provided by the GLib are used.

### 4.4.1. Create and Open

The create operation constructs all data structures necessary for an object store and puts them in a specific order on the chosen storage device (see Chapter 3). The first step is to open the device to make it accessible for the next modifications and then mapping the device as described above. The data structures are created in the same order as they will be placed on the storage device, meaning that the header is created first and the block bitmap is created at the end. Each data structure is located at an address being a multiple of the page size since memory mapping only works on entire pages of memory. Putting the data structures on page-aligned addresses promises memory mapping to load the data structures more efficiently. A data structure not placed in such a way, could produce a load of unnecessary data. This causes redundant read-modify-write of pages which content was not actually changed. Read-modify-write is an atomic command reading data from memory, modifying it and writing it back to memory. In order to achieve a page-aligned placement, a so-called *padding* is calculated and added to the actual offset where the data structure is intended to be placed. The padding is calculated from

$$p = (ps - (o \pmod{ps})) \quad (4.1)$$

with  $p$  being the resulting padding,  $ps$  the page size of the system and  $o$  the actual offset. Then, the new padded offset  $o_p$  results in

$$o_p = o + p \quad (4.2)$$

As the design of the object store in chapter 3 already illustrated, the header permanently stored at block address 0 of the disk. This address is already page-aligned, that is why no padding has to be calculated here. The header attributes are set to default values, but need to be updated during the whole object store creation process. The first attributes that can be set are the size of the storage device, the number of inodes, the block size and the path of the device as they are given as input parameters of the create operation. The following step is the creation of the inode table containing as many entries as inodes have been specified. The attributes of each inode are set to default values. The offset where the inode table is placed is calculated from the size of the header and the header padding bytes.

The inode bitmap, placed at the offset being a sum of the size of the header and the inode table - including both the padding - is initialized with zero bits representing that all inodes are available. After the bitmap creation, the next free inode index can be set in the header, that is the inode with an ID equal to zero.

The block bitmap is created with a fixed size given by the number of blocks that are available before the creation of the block bitmap. But as the block bitmap is created, the number of available blocks decreases with the size of the block bitmap, so that this number has to be updated. So, the creation of the block bitmap includes the following steps:

1. Calculate the current number of available blocks,

2. create the block bitmap,
3. re-calculate the number of blocks with due regard to the size of the block bitmap,
4. initialize the bitmap with this number of bits.

The address of the first data block following the metadata part is set to zero. Consequently, the next available data block that can be used is set to zero in the header. The number of blocks re-calculated in step 3 are set as the current available number of blocks in the header. The data block area starts at the device offset that is the metadata size including the padding of each data structure. At the end of the creation process, the object store is synchronized with the disk. This ensures the data structures to be available on the disk. Listing 4.4 shows the corresponding function call. `msync` takes three parameters: the address of the mapping, the length of the mapping and the synchronization mode. The address has to be page-aligned. The synchronization mode allows three parameters; here, the `MS_SYNC` flag is set, enabling `msync` to wait until the synchronization process has completed. The other flags and further information about the system call `msync` can be found in its man page.

*Listing 4.4: Synchronization of the whole mapping.*

```
1  msync(objectstore->mapping, dev_size, MS_SYNC);
```

In order to open the object store, the device it is installed on has to be opened and the mapping to the memory space has to be established again. The open operation reads the data structure located at the offsets calculated from the size of the data structures and the paddings as it was described above. In the open operation there is no need for a re-creation of the data structures. A new instance of the object store data structure is created and the information needed during runtime is set.

#### 4.4.2. Close

The close operation closes the device, releases the corresponding mapping and temporary resources. First it is checked, if there exist any references to the object store using the corresponding reference counting operation. If so, the object store cannot be closed. Otherwise, by calling `munmap` (Listing 4.5) the mapping is deleted for the specific address range. The allocated memory for the data structures (inode table, header and bitmaps) is released automatically with the deletion of the mapping.

Then the device being mapped is closed and the last reference to the object store (the object store reference to itself) is released.

*Listing 4.5: Un-mapping the device.*

```
1  int munmap(void *addr, size_t length)
```

## 4.5. Object and Inode Operations

An object is internally associated with an inode. That is why the object data structure contains a pointer to the inode it is related with. Each object data structure contains a pthread mutex. This is necessary, so that the object can be locked appropriately when several threads share an object. This mutex is initialized when the object is created or opened and it is destroyed when the object is deleted or closed. An object is associated with a queue that is necessary to store pending read and write requests when transactions are enabled (see Section 4.6).

*Listing 4.6: Object data structure.*

```
1 struct Object{
2     Inode* inode;
3     ObjectStore* my_objectstore;
4     GQueue* my_queue;
5     pthread_mutex_t lobject_mutex;
6 };
```

In the following sections, object and inode operations that are of special interest will be explained.

### 4.5.1. Object I/O Operations

Writing and reading data is the process of copying a specific amount (referred to as count) of data on a defined location, given as an offset, from a data buffer into the object (write operation) and from the object into the buffer (read operation), respectively. Three aspects have to be considered and should be handled carefully during the writing and reading process:

**A1** The object size can be exceeded in two cases:

**C1** The offset can be greater than the object size.

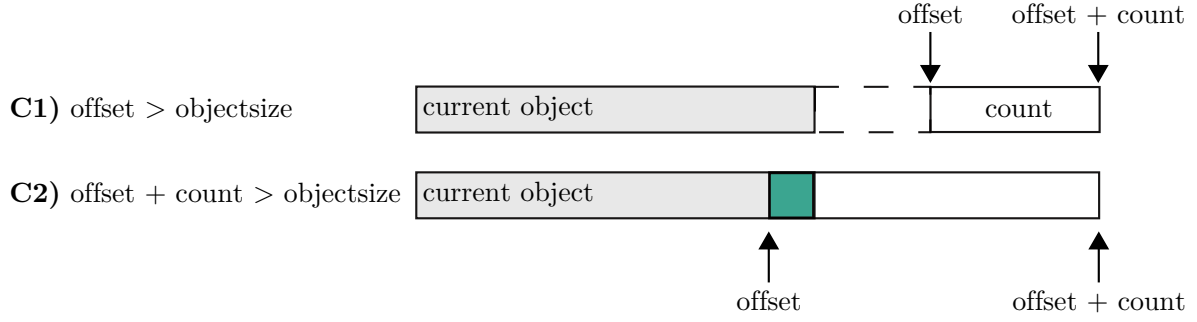
**C2** The sum of offset and count exceeds the object size.

In the present implementation this is not regarded as an error case.

**A2** Data of an object seems to be consecutive to the user, that means the user sees the object as a whole. But internally, the object is spread over the disk in non-consecutive regions. That is why the offset specified by the user does not refer to the physical offset on the storage device.

**A3** Data may have to be copied over several extents, for the same reasons as in A2.

The situations A1 to A3 are not mutually exclusive. It should be mentioned that A2 and A3 have to be handled with each write or read operation due to the spreading of the extents. A1 is considered in combination with A2 and A3; it does not necessarily occur with each write or read operation. The ordering of these cases corresponds to how they are considered in the write and read operation, respectively. The cases are handled as follows:



**Figure 4.4.:** Two cases of calculating the size of the extent needed to be allocated.

**A1** The first case should be considered first when reading or writing data. Concerning the write operation it is probably necessary to allocate new extents with a specific size for the object. Depending on where the offset is located, the size of the new extent is calculated differently. If the offset is greater than the allocated object size, the calculation can be motivated as follows: The distance between the end offset of the object and the given offset, that is  $offset - objectsize$ , represents the required size of the extent between the object's end offset and the offset. Since bytes will be written from the given offset with a size of  $count$  additional blocks have to be allocated covering this size. Therefore the size of the new extent results in:

$$size = count + (offset - objectsize) \quad (4.3)$$

If the offset lies within the object, but the count exceeds the object size the extent size is calculated as follows: Since the distance  $(objectsize - offset)$  between the end offset of the object and the local offset is already covered by an extent, there is no need for a new one in this region. Only the remaining size

$$size = count - (objectsize - offset) \quad (4.4)$$

is needed as the size for the new extent.

Figure 4.4 illustrates both cases of calculating the extent size. In the first case, where the offset exceeds the object end offset, the dashed-lined rectangle represents the part between the end of the object and the offset the objects has to be extended to. The second additional part is represented by the solid-lined rectangle forming together with the dashed-lined rectangle the resulting size of the new extent. In the second case, the blue rectangle represents the covered area between the offset and the end offset of the object. This area is omitted when calculating the size of the new extent. The remaining white solid-lined rectangle illustrates the size of the new extent. Concerning the read function, no extent allocation is necessary. If the offset is greater than the allocated object size, simply zero bytes are returned. In the other case, the amount calculated from equation 4.4 is read.

**A2** The second case requires the conversion of the local object offset into the corresponding global offset on the underlying storage device before data can be written or read.

As the object is internally mapped on extents spread over the disk, the offset has to be found by traversing the extent list and extracting the correct extent associated with this offset. Starting with the first list element, the list is traversed as long as the local offset is greater than the extent currently visited. While iterating over the extent the local offset is decreased by the size of this extent. As the offset is shrinking with every extent being traversed, an extent will be definitely reached having a size greater than the modified local offset. Then the real offset can be simply calculated from

$$\text{offset} = M + \text{startblock} * \text{blocksize} + \text{offset\_mod} \quad (4.5)$$

where  $M$  is the size of the metadata - sum of inode table, header and bitmaps -  $\text{startblock}$  the start block of the extent where the offset is located and  $\text{offset\_mod}$  the local object offset that has been decreased by the overall size of the extents over that have been iterated.

**A3** As already mentioned in the second case object data like the real offset can only be extracted by traversing its extent list. When copying data from/to the object a similar approach is necessary. Starting at a specific offset located in the corresponding extent a number of bytes is supposed to be copied. This is done until the end of the extent is reached and the count is decreased by the number of bytes that have been copied (temporary variable `local_count`). If there are bytes left that have to be written the next extent has to be visited and the remaining bytes are copied into/from this extent. This is repeated until no bytes are left.

In Listing 4.7 the main implementation aspects of the copy operation are listed. This function is used for both read and write. That is why a parameter has to be set, specifying the so-called *copy mode*. In order to copy the data from one memory location into another, the function `memcpy` is used (see lines 11 and 20). This function specifies a destination buffer (`dest`) where the data is copied to and a source buffer (`src`) from where the data is taken. The third parameter specifies the amount of data to be copied. Depending on what copy mode has been chosen (`read_or_write`), the buffers `dest` and `src` are set according to the operations. For example if a write operation is executed, `dest` corresponds to the extent mapping and `src` to the data buffer.

The operation starts at the extent where the real offset was located (A2). In order to ensure that the entire amount of data (specified by `count`) will be copied, the operation is processed in a while-loop. This checks whether the bytes currently processed (`bytes_written`) already reached the desired total amount of bytes. Whenever a specific amount of data was written, the variable `bytes_written` is increased by this amount.

An additional variable is used to store the remaining amount of bytes that still has to be copied (`local_count`). This variable is initialized with the value of `count`. When this amount exceeds the available space of the current extent, only the amount between the end offset and the real offset is written (lines 6-20) and `local_count` is decreased by this amount (line 23). Then the next extent is fetched and `src` and `dest` are set according to the copy mode (lines 30 and 34). Since `local_count` is decreasing with every extent being traversed, an extent will be reached which end offset is greater than this count. Consequently, the remaining count will be copied (lines 39-41), and the procedure is finished.

*Listing 4.7: Copy data operation.*

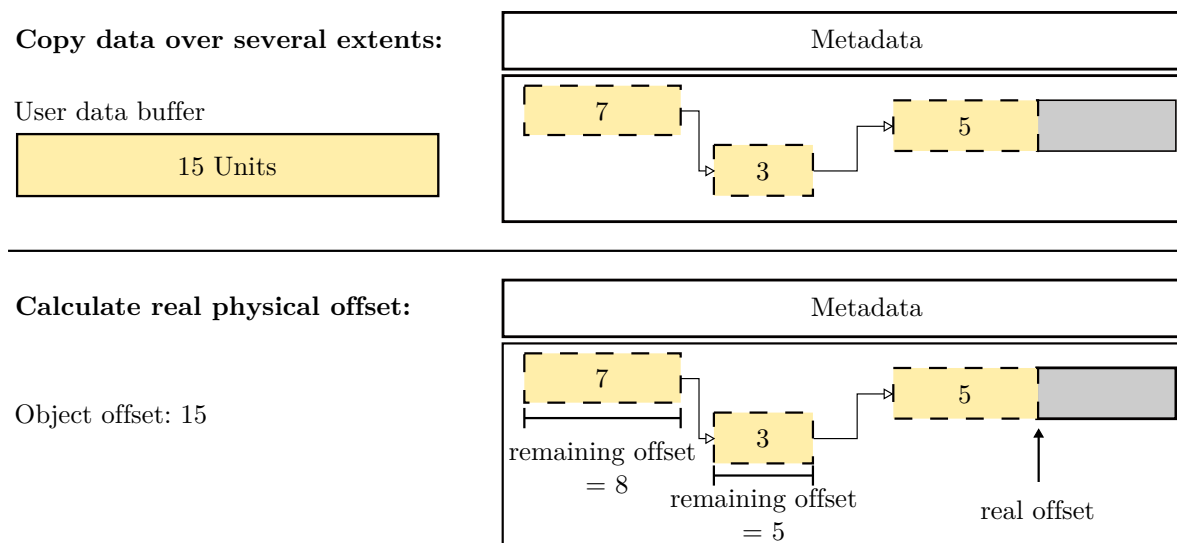
```

1  lobject_copy_data(Object* object, gpointer dest, gpointer src, ...,
    ↪ guint64 count, guint8 read_or_write, ...)
2  {
3      local_count = count;
4      while(bytes_written < count)
5      {
6          if(real_offset + local_count > end_offset)
7          {
8              amount = end_offset - real_offset;
9              if(read_or_write == WRITE)
10             {
11                 memcpy((gchar*)dest, (gchar*)src+bytes_written,
                    ↪ amount);
12
13                 if(LOBJECT_SYNC)
14                 {
15                     lobject_synchronize(object, dest, amount, ...);
16                 }
17             }
18             else
19             {
20                 memcpy((gchar*)dest+bytes_written, (gchar*)src,
                    ↪ amount);
21             }
22
23             bytes_written += amount;
24             local_count -= amount;
25
26             //get next extent
27             //real_offset is the start block address of next extent
28             if(read_or_write == WRITE)
29             {
30                 dest = lstore_get_mapping(objectstore, real_offset);
31             }
32             else
33             {
34                 src = lstore_get_mapping(objectstore, real_offset);
35             }
36         }
37         else
38         {
39             end_offset = real_offset + local_count;
40             //copy local_count, possibly synchronization
41             bytes_written += local_count;
42         }
43     }
44     //...
45 }

```

Figure 4.5 illustrates the traversal of the extent list when data is copied or the real offset is calculated. In this example a total amount of 15 data units is written and an object

offset of 15 is converted into the physical disk offset. The numbers depict the amount of data that has been traversed on the extent. This corresponds to either the object offset that will be converted to the real offset on the device or the amount of data that will be copied over the extent list. Since the first extent only has a length of 7, the next extent is examined and the object offset (or the amount of data that has still to be copied) decreases to 8. Iterating over the next extent results in the remaining local offset of 5 and the last amount to be copied to the extent, respectively. This can now be used to calculate the real offset.



**Figure 4.5.:** Extent traversal like it is executed when data is copied (top figure) or the real disk offset is calculated (bottom figure).

#### 4.5.2. Object and Inode Allocation

As mentioned in Section 3.2.1 an object is related with exactly one inode holding relevant meta information about the object. That is why an inode has to be allocated before the object can be used. As long as there are inodes available, a new object can be created. This is where the inode bitmap and the inode table come into play. Having stored the index of the next free inode in the header, the unused inode simply can be retrieved from the inode table. Then the desired inode can be retrieved by accessing it using the inode ID as an array index. In order that only one thread calls the inode allocation process, it is locked as long as one thread allocates an inode. This ensures that only one thread accesses the inode bitmap. In case multiple threads are sharing the same object, it has to be ensured that one thread creates it and broadcasts the object to the other threads. The corresponding bit is toggled having then a value of one indicating that this inode is in use now. The number of allocated inodes is incremented by one. If there are still inodes available after the incrementation, the bitmap is scanned for the next free inode



(the next bit of value zero). The corresponding index is stored in the header as the next available inode.

### 4.5.3. Object and Inode Deallocation

When deleting an object, the corresponding inode has to be made available again, so that it can be used for other objects. Since the deleted inode could have held extents, the blocks associated with these extents have to be freed. For this, the extents have to be traversed and the blocks allocated by these extents are freed simultaneously. The corresponding procedure toggles the bits for each extent back to bits of value zero. For this each extent information associated with the extents of the inode has to be extracted. The procedure of finding consecutive bits starts from the start index of the next and toggles the number of bits given by the number of blocks held by the extent. This is repeated until all extents have been traversed. The inode is set as unused in the inode bitmap by resetting the bit to zero. All the attributes of the inode are reset to default values. The inode bitmap is finally scanned for the next free inode.

The inode deallocation is locked for one thread. If several threads share the same object, only one thread is allowed to delete the object.

## 4.6. Transactions

Concerning transaction from the user's view, `ltransaction_start` and `ltransaction_execute` are the key functions (see Section 3.4.2). `ltransaction_start` is invoked by the user in order to indicate that the following operations should be transformed into one operation. In this operation the queue storing the pending requests is initialized. A request is a data structure holding the attributes relevant for read and write operations (Listing 4.8). Each object holds its own queue. In this implementation, transactions are realized for objects that are not shared among several threads.

**Listing 4.8:** A request saving the key attributes for write and read operations.

```

1 struct Request
2 {
3     guint64 size;
4     guint64 offset;
5     gpointer data;
6     LObjectSyncMode sync;
7     guint8 type_of_request;
8 };

```

When a transaction was invoked by the user, read and write operations are not immediately executed. For that, these operations have been adapted. The read and write operations act as wrapper functions. Listing 4.9 shows an excerpt of the main part of these operations. Instead of doing the actual copy operation (see Section 4.5.1), the requests are first collected and pushed to the queue (lines 5 and 6). Each request is initialized with the values of the current read or write call. If the queue was not initialized,

meaning that the start function was not called and a transaction is not desired, then the operation calls an internal function doing the actual work (line 15-18).

Due to memory constraints, the transaction may have to be executed before it was explicitly invoked by the user. That is why the queue is checked if it contains a specific number of elements (line 3). If this number (`length`) is exceeded then the transaction has to be executed immediately with the current requests in the queue. After doing that, the queue has to be re-initialized, otherwise the ordinary read and write operations are executed.

**Listing 4.9:** *Push read and write request into a queue.*

```
1  if(object->my_queue!=NULL)
2  {
3      if(g_queue_get_length(object->my_queue) < length)
4      {
5          new_request = ltransaction_create_request(offset, count,
6              ↪ buffer, REQUEST_TYPE, mode);
7          ltransaction_push_request(new_request, object->my_queue);
8      }
9      else
10     {
11         ltransaction_execute(object,&local_error);
12         //if local_error != NULL -> error handling
13         ltransaction_start(object);
14     }
15 }
16 else
17 {
18     //do ordinary writing or reading
```

Listing 4.10 shows the merging process of both read and write operations in `ltransaction_execute`. As long as the queue is not empty, a new main request is created and the merging process is executed. The start element for the main request is simply the head element of the queue. Read and write operations are handled differently in this process. Write requests are always removed from the queue if they fulfill the conditions and are merged with the main request, while read requests are kept in the queue until the actual read operation is executed.

In order to check if the conditions for merging requests (see Section 3.4.2) are met for a write request in the queue (line 5), it is first “peeked”, meaning that it is not removed from the queue (lines 4 and 12). Only when the request currently checked fulfills the conditions, it is merged with the main request and then it is removed from the queue. Then, the next head element of the queue is examined. Concerning the read requests, a counter is incremented indicating to which element the conditions are met and which element should be examined in the next iteration (lines 16 and 17).

**Listing 4.10:** *Checking if requests can be merged.*

```
1  req = g_queue_pop_head(queue);
2  ltransaction_create_main_request(object,main_req,req,&local_error);
```

```

3 //...
4 req_tmp = g_queue_peek_head(queue);
5 while(conditions)
6 {
7     ltransaction_merge_requests(main_req, req_tmp);
8     if(main_req->type_of_request == WRITEREQ)
9     {
10         req = g_queue_pop_head(queue);
11         g_free(req);
12         req_tmp = g_queue_peek_head(queue);
13     }
14     else
15     {
16         req_tmp = g_queue_peek_nth(queue, i);
17         i++;
18     }
19 }

```

If a request in a queue does not meet the conditions, the actual I/O operations are executed. Concerning the write operation, the values stored in the main request data structure are taken as parameters for the internal write function.

The execution of the read operation is depicted in Listing 4.11. Read requests are not merged into one operation, since the read buffers provided by the original read requests should contain the readout data at the end. If a request is found out to be a read operation, it and all the following possible read requests are kept in the queue. By checking the merge conditions it can be detected whether the read requests are contiguous. The main request is used here for storing the size of this contiguous region. If the read requests are contiguous, it is reasonable to read ahead the given pages. This is achieved by using the function call `madvise` providing several flags to invoke a special handling of address ranges (line 4). Here, `MADV_SEQUENTIAL` is used. It expects pages to be in a sequential order and therefore performs an aggressive read-ahead of pages. Now the requests can be gradually removed from the queue while executing the read process (lines 8 and 9).

**Listing 4.11:** *processing the read requests*

```

1 lobject_get_real_offset(object, main_req->offset, &real_offset,
   ↪ NULL);
2 mapping = lstore_get_mapping(objectstore, real_offset);
3 mapping = lutils_get_address_at_page_boundary(mapping, &rest);
4 madvise(mapping, main_req->size+rest, MADV_SEQUENTIAL)
5 //...
6 while(i<index)
7 {
8     req_tmp = g_queue_pop_head(queue);
9     bytes +=
   ↪ lobject_read_internal(object, req_tmp->offset, req_tmp->size,
10         req_tmp->data, &local_error);
11     if(local_error)
12     {
13         g_propagate_error(error, local_error);

```

```
14     }
15     g_free(req_tmp);
16     i++;
17 }
```

## Chapter Summary

This chapter describes implementation fundamentals and highlights. The object store is implemented in an object-oriented way. Each data structure designed in the previous chapter constitutes a programming unit and is represented by an opaque data structure and a set of operations. The internal workings of the API operations are implemented. Bitmap algorithms and the extent allocation procedure provide the basis for the functionality of the object and object store operations.

# Chapter 5.

## Evaluation

The focus of the previous two chapters was the design and technical realization of the object store. Another important issue is the performance evaluation.

The performance is measured with benchmark programs and can be evaluated in two categories. In the first category, the performance of read and write operations is evaluated. Several configurations are possible for the read and write operations. The results for configurations that are of special interest are presented in this chapter. In addition to this, the performance of LEXOS is compared with the performance of common file systems. The second important category is metadata performance. This includes the often-used operations for creating, deleting, opening objects and querying the current status of the object.

All benchmarks used in this evaluation were conducted on a personal computer with the following technical configuration: 8 GB of random access memory, an Intel Core i7 CPU (2.8 GHz) and an HDD with a storage capacity of 500 GB. The write speed of the HDD is nearly 128 MB/s, while its read performance is approximately 131 MB/s.

The chapter is divided in two parts: the evaluation of I/O and metadata performance of the object store. The I/O performance is presented first by giving an overview to the benchmark and its possible configurations. Subsequently the benchmark results are discussed. The second part evaluates the metadata performance.

### 5.1. Small-Access Benchmark

The purpose of the so-called small-access benchmark is to measure the read and write speed of the object store. The benchmark processes single data blocks in iterations (small access). In order to do this, a total amount of data has to be specified that is written or read, respectively. This amount of data is specified by a total number of blocks with a specific block size. In addition to do this, various options can be set. These are:

**Access Pattern** Sequential and randomized access pattern are possible.

**Object Sharing** Each thread can hold its own object or several threads share the same object.

**Synchronization** A modified data block can be flushed immediately to disk after writing it.

The benchmark contains two loops, where it first writes data to an object and then, in the second loop, reads it back from the object. The main loop is shown as pseudocode in Listing 5.1. The loop iterates over the number of blocks that has been specified for the object which is either shared or private. In case of the read operation, no synchronization of the data is necessary. Depending on what access pattern has been chosen (sequential or random), the offset is set appropriately.

**Listing 5.1:** *Main loop of the small-access benchmark.*

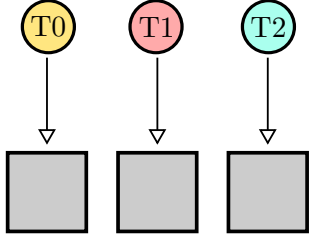
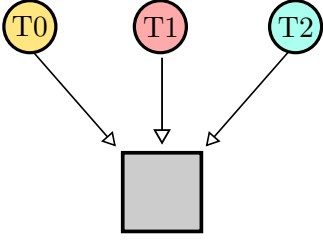
```

1  for(i=0; i<number_of_blocks; i++)
2      write(object=private/shared, offset=sequential/random,
           ↪ synchronization=on/off)
3      OR
4      read(object=private/shared, offset=sequential/random)
5  endfor

```

In a sequential access pattern, blocks are written and read in an ascending order. When a randomized access pattern is chosen, the blocks are processed in an arbitrary order. These two patterns can be combined with private and shared objects. In each case the offset is calculated differently. This will be explained in the following:

The first two options are illustrated in Figure 5.1. This figure shows an example benchmark configuration with three threads (named by the thread IDs T0, T1 and T2) and nine blocks (b0 - b8) that are processed. These nine blocks are equally divided among the threads, so that each thread has to process three blocks. The options “private object” and “shared object” are illustrated in separate columns. For each option, a sequential or random access pattern can be applied; these patterns are depicted in separate rows. The figure shows the logical view of the objects, that is how the object is viewed as a whole. In contrast to this view, in the physical view the object is distributed over the disk. The left column shows the case of private objects. Since three threads are involved in the benchmark and each thread has its own object, three objects are processed. In a sequential access pattern the iteration number refers to the block where data is written. For example in iteration 0 data is written to block b0, in iteration 1 it is written to block b1 and so forth. The offset of the block is a multiple of the block size and the iteration number. For a randomized access pattern, a number is chosen arbitrarily from an interval ranging from zero to the number of blocks each thread has to process. That number is further used for the calculation of the block offset where data is written or read. In Figure 5.1 an exemplary random permutation of the blocks is given for each thread. The random permutation is illustrated as a 3-tuple containing numbers that have been picked randomly in each iteration. An element of the tuple refers to the randomized number selected in an iteration, e.g., the first element of the tuple is the randomly chosen number in iteration 0. In case of thread T0 in iteration  $i = 0$  data is written to block b1 first. In the right column, it is shown how both access patterns are applied on a shared object. Since all threads are working on one object, the offset has to be calculated in this way that not more than one thread process the same block of the object. This is achieved by applying the offset calculation for shared objects given in Figure 5.1 (right column, second row). In this way, the threads alternate between the block numbers. For example

	Private Object	Shared Object
		
Sequential Access	$\text{blocksize} * i$ <div style="display: flex; align-items: center;"> <div style="margin-right: 10px;">T0:</div> <div style="border: 1px solid black; padding: 2px;"> <div style="background-color: yellow; padding: 2px;">b0 i=0</div> <div style="background-color: red; padding: 2px;">b1 i=1</div> <div style="background-color: cyan; padding: 2px;">b2 i=2</div> </div> </div> <div style="display: flex; align-items: center; margin-top: 5px;"> <div style="margin-right: 10px;">T1:</div> <div style="border: 1px solid black; padding: 2px;"> <div style="background-color: red; padding: 2px;">b0 i=0</div> <div style="background-color: red; padding: 2px;">b1 i=1</div> <div style="background-color: red; padding: 2px;">b2 i=2</div> </div> </div> <div style="display: flex; align-items: center; margin-top: 5px;"> <div style="margin-right: 10px;">T2:</div> <div style="border: 1px solid black; padding: 2px;"> <div style="background-color: cyan; padding: 2px;">b0 i=0</div> <div style="background-color: cyan; padding: 2px;">b1 i=1</div> <div style="background-color: cyan; padding: 2px;">b2 i=2</div> </div> </div>	$\text{blocksize} * (\#threads * i + \text{thread\_id})$ <div style="display: flex; align-items: center;"> <div style="margin-right: 10px;">b0 b1 b2 b3 b4 b5 b6 b7 b8</div> <div style="border: 1px solid black; padding: 2px;"> <div style="background-color: yellow; padding: 2px;">i=0</div> <div style="background-color: red; padding: 2px;">i=0</div> <div style="background-color: cyan; padding: 2px;">i=0</div> <div style="background-color: yellow; padding: 2px;">i=1</div> <div style="background-color: red; padding: 2px;">i=1</div> <div style="background-color: cyan; padding: 2px;">i=1</div> <div style="background-color: yellow; padding: 2px;">i=2</div> <div style="background-color: red; padding: 2px;">i=2</div> <div style="background-color: cyan; padding: 2px;">i=2</div> </div> </div> <div style="display: flex; align-items: center; margin-top: 5px;"> <div style="margin-right: 10px;">T0 T1 T2 T0 T1 T2 T0 T1 T2</div> </div>
Random Access	<div style="display: flex; align-items: center; margin-bottom: 5px;"> <div style="margin-right: 10px;">Random permutation:</div> <div style="margin-right: 20px;">T0: (1,2,0)</div> <div style="margin-right: 20px;">T1: (2,0,1)</div> <div>T2: (0,2,1)</div> </div> $\text{blocksize} * i$ <div style="display: flex; align-items: center;"> <div style="margin-right: 10px;">T0:</div> <div style="border: 1px solid black; padding: 2px;"> <div style="background-color: yellow; padding: 2px;">b2 i=2</div> <div style="background-color: red; padding: 2px;">b0 i=0</div> <div style="background-color: yellow; padding: 2px;">b1 i=1</div> </div> </div> <div style="display: flex; align-items: center; margin-top: 5px;"> <div style="margin-right: 10px;">T1:</div> <div style="border: 1px solid black; padding: 2px;"> <div style="background-color: red; padding: 2px;">b1 i=1</div> <div style="background-color: red; padding: 2px;">b2 i=2</div> <div style="background-color: red; padding: 2px;">b0 i=0</div> </div> </div> <div style="display: flex; align-items: center; margin-top: 5px;"> <div style="margin-right: 10px;">T2:</div> <div style="border: 1px solid black; padding: 2px;"> <div style="background-color: cyan; padding: 2px;">b0 i=0</div> <div style="background-color: cyan; padding: 2px;">b2 i=2</div> <div style="background-color: cyan; padding: 2px;">b1 i=1</div> </div> </div>	$\text{blocksize} * (\#threads * i + \text{thread\_id})$ <div style="display: flex; align-items: center;"> <div style="margin-right: 10px;">b0 b1 b2 b3 b4 b5 b6 b7 b8</div> <div style="border: 1px solid black; padding: 2px;"> <div style="background-color: yellow; padding: 2px;">i=2</div> <div style="background-color: red; padding: 2px;">i=1</div> <div style="background-color: cyan; padding: 2px;">i=0</div> <div style="background-color: yellow; padding: 2px;">i=0</div> <div style="background-color: red; padding: 2px;">i=2</div> <div style="background-color: cyan; padding: 2px;">i=2</div> <div style="background-color: yellow; padding: 2px;">i=1</div> <div style="background-color: red; padding: 2px;">i=0</div> <div style="background-color: cyan; padding: 2px;">i=1</div> </div> </div> <div style="display: flex; align-items: center; margin-top: 5px;"> <div style="margin-right: 10px;">T0 T1 T2 T0 T1 T2 T0 T1 T2</div> </div>

**Figure 5.1.:** Overview of the possible configurations of the small-access benchmark.

in iteration 0, thread T0 processes block b0, T1 processes block b1 and T2 processes block b2 and so forth. The next block T0 can process is block b3. The same calculation is used for a randomized access pattern, but like for private objects, with randomly chosen numbers. In this example the same random permutation as for private objects is set.

The small-access benchmark was used for the following tests:

**Block Size Test** LEXOS was executed with several block sizes using a fixed total amount of data with both access patterns. The resulting performance using these different block sizes was investigated. Based on this results, a block size was chosen for the following benchmarks.

**LEXOS Options Test** This investigates the different options that can be set in the benchmark (access pattern, object sharing and synchronization). Moreover, the performance benefits of using transactions are presented.

**Performance Comparison** Three file systems have been chosen that are interesting for comparison with LEXOS. If possible, the same options are tested as for the LEXOS options test described above. Transactions can not be compared, since the other file systems do not support a similar feature. Write and read performance are evaluated separately.

In order to obtain meaningful results, the benchmark tests were repeated at least three times. For all following plots of the benchmark tests, the resulting standard deviation is depicted as an error bar for each data point. If possible, the cache of the tested file system was dropped after each run in order to avoid that data is read directly from the cache. The results of each test now will be presented in separate sections.

### 5.1.1. Block Size Test

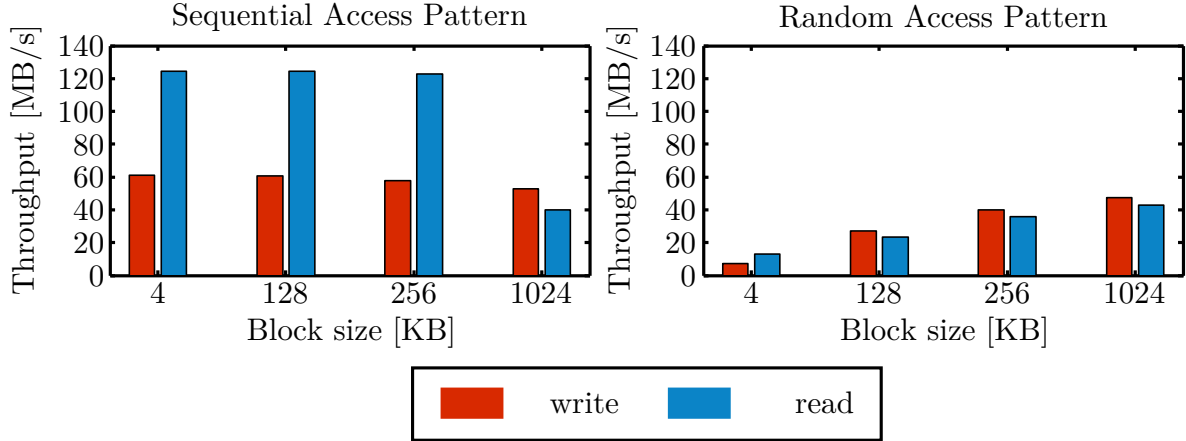
The purpose of this benchmark is to investigate how LEXOS handles different block sizes. LEXOS was tested with several different block sizes both, in a sequential and a randomized access pattern. Besides that the most basic options are used. During this benchmark data is not synchronized after writing it and no transactions are used. Moreover the benchmark is executed with one object and one single thread. The block sizes that have been tested range from 4 KB to 1 MB. For each test an object store was created holding blocks with the size to be tested. The processed amount of data that was fixed at 32 GB for a sequential access pattern and at 500 MB for a random access pattern.

The left plot in Figure 5.2 shows the resulting throughput for a sequential access pattern. For each block size the read speed is nearly the same. The write speed reaches a maximum throughput of nearly 60 MB/s with a block size of 4 KB. With an increasing block size the throughput slightly decreases. It can be seen that the read performance decreases significantly when a block size of 1 MB is used. It has to be investigated with further profiling tools which functions are possible performance bottlenecks.

For the following benchmark tests using a sequential access pattern, a block size of 4 KB was chosen due to its maximal performance.

The right plot in Figure 5.2 shows the results concerning the randomized access pattern measured with the same block sizes as they are used with the sequential access pattern. In contrast to the sequential I/O pattern the write-speed increases when a bigger block size is used. The same holds for the read speed. When the block size is greater than 4 KB the read speed is slightly smaller than the write speed. In order to investigate how LEXOS and the other file systems can handle data with a small block size, a block size of 4 KB was chosen for the following benchmarks using a randomized access pattern. The influence of a randomized access pattern is maximal on the performance for small block sizes. Thus it is most interesting to investigate small block sizes. Notwithstanding this, such small accesses are commonly applied, e.g., in earth system simulations [SKH<sup>+</sup>12].





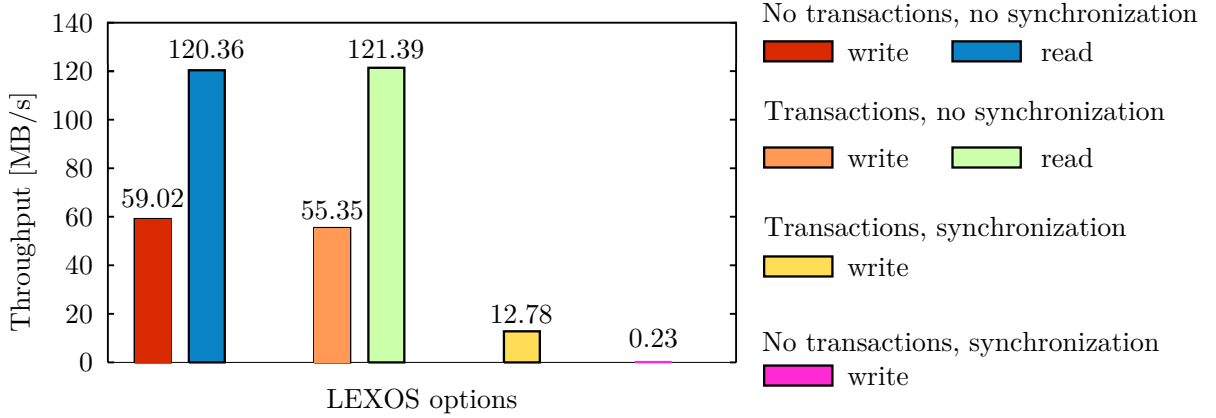
**Figure 5.2.:** Block size test for a sequential and random access pattern (one object and one thread).

### 5.1.2. LEXOS Options Test

In this section the performance of the different options of LEXOS is investigated. The API of LEXOS allows to batch multiple operations in transactions. This test investigated the performance benefits of using transactions and compares it with the read and write speed of tests without transactions. Moreover, the synchronization configuration is tested. For the tests without synchronization a total amount of data of 32 GB is set. As synchronization is a time-consuming operation, the processed amount of data is reduced to 1 GB. The test is executed with one thread and a sequential access pattern.

Figure 5.3 shows the resulting read-write speed of the four test configurations. As can be seen, transactions do not affect the read and write performance when no synchronization is applied. Using transactions results in a slightly slower write performance than using no transactions. This can be explained by the additional management overhead that is related with transactions, for example keeping requests in a queue, the request merging process etc. By applying additional optimizations for sequential access patterns, the performance could be further improved. For a randomized access pattern, transactions do not benefit the performance. Due to this randomized data processing there can not be find any consecutive data blocks that could be processed in one operation. That is why transactions are not tested with this pattern.

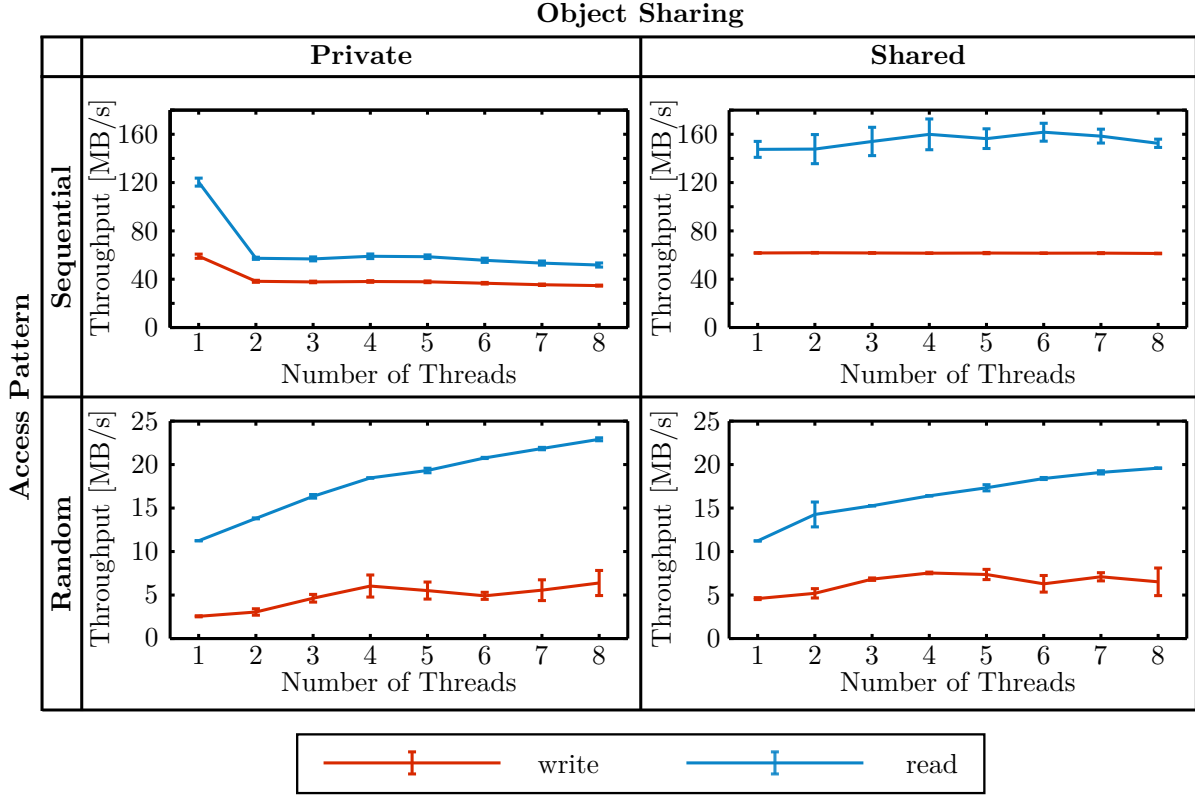
As already mentioned, synchronizations are very time-consuming operations. Consequently the write performance is very low when synchronization is used without transactions (0.23 MB/s). Normally, when no synchronization is used, data is first collected by the kernel in order to merge the data and process it as one operation. But in case of synchronization, it has to be ensured that data is immediately flushed to the storage device. This is very inefficient when single small data blocks (in this case 4 KB) are written. As can be seen in Figure 5.3, transactions perform very well in combination with synchronization: The write speed is increased by more than a factor of 50. In the current implementation, a transaction is allowed to merge data up to 1 MB. In



**Figure 5.3.:** Comparison of different options of LEXOS (one object, sequential access pattern, one thread).

this test, a sequential access pattern was used, that is why data can always be merged in 1 MB chunks of data. So, instead of synchronizing blocks of 4 KB, data chunks of 1 MB are synchronized. Synchronization of data is important and necessary in many applications. For example in database applications synchronization plays a key role concerning the data consistency. Another prominent example are earth system simulations which results are required to be stored permanently during the simulation execution, so that in case of a system failure not all of the results are lost. An example model was investigated (General Estuarine Transport Model - GETM [BBU13]) and it was found that its performance does suffer from a permanent synchronization of small amounts of data [SKH<sup>+</sup>12]. That is why transactions are considered as very advantageous in this case.

The previous benchmark test is focused on the evaluation of the transaction and synchronization feature implemented in LEXOS. This was tested with one object and one thread. Another interesting part of the evaluation is to investigate the read-write performance of LEXOS by using several threads and with different benchmark configurations: The options access pattern and object sharing can be combined. These can in turn be combined with the synchronization option. This results in eight test cases. Results with and without synchronization will be discussed separately. The test cases are sorted by the access pattern and object sharing option in a  $2 \times 2$  matrix, where the columns refer to the object sharing option (private object, shared object) and the row to the access pattern (sequential, random). The following two figures (Figure 5.4 and Figure 5.5) plot the throughput depending on the number of threads. Up to eight threads were used. Figure 5.4 shows the resulting read-write performance of the four test cases without synchronization. In all test cases, the read speed is greater than the write performance. During the write operation it becomes necessary to allocate new extents for the objects, so that there is a management overhead reducing the performance. This overhead of extent allocation does not occur during the read operation, which is why the read performance is noticeably higher than the write performance.



**Figure 5.4.:** Comparison of different benchmark configurations with several threads without synchronization.

The sequential access pattern has a higher performance, both in read and write, than the randomized access pattern. In a sequential access pattern fewer disk seeking operations are necessary, because data is processed contiguously. Concerning the sequential access pattern, different performance results can be identified with private and shared objects. When using a shared object, the write performance is constant, whilst when using private objects the write performance decreases when more than one thread is used. Concerning the shared object case, the read performance is slightly greater than the measured read performance of the storage device. The read performance of LEXOS with private objects has the same curve shape like the write performance. The reasons for these results could be the following:

**Private Objects** The more objects are created, the more extents from different objects are stored on the disk. The threads compete for disk space and they are possibly not able to store the extents for their objects consecutively on the disk. That is why more seeking operations are necessary when reading from and writing to an object with several threads involved. It should be noted that extents are always allocated with the maximum extent size in order to avoid a high order of object fragmentation when several threads and objects are involved. If this is not used, the extent list of one object could contain a lot of extents. Consequently more seeking operations would be necessary and the performance would additionally suffer. In

addition to this, the extent allocation is locked for only one thread at a time in order to avoid conflicts during the allocation, reducing the write performance in consequence.

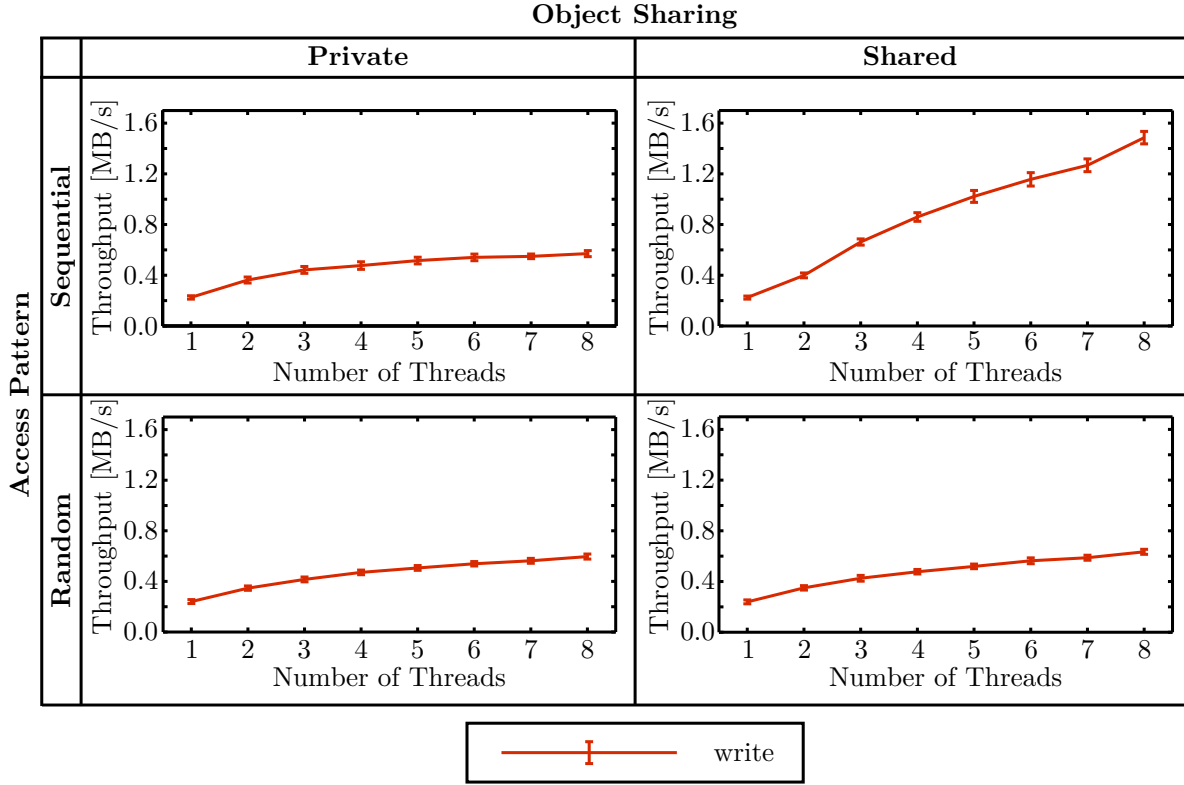
**Shared Objects** In contrast to private objects, the extents can be stored consecutively on the disk, although there are several threads involved. This results in fewer disk seeking and consequently in a higher throughput than with private objects.

Normally, the read throughput of a file system can reach the read performance of the storage device at maximum. The measured read performance of the disk used for the benchmark was measured at nearly 131 MB/s. LEXOS reaches nearly 160 MB/s. The function call `mmap` used for memory mapping possibly applies a caching technique allowing to reach this performance. Moreover, `mmap` uses read-ahead by default. Since all extents are consecutive on the disk and one object is tested, the correct data related with this object is read. It seems that the total amount of data (32 GB) is not enough to avoid data to be read directly from the cache. Normally, data should be always read from the disk, since the main memory has a capacity of 8 GB. Only this amount can be held in the main memory at maximum.

The read-write speed curves concerning the randomized access pattern differ greatly from the read-write speed curves of the sequential access pattern. For shared and private objects, the read performance noticeably increases with an increasing number of threads. The write performance increases slightly, but it shows more variations in the curve shape than the read performance, which is more stable. The write performance of private and shared objects is nearly the same with a randomized access pattern. In contrast to the sequential access pattern, the read performance of private objects is slightly greater than for shared objects. Several reasons could explain the behavior of the read-write performance for a randomized access pattern:

- `mmap` may be well-suited for randomized access. `mmap` may recognize random access and use appropriate optimizations for this pattern.
- The total amount of data is smaller than it was used in a sequential access pattern. It was expected that writing and reading a randomized access pattern is more time-consuming than processing a sequential access pattern. Consequently fewer extents are necessary to be allocated for one object than in the test case with a sequential access pattern. This may enhance the scalability concerning the read operation, even though the data is read randomly.

Figure 5.5 shows the resulting write performance when using synchronization with the four benchmark configurations. The read performance is not shown, since data has not to be synchronized when it is read and therefore has no effect on the read performance. Three of the four test cases shown in Figure 5.5 show nearly the same results. It is noticeable that only the write performance with a shared object and a sequential access pattern differs from the results of the other test cases by showing a remarkable speedup. The same reason as for shared objects and a sequential access pattern without



**Figure 5.5.:** Performance evaluation of LEXOS by using different benchmark configurations with several threads with synchronization.

synchronization holds for this result: Since all extents are consecutive and can be written in an ascending order, this case promises a higher performance than the other test cases. In the other three test cases, the write performance seems to converge to a specific value, indicating that there may be a bottleneck as more threads are executing I/O operations on objects. Concerning private objects this could again be explained by the distribution of the extents of the different objects leading to a lot of seeking operations. In addition to this, the synchronization operation considerably reduces the write performance.

As it was observed in the previous results without synchronization, a randomized access pattern significantly reduces the write speed with regard to the sequential access pattern. In combination with synchronization, the throughput is reduced even more.

### 5.1.3. Performance Comparison with File Systems

In order to evaluate the quality of LEXOS, its read and write performance were compared with the performance of three file systems. As mentioned in Section 3.5 there does not exist any object store that would be comparable with LEXOS. For this reason, three important and well-known file systems have been chosen that can be compared with LEXOS to some degree:

**ext4** The de-facto standard file system in Linux-based operating systems. As mentioned

in Chapter 2 it also uses the extent mapping scheme. The ext file systems are developed since 1991.

**ZFS** Uses an object store for storage management and is therefore of special interest concerning the performance comparison with LEXOS. It was developed for Solaris [Ora12] operating systems (since 2001) and is well-optimized for these systems. ZFS is also available for Linux, but this version is less optimized. Since the object store is developed for Linux operating systems and the benchmarks were also executed on a Linux system, the corresponding version of ZFS was used for comparison.

**btrfs** btrfs [RBM13] also uses an object store and the extent mapping scheme. btrfs is developed since 2007 and constitutes the most recent file system compared to ZFS and ext4. That is why it may be less optimized than the other two file systems.

From now on, the term *object* also refers to the term *file* unless it is not mentioned explicitly. The next sections separate the evaluation of read and write performance. The diagrams are again organized in a  $2 \times 2$  matrix as it was shown in Section 5.1.2. In all test cases and for all file systems, a block size of 4 KB was used.

### Write Performance without Synchronization

Figure 5.6 shows the resulting write performance of each file system and LEXOS without synchronization with respect to the number of threads. When the sequential access pattern was tested, a total amount of 32 GB was used, while the amount of data for a randomized access pattern was set to 1 GB for the same reasons as in Section 5.1.2. The results are similar for the two object sharing options, whereas the results differ for the two access patterns. That is why, the results of the patterns will be described separately in the following:

**Sequential Access Pattern** The write performance of LEXOS is lower than the performance of the other file systems, both for private and shared objects. With one thread it nearly has the same performance as ZFS. btrfs and ZFS have a constant throughput with an increasing number of threads. ext4 shows a greatly varying performance curve. btrfs has the highest performance among all file systems and LEXOS. btrfs is a copy-on-write file system and has to perform copy-on-write permanently. With this technique it is more challenging to achieve a reasonable performance. Nevertheless it shows a higher performance than ext4 which has been developed and optimized over years. btrfs nearly reaches the write performance of the underlying storage device. It also uses the extent mapping scheme, but in contrast to LEXOS it uses better optimized data structures and allocation algorithms in order to manage the extents [RBM13]. This suggests that further improvements of LEXOS could increase its performance to a similar level.

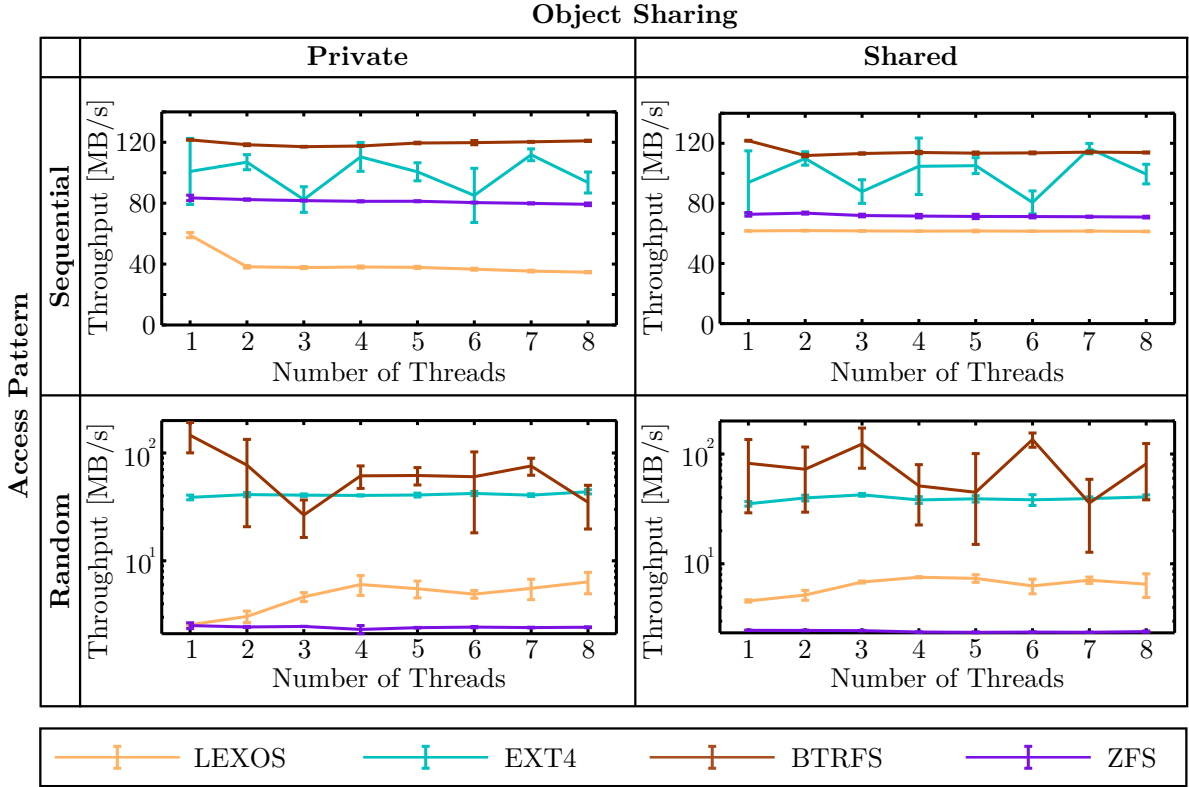
Concerning the test with one shared object, LEXOS nearly has the same throughput as ZFS. Both show the same constant curve shape with an increasing number

of threads. ZFS uses the Solaris Porting Layer (SPL), a Linux kernel module providing Solaris kernel APIs, in order to work within a Linux operating system. ZFS was mainly designed for Solaris and therefore uses its interfaces [Daw07]. The SPL maps these interface to the Linux interface. This adds a certain amount of management overhead and may impact the performance of ZFS. ext4 and btrfs directly use the Linux interfaces.

btrfs has again the highest throughput. It is noticeable that it shows a similar curve shape like LEXOS does in case of private objects. In the private object case, for LEXOS a decrease of the throughput was observed when more than one thread is used, whilst in the shared object case the throughput stays constant. For btrfs the relation inverses concerning the object sharing option and a sequential access pattern: With a shared object, btrfs has its maximum throughput with one thread, while the throughput then decreases with two threads. With private objects the write speed is constant. Concerning btrfs, completely different reasons could explain this behavior. Since the threads share one file, extensive locking may be necessary when multiple threads want to write to this file. In case of private objects, no file locking may be necessary. This could explain the slightly better performance.

**Randomized Access Pattern** The diagrams concerning the randomized access pattern plot the throughput on a logarithmic scale. In this pattern, LEXOS has a considerably higher performance than ZFS. ZFS has to apply copy-on-write permanently. Due to the randomized access pattern, no further optimizations can be applied. There is a low chance that several data blocks can be merged in larger chunks, so that this chunk can be processed in one operation. For each data block, the expensive copy-on-write operation is necessary. This could be a reason for the performance impact.

Compared with the performance results of the file systems, LEXOS performance even shows a speedup when several threads and private objects are used. In contrast to the sequential access pattern, btrfs now shows a greatly varying performance curve shape and the performance of ext4 is constant and stable with an increasing number of threads. Probably, btrfs is not yet optimized for random access patterns, especially in combination with the copy-on-write technique.



**Figure 5.6.:** Comparison of the write performance without synchronization.

### Write Performance with Synchronization

Figure 5.7 summarizes the write performance of each file system and LEXOS concerning the four test cases with synchronization. In all cases a total amount of data of 250 MB to 500 MB was used.

In all test cases, LEXOS outperforms the file systems ext4 and btrfs. It should, however, be noted that LEXOS converges to a specific value and may not increase with regard to the number of threads in most cases. Only in the case of shared objects and a sequential access pattern, LEXOS achieves a remarkable speedup compared to btrfs and ext4. btrfs shows a small speedup in its performance in all test cases, ext4 only for private objects. As can be observed for the test case with shared objects and sequential I/O, LEXOS has essentially the same performance as ZFS.

ZFS has the highest throughput among the other file systems. It uses a so-called *intent log*, a fixed-size pre-allocated part of the disk where the incoming data is first stored before it is synchronized on the disk. In the background, the data is migrated later to the real locations on the disk. The results may indicate that ZFS first writes the data in a sequential order in this intent log, leading to fewer seeking operations and consequently a higher throughput than the other file systems.



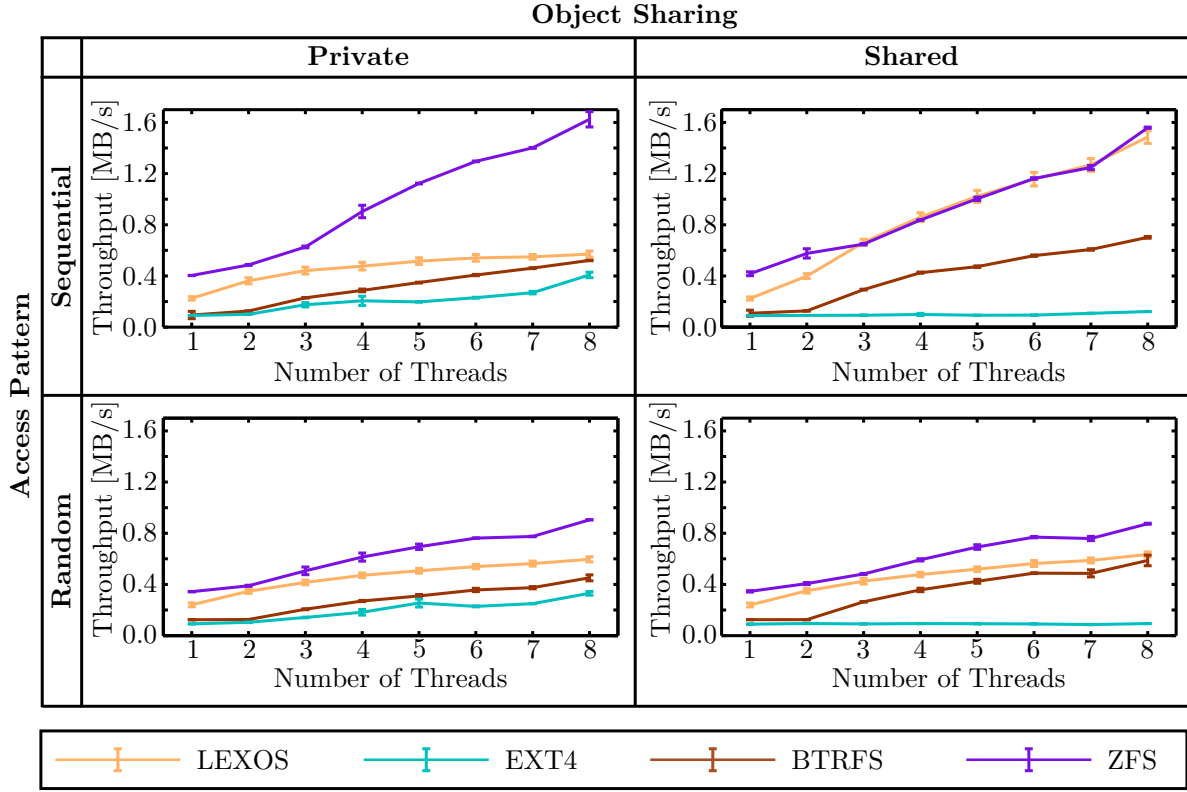


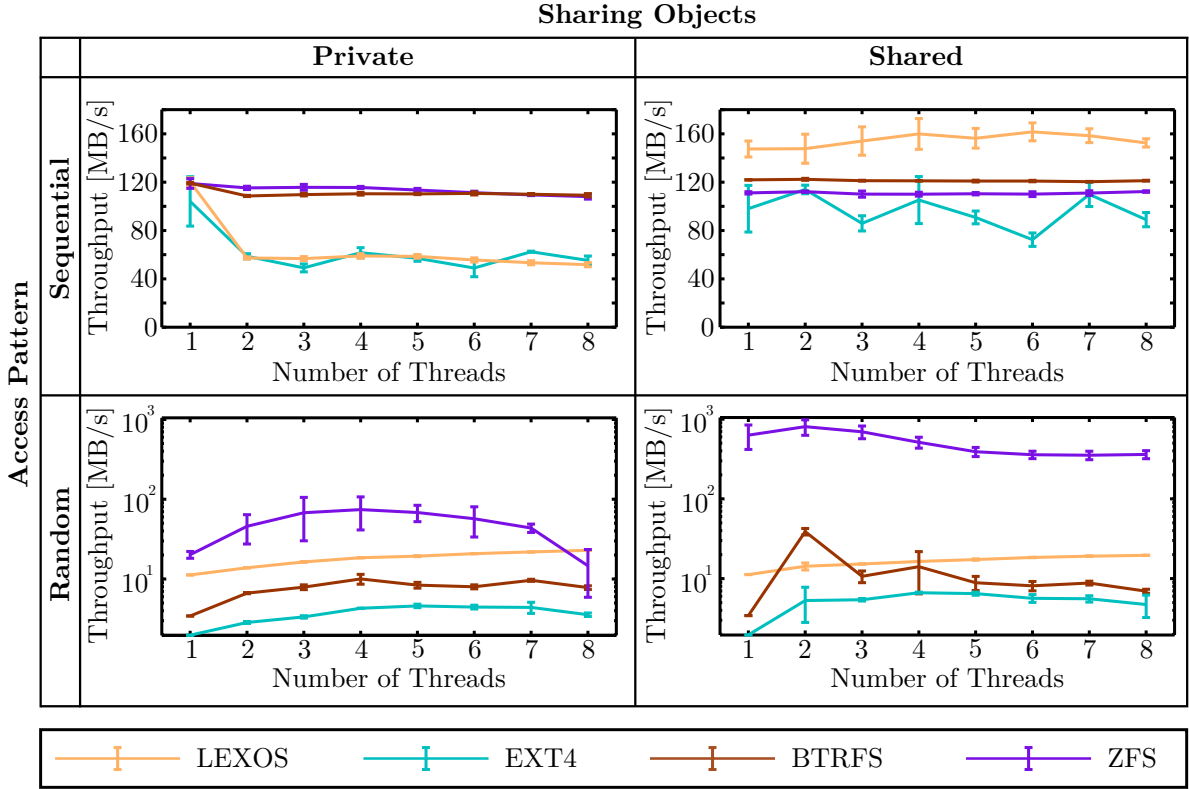
Figure 5.7.: Comparison of the write performance with synchronization.

## Read Performance

The results of read performance of each file system and LEXOS are presented in Figure 5.8 for the four test cases. The same amount of data as for the write performance test without synchronization was used.

The results of the read performance of each file system and LEXOS vary in dependence of what access pattern is used. The results of the read performance are regarded separately with respect to the access pattern:

**Sequential Access Pattern** Regarding the private object test case, the results of the throughput of LEXOS agree with the results of ext4. The performance curve shape of both are nearly the same, but ext4 shows more variances in its results. Again, as it was mentioned in the analysis of the write performance, this behavior of ext4 is very unusual as it is regarded as a well-optimized file system. Further experiments were not conducted in order to explain this, since this is beyond the scope of this thesis. When using one thread, LEXOS shows the same disk read performance as btrfs and ZFS. With more than one thread, the performance of LEXOS decreases as it was already observed in Section 5.1.2. As it was suggested in the analysis of the write performance, a similar performance as of btrfs could be reached if the extent data structures implemented by LEXOS were further optimized.



**Figure 5.8.:** Comparison of the read performance.

Concerning the shared object test case, LEXOS outperforms the other file systems. LEXOS provides an excellent performance compared to the other file systems. As it was mentioned in the LEXOS options test (see Section 5.1.2), some parts of the data may be read from the cache. ZFS and btrfs nearly reach the disk performance and have a constant performance with an increasing number of threads. ext4 shows more variations in its performance as it is the case with private objects.

**Randomized Access Pattern** For a random access pattern it was necessary to use a smaller amount of data than it was used with the sequential access pattern. This amount could easily be held in the file system cache, once it was retrieved from the disk. In order to obtain meaningful results, the cache was cleared after the data was written from the disk.

Compared to the file systems, LEXOS shows an increase of its read performance with an increasing number of threads both with private and shared objects. In case of private objects, LEXOS has a higher throughput than btrfs and ext4. ZFS shows a higher read performance than the other file systems. The reason that it uses its own cache implementation, namely the adaptive replacement cache [MM03]. That is why clearing the Linux page and buffer cache has no effects on the ZFS cache. No convenient way was found to clear the cache.

## 5.2. Metadata Performance

In the second benchmark, the metadata performance of the object store and the file systems is investigated. This means that the performance of the frequently-used operations `create`, `delete`, `open` and `status` is measured. The aim of this benchmark is to make sure that the object store indeed provides a good metadata performance compared to traditional file systems as this was motivated in the introduction. Moreover the results should justify why object stores should rather be used as a low-level abstraction of storage instead of full-featured file systems. In order to do this, the metadata performance of LEXOS is compared with the performance of the three file systems used for comparison in the previous benchmark tests.

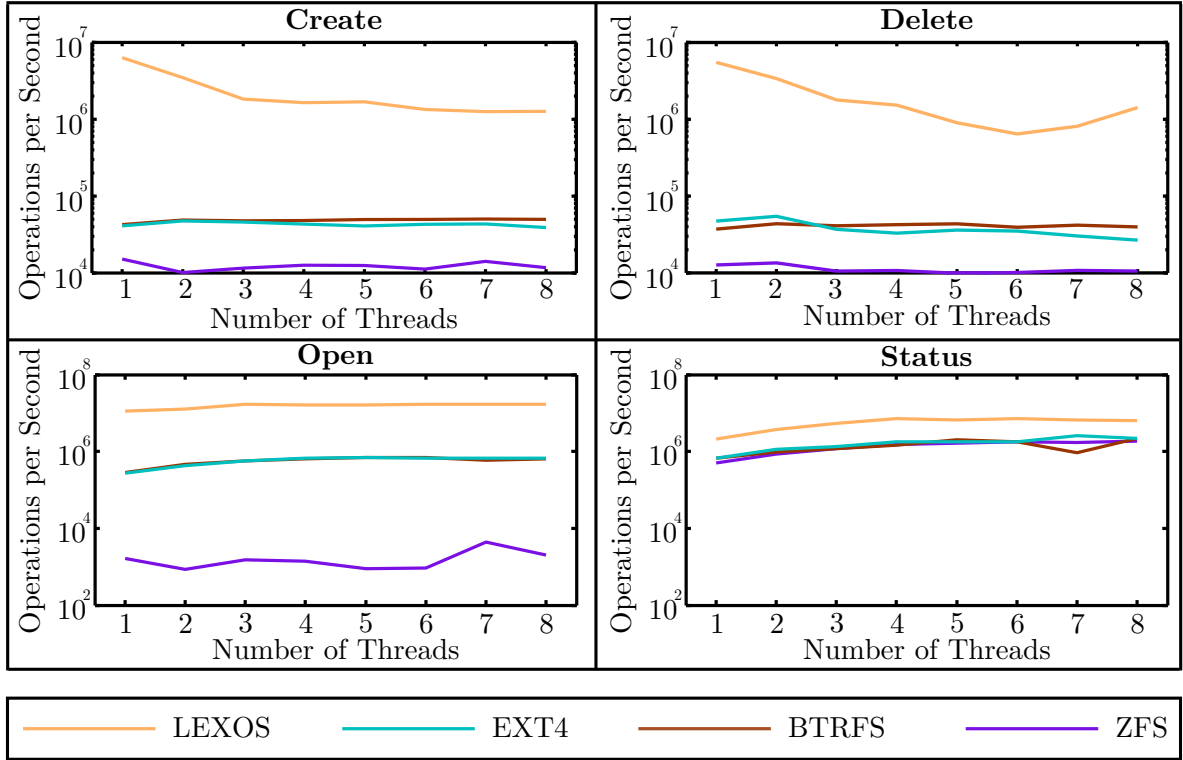
In this benchmark, 500 000 objects were processed. This number of objects was chosen in order to have a fair test with all file systems processing the same number of objects. It should be noted that this number of objects may be small enough to be held in the memory during the benchmark. That is why memory performance may be reached. LEXOS and ext4 were already tested with 10 000 000 objects. The results are plotted in Appendix D. ZFS and btrfs were also tested with this number of objects, but the measurements were stopped after a while due to a long running time. In order to investigate and compare the metadata performance among the different file systems in more detail, further measurements are necessary.

The benchmark was executed with several threads, where the number of objects was equally divided among the threads. The close operation was not explicitly measured, but in combination with other operations. For example, if an object was created successfully it was closed immediately.

Concerning the file systems, all files are managed in one directory. So, the file systems act like a flat file systems as it is implemented by the object store.

Figure 5.9 shows the results of the metadata performance of each operation. The diagrams plot the speed (in operations per second) on a logarithmic scale with respect to the number of threads. As it was expected, LEXOS brings excellent results in metadata performance and outperforms the other file systems, which are magnitudes slower.

Concerning the create operation this may have several reasons. For example, file systems have a lot of management overhead due to path lookup, checking permissions etc. In contrast to this, LEXOS only needs to fetch the corresponding inode from the inode table without doing any preprocessing. This is a fast operation, since the index of the next free inode is stored in the header. However, the throughput decreases with an increasing number of threads. A reason for this is the locking of the inode allocation procedure. Only one thread is allowed to allocate an inode at a time. ZFS has the lowest performance in creating files, while ext4 and btrfs have nearly the same performance in this operation. Similar results can be seen in the bottom left plot of Figure 5.9 showing the performance results concerning the open operation. In order to open a file, another path lookup and permission check is necessary. ext4 and btrfs have nearly the same performance, since both file systems may first use the VFS inode cache in order to find the corresponding inode that was recently used. Again, LEXOS only needs the object ID to access the inode table which explains the high performance.



**Figure 5.9.:** Comparison of the metadata performance.

The status operation is a very common operation concerning metadata. This operation returns a struct for an object/a file containing important information about it. The performance of LEXOS has its maximum with four threads. When more than four threads are used, the performance slightly decreases. ZFS, ext4 and btrfs have nearly the same performance concerning the status operation.

Similar to the create operation, the performance concerning the deletion of object decreases with multiple threads. When an inode is de-allocated, the corresponding operation is locked for one thread at a time what may impact the performance. ZFS again shows the lowest performance, while the performance of btrfs and ext4 lies at the same magnitude.

## Conclusion and Chapter Summary

LEXOS can compete with well-established file systems concerning the performance. Especially the read operation shows an excellent performance. Synchronization is an expensive operation and it reduces the write performance significantly. Nonetheless, LEXOS shows a good performance and even outperforms the more optimized file system ext4 and btrfs in some cases. All in all, there remain a lot of possibilities for further optimizations. The extent data structure and the allocation algorithm were found to be the main performance bottleneck and are therefore of special interest for further

improvements. LEXOS provides a transaction feature that allows read and write requests to be batched in one operation. With this, the object store obtains additional information about the data to be processed. This allows to apply optimizations on this data in order to reach a higher performance. Such an operation is not implemented by the other file system that were tested. Benefits were especially observed with this feature in combination with data synchronization. Using transactions results in a speedup of 50 compared to the write performance without transactions and with synchronization. Regarding the metadata performance, the object store outperforms all tested file systems significantly. A high metadata performance was the main design aspect of the object store.



# Chapter 6.

## Conclusion and Future Work

### 6.1. Summary and Conclusion

In this thesis, a new object store was designed and developed. With this object store it is possible to access a storage device via objects. It is used as a low-level abstraction layer for storage. Additionally it performs basic storage management either as a light-weight file system or as a supporting layer for other file systems. Its I/O and metadata performance was evaluated and compared with three well-known file systems by testing several different benchmark configurations.

The object store was designed to be *low-level* and *extent-based*: It has direct access to the storage device and manages the data on it by using appropriate data structures. The internal representation of objects was implemented by inodes, a well-known concept from file systems that store object relevant metadata. Moreover, inodes maintain links to the locations of the data belonging to the object. In order to store and track object-relevant data on the device, the extent mapping scheme is used. Extents are stored in a singly-linked list. Free block and inode management are handled by using bitmaps.

Additionally, a completely new interface was designed. This interface provides operations to open and close the object store and moreover to manipulate objects. This includes metadata operations like create, open, close and delete an object and additionally I/O operations, so that data can be written to and read from the object. A special feature of this interface is that it is possible to pass additional information about the data that will be processed. With this information the object store can apply appropriate optimization, so that the data can be handled more efficiently. As a first step, transactions were implemented. These transactions allow to batch several read or write operations. The transaction procedure checks if these operations are contiguous and can be then executed as one operation. Concerning write operations and synchronization this was found out to be very efficient.

The results in Section 5.2 show that the object store has an excellent metadata performance. When the object store is applied in parallel file systems, this is an essential requirement. A parallel file system has to handle a lot of file requests from numerous clients. The overall file system performance would extremely suffer from a bottleneck in

metadata management. A lot of parallel file systems use traditional file system as an object store back-end. But as it was found out in Section 5.2, some of these file systems provide poor metadata performance. That is why using an object store like LEXOS instead of a full-featured file system would eliminate this bottleneck.

## 6.2. Future Work

Although the object store already provides basic storage management, an excellent metadata performance and a good I/O performance, some optimizations and extensions are possible. In the following, some suggestions and directions for future work are stated.

As it was detected in Chapter 5, the extent list constitute the main performance bottleneck of the object store. When several objects are processed, there is a low chance that extents of one object are consecutive on the disk. This was especially observed with the sequential access pattern. A lot of disk seeking operations are necessary in order to find a specific extent. That is why it is reasonable to further improve the extent list data structure. For example, the inode could store more than one extent information, so that the effort due to the list traversal can be reduced. Another possibility would be to apply a different data structure, e.g., a tree data structure as it is used in the file systems btrfs and ext4.

The extent list traversal is an often used operation in the object store. It is applied when data is copied to/from an object and when the physical offset is calculated. As the size of the extent list grows, the operation performs slower. Consequently, there is a need to further improve this operation or even to apply a different extent data structure.

The extent allocation algorithm was also found to be an expensive operation. It has a great impact on the write operation, since it has to be applied when the object size has to be extended. A suggestion is to modify the block bitmap. Instead of encoding each single block, free extents can be binned by size and sorted by location in order to quickly locate free space possibly near to the object that has to be extended.

The transaction operation only provides a basic merging of operations. The merging conditions can be further improved and refined, in order to allow a more efficient handling of data.

The object store library is thread-safe. However, the locking of specific functions was implemented very coarsely which may have an impact on the performance. In future work, it can be investigated in which way the locking can be adjusted more finely. For example, the extent allocation is currently locked for one thread at the time. It has to be analyzed how the code can be adapted and the locking can be set, so that more than one thread can execute this operation in order to achieve more concurrency.

The technique *copy-on-write* was also considered for this object store, but not as a mandatory feature. However, it would be hugely useful, especially in terms of data consistency and additional features that are possible with copy-on-write. Snapshotting



is one of these features.

In summary it can be stated that the object store has a great potential. With this object store design and implementation, a good foundation for further optimizations, research and development was created.



# Bibliography

- [ADD<sup>+</sup>08] Nawab Ali, Ananth Devulapalli, Dennis Daless, Pete Wyckoff, and P. Sadayappan. Revisiting the Metadata Architecture of Parallel File Systems. Technical Report OSU-CISRC-7/08-TR42, 2008.
- [AEHH<sup>+</sup>11] Sadaf R. Alam, Hussein N. El-Harake, Kristopher Howard, Neil Stringfellow, and Fabio Verzelloni. Parallel I/O and the Metadata Wall. In *Proceedings of the sixth workshop on Parallel Data Storage*, PDSW '11, pages 13–18, New York, NY, USA, 2011. ACM.
- [Ama13] Amazon Simple Storage Service. <http://aws.amazon.com/en/s3>, 2013.
- [BAH<sup>+</sup>03] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The Zettabyte File System. Technical report, 2003.
- [BBU13] Hans Burchard, Karsten Bolding, and Lars Umlauf. GETM: Documentation of Stable Version 2.4.0. <http://www.getm.eu/data/getm/doc/getm-doc-stable.pdf>, 12 2013.
- [BHB99] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux As a Case Study: Its Extracted Software Architecture. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 555–563, New York, NY, USA, 1999. ACM.
- [CCDSP01] Giuseppe Cattaneo, Luigi Catuogno, Aniello Del Sorbo, and Pino Persiano. The Design and Implementation of a Transparent Cryptographic File System for UNIX. In *USENIX Annual Technical Conference, FREENIX Track*, number 1-880446, pages 10–3. Citeseer, 2001.
- [Daw07] Pawel Jakub Dawidek. Porting the ZFS File System to the FreeBSD Operating System. *Proc. of AsiaBSDCon*, pages 97–103, 2007.
- [Ger] German Climate Computing Center. <http://www.dkrz.de>.
- [GJS<sup>+</sup>13] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison Wesley, 2013.
- [GLi] GLib Reference Manual. <https://developer.gnome.org/glib/2.38/>.
- [GNA<sup>+</sup>97] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David

- Rochberg, and Jim Zelenka. File Server Scaling with Network-attached Secure Disks. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '97, pages 272–284, New York, NY, USA, 1997. ACM.
- [GNU13] GNU C Library. <http://www.gnu.org/software/libc/>, 2013.
- [Goo] Google Cloud Storage. <https://cloud.google.com/products/cloud-storage>.
- [MCB<sup>+</sup>07] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The New Ext4 Filesystem: Current Status and Future Plans. In *Proceedings of the Linux Symposium*, volume 2, pages 21–33. Citeseer, 2007.
- [Mcv91] L.W. Mcvoy. Extent-like Performance from a UNIX File System. In *Proceedings of Winter 1991 USENIX (Dallas, TX)*, pages 33–43, 1991.
- [MM03] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association.
- [Ope] Open Group Base Specifications Issue 7. <http://pubs.opengroup.org/onlinepubs/9699919799/>.
- [Ora12] Oracle Solaris 11.1 Data Sheet. <http://www.oracle.com/us/products/servers-storage/solaris/oracle-solaris-11-ds-186774.pdf>, 2012.
- [POS13] POSIX Threads Programming - Tutorial. <https://computing.llnl.gov/tutorials/pthreads/>, 2013.
- [RBM13] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-Tree Filesystem. *Trans. Storage*, 9(3):9:1–9:32, August 2013.
- [RG10] Aditya Rajgarhia and Ashish Gehani. Performance and Extension of User Space File Systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 206–213, New York, NY, USA, 2010. ACM.
- [Sch03] Philip Schwan. Lustre: Building a File System for 1,000-node Clusters. In *Proceedings of the Linux Symposium*, 2003.
- [SDA<sup>+</sup>10] Richard P. Spillane, Sagar Dixit, Shrikar Archak, Saumitra Bhanage, and Erez Zadok. Exporting Kernel Page Caching for Efficient User-Level I/O. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–13, Washington, DC, USA, 2010. IEEE Computer Society.

- [SKH<sup>+</sup>12] Sandra Schröder, Michael Kuhn, Nathanael Hübbe, Julian Kunkel, Timo Minartz, Petra Nerge, Florens Wasserfall, and Thomas Ludwig. Scientific Computing: Performance and Efficiency in Climate Models. In Erwin Grosspietsch and Konrad Klöckner, editors, *Proceedings of the Work in Progress Session, 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, number 31 in SEA-Publications, Johannes Kepler University Linz, 2012. Munich Network Management Team, Institute for Systems Engineering and Automation.
- [SM09] Margo Seltzer and Nicholas Murphy. Hierarchical File Systems Are Dead. In *Proceedings of the 12th conference on Hot topics in operating systems*, HotOS’09, pages 1–1, Berkeley, CA, USA, 2009. USENIX Association.
- [Tan08] A.S. Tanenbaum. *Modern Operating Systems*. GOAL Series. Pearson Prentice Hall, 2008.
- [WBM<sup>+</sup>06] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI ’06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.
- [WLBM07] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters. In *Proceedings of the 2Nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing ’07*, PDSW ’07, pages 35–44, New York, NY, USA, 2007. ACM.
- [XXSM09] Jing Xing, Jin Xiong, Ninghui Sun, and Jie Ma. Adaptive and Scalable Metadata Management to Support a Trillion Files. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC ’09, New York, NY, USA, 2009. ACM.



# List of Figures

2.1	The difference between the block-oriented access of storage (left) and an object-based interface (right). . . . .	13
2.2	The inode pointer structure implemented by the ext file system [CCDSP01]. . . . .	15
2.3	Extent mapping between the data of a file and the disk blocks. . . . .	16
2.4	The extent tree of ext4 [MCB <sup>+</sup> 07]. Once the first four extents are exceeded, a tree is spanned having additional extents as leaf nodes. . . . .	17
2.5	A file being loaded into memory of a process by memory mapping. . . . .	19
3.1	Usage of LEXOS in the user space. (a) A user application program linked with the object store library <i>liblexos</i> . (b) A user application program linked with a file system library <i>libfs</i> . The file system library is linked with the object store library. . . . .	23
3.2	The object store layout like it is installed when the store is created on a device. . . . .	25
3.3	The extent list data structure. It constitutes the internal representation of object-related data. . . . .	26
3.4	Merging of the first three write request into one operation with offset 0 and count 20. . . . .	31
4.1	Creating a bit mask for a specific bit field. . . . .	37
4.2	Toggling (XOR) and querying (AND) specific bits in a bitmap. . . . .	38
4.3	Two possibilities of extent allocation. Further blocks can be added to the last extent of object 1, if the next free block is right beside it. The extent of object 2 cannot be extended with further blocks. A new extent can be created for it at one of the two possible next free blocks. . . . .	40
4.4	Two cases of calculating the size of the extent needed to be allocated. . . . .	45
4.5	Extent traversal like it is executed when data is copied (top figure) or the real disk offset is calculated (bottom figure). . . . .	48
5.1	Overview of the possible configurations of the small-access benchmark. . . . .	55
5.2	Block size test for a sequential and random access pattern (one object and one thread). . . . .	57
5.3	Comparison of different options of LEXOS (one object, sequential access pattern, one thread). . . . .	58
5.4	Comparison of different benchmark configurations with several threads without synchronization. . . . .	59

5.5	Performance evaluation of LEXOS by using different benchmark configurations with several threads with synchronization. . . . .	61
5.6	Comparison of the write performance without synchronization. . . . .	64
5.7	Comparison of the write performance with synchronization. . . . .	65
5.8	Comparison of the read performance. . . . .	66
5.9	Comparison of the metadata performance. . . . .	68
D.1	Metadata performance measured with 10 000 000 objects. . . . .	95



# List of Listings

4.1	Extent Information . . . . .	38
4.2	Object store data structure. . . . .	41
4.3	Creation of the objectstore mapping . . . . .	41
4.4	Synchronization of the whole mapping. . . . .	43
4.5	Un-mapping the device. . . . .	43
4.6	Object data structure. . . . .	44
4.7	Copy data operation. . . . .	47
4.8	A request saving the key attributes for write and read operations. . . . .	49
4.9	Push read and write request into a queue. . . . .	50
4.10	Checking if requests can be merged. . . . .	50
4.11	processing the read requests . . . . .	51
5.1	Main loop of the small-access benchmark. . . . .	54
A.1	Using the mklexos command to create an object store from the command line. . . . .	86
B.1	Object store operations. . . . .	87
B.2	Object operations . . . . .	88
B.3	Transaction API. . . . .	90
C.1	API usage example. . . . .	91



# Appendices



# Appendix A.

## Setup and Configuration of LEXOS

The code of LEXOS is available in the *Git* repository that has to be cloned:

```
1 git clone  
  ↪ https://redmine.wr.informatik.uni-hamburg.de/git/julea-lexos
```

Beside the code the repository includes scripts in order to configure and install the object store. For this, the *waf*<sup>1</sup> build system is used.

In order that LEXOS can be used, the GLib has to be installed on the system. Then, LEXOS is configured by

```
1 ./waf configure
```

This checks if all preliminary libraries are available on the system and sets the path where LEXOS will be installed. By default these are `/usr/lib`, `/usr/bin` and `/usr/include`. If another path is desired, it can be defined by the `prefix` option. This command for example sets the home directory as the install path:

```
1 ./waf configure --prefix=$HOME
```

As the next step, the code is compiled by

```
1 ./waf
```

And the header files and the LEXOS library are installed by

```
1 ./waf install
```

After this, LEXOS can now be used to create and manage an object store.

### A.1. `mklexos` - Command Line Tool

LEXOS provides a command line tool in order to the object store creation from the command line. This command takes the path of the device, the number of inodes and the block size as parameters. Listing A.1 shows an example of the usage of this command. If LEXOS was configured to be installed on a path different from `/usr/*`, the environment variable `LD_LIBRARY_PATH` has to be set to the path where the LEXOS library is located.

---

<sup>1</sup><https://code.google.com/p/waf/>

**Listing A.1:** *Using the `mklexos` command to create an object store from the command line.*

```
1 mklexos --number-of-inodes=100 --block-size=4096 --path=/dev/sdb
```

Setting the path is mandatory, otherwise the command is not executed. If the number of inodes and the block size is not set, default values are used. After creating the object store, it is closed immediately.

# Appendix B.

## LEXOS API

In the following all API operations are listed. Their functionality, input and output parameters are described in each listing.

### B.1. Object Store Operations

*Listing B.1: Object store operations.*

```
1  /**
2  * Creates the object store.
3  *
4  * @param path Path of the device managed by object store.
5  * @param blocksize Blocksize of each block
6  * @param number_of_inodes Number of inodes
7  * @param error GLib error handling variable
8  * @return objectstore the object store
9  */
10 ObjectStore* lstore_create(const gchar *path, guint64 blocksize,
    ↪  guint64 number_of_inodes, GError** error);
11
12 /**
13 * Opens the objectstore.
14 *
15 * @param path Path to an existing object store, for example /dev/sdb
16 * @param error GLib error handling variable
17 * @return objectstore the object store
18 */
19 ObjectStore* lstore_open(const char* path, GError** error);
20
21 /**
22 * Closes the objectstore.
23 *
24 * @param objectstore an opened object store
25 * @param error GLib error handling variable
26 * @return ret TRUE, if success
27 */
28 gboolean lstore_close(ObjectStore* objectstore, GError** error);
```

## B.2. Object Operations

*Listing B.2: Object operations*

```
1  /**
2   * Creates a new object.
3   *
4   * @param objectstore an existing objectstore where the object store
5   *   ↪ will be created.
6   * @param size a size in bytes for a possible preallocation
7   * @param error GLib error handling variable
8   * @return object a new object
9   */
10 Object* lobject_create(ObjectStore* objectstore, guint64 size,
11   ↪ GError** error);
12
13 /**
14 *
15 * Gives the object a new size that is greater than the current size.
16 *
17 * @param object the object to be resized.
18 * @param size the new size
19 * @param error GLib error handling variable
20 * @return ret TRUE is success
21 */
22 gboolean lobject_resize(Object* object, guint64 size, GError**
23   ↪ error);
24
25 /**
26 * Deletes an object.
27 *
28 * @param object the object to be deleted
29 * @param error GLib error handling variable
30 * @return ret TRUE if success
31 */
32 gboolean lobject_delete(Object* object, GError** error);
33
34 /**
35 * Opens an object in an object store.
36 *
37 * @param objectstore an existing objectstore
38 * @param object_id unique identifier of the object to be opened.
39 * @param error GLib error handling variable
40 * @return object opened object having the specified object_id
41 */
42 Object* lobject_open(ObjectStore* objectstore, guint64
43   ↪ object_id, GError** error);
44
45 /**
46 * Closes an object.
47 *
48 * @param object the object to be closed.
```



```

45  * @param sync flag if content of object should be synchronized
    ↪ before closing.
46  * @param error GLib error handling variable
47  * @return TRUE if success
48  **/
49  gboolean lobject_close(Object* object, LObjectSyncMode sync,
    ↪ GError** error);
50
51  /**
52  * Returns the object status in form of a struct.
53  * Contains the path of the object store, the id and the size of the
    ↪ object.
54  *
55  * @param[in] object the object which status is queried
56  * @param[out] ObjectStat a struct containing information about the
    ↪ object.
57  */
58  ObjectStat* lobject_get_stat(Object* object);
59
60  /**
61  * \brief Write a specific number of bytes from a buffer into an
    ↪ object beginning at an offset within this object.
62  *
63  *
64  * @param object an object
65  * @param offset location at which will be written to the object
66  * @param count number of bytes that will be written from buf to
    ↪ object.
67  * @param buffer data that will be written to the object.
68  * @return bytes Number of bytes that will be written to the object.
    ↪ Can be less than count.
69  */
70  guint64 lobject_write(Object* object, guint64 offset, guint64 count,
    ↪ gpointer buffer, LObjectSyncMode mode, GError** error);
71
72  /**
73  * \brief Reads a specific number of bytes from an object beginning
    ↪ at an offset.
74  *
75  *
76  * @param object an object
77  * @param offset location from where data is readout
78  * @param count number of bytes that will be read
79  * @param buffer buffer holding readout data
80  * @return bytes number of bytes that were readout
81  */
82  guint64 lobject_read(Object* object, guint64 offset, guint64
    ↪ count, gpointer buffer, GError** error);

```

## B.3. Transactions

*Listing B.3: Transaction API.*

```
1  /**
2  * Starts a transaction. This initializes the queue of the object for
   ↪ which the transaction is started.
3  *
4  * @param object an object for which the transaction is invoked
5  */
6  void ltransaction_start(Object* object);
7
8  /**
9  * \brief Executes the transaction by processing the queue of
   ↪ write/read requests.
10 *
11 * Write requests are merged in a larger request, while read
   ↪ requests are handled separately (because of separated read
   ↪ buffers).
12 *
13 * @param object an object for which a transaction is executed
14 * @param error GLib error handling variable
15 * @return bytes number of bytes processed
16 */
17 gint64 ltransaction_execute(Object* object, GError** error);
```

# Appendix C.

## API Usage Example

Listing C.1 shows an example how the API can be used in order to manage object within an object store. In order that this example can be used, an object store has to be created before. In this example a file named `test.dat` is used that will be managed by an object store. This file can be replaced with an arbitrary one (for example with a device file). Its size has to be chosen appropriately, so that all metadata and data can be hold. In this example one object is processed. First random data of a size of 1 MB is written and then read back from the object.

The code has to be compiled first, either with the make tool or the waf build system. LEXOS has to be installed on the system. When it was not installed to the standard paths (`/usr/*`), then the `LD_LIBRARY_PATH` and the `PKG_CONFIG_PATH` have to be set in advance. Moreover, the GLib is needed to execute the example code. Consequently, the code has to be linked with the LEXOS and the GLib libraries.

*Listing C.1: API usage example.*

```
1  #include <lexos.h>
2  #include <glib.h>
3
4  int main(int argc, char const *argv[])
5  {
6      GError* error;
7      ObjectStore* store;
8      Object* object;
9      ObjectStat* status;
10     guint64 id;
11     guint64 size = 1048576; //1MB
12     guint64 offset = 0;
13     guint64 bytes;
14     guint64 prealloc_size = 1048576;
15
16     gchar write_buffer[size]; //random data
17     gchar read_buffer[size];
18
19     error = NULL;
20
21     store = lstore_open("test.dat",&error);
22     if(error) //error handling
23     {
24         g_print("%s\n",error->message);
```

```
25     return 1;
26 }
27
28 object = lobject_create(store, prealloc_size, &error);
29 if(error) //error handling
30 {
31     g_print("%s\n", error->message);
32     return 1;
33 }
34 else
35 {
36     status = lobject_get_stat(object);
37 }
38
39 lobject_close(object, LOBJECT_NO_SYNC, &error);
40 //lobject_close(object, LOBJECT_SYNC, &error);
41 if(error) //error handling
42 {
43     g_print("%s\n", error->message);
44     return 1;
45 }
46
47 object = lobject_open(store, status->id, &error);
48 if(error) //error handling
49 {
50     g_print("%s\n", error->message);
51     return 1;
52 }
53
54 bytes = lobject_write(object, offset, size, write_buffer,
55                        LOBJECT_NO_SYNC, &error);
56 g_assert(bytes == size);
57 if(error) //error handling
58 {
59     g_print("%s\n", error->message);
60     return 1;
61 }
62 bytes = lobject_read(object, offset, size, read_buffer, &error);
63 g_assert(bytes == size);
64 if(error) //error handling
65 {
66     g_print("%s\n", error->message);
67     return 1;
68 }
69
70 lobject_delete(object, &error);
71 if(error) //error handling
72 {
73     g_print("%s\n", error->message);
74     return 1;
75 }
76
```

---

```
77 | lstore_close(store,&error);
78 | if(error)//error handling
79 | {
80 |     g_print("%s\n",error->message);
81 |     return 1;
82 | }
83 |
84 | return 0;
85 | }
```

The code can be compiled with the following command

```
1 | gcc lexos_test.c $(pkg-config --cflags --libs glib-2.0)
   | ↪ $(pkg-config --cflags --libs lexos) -o test
```

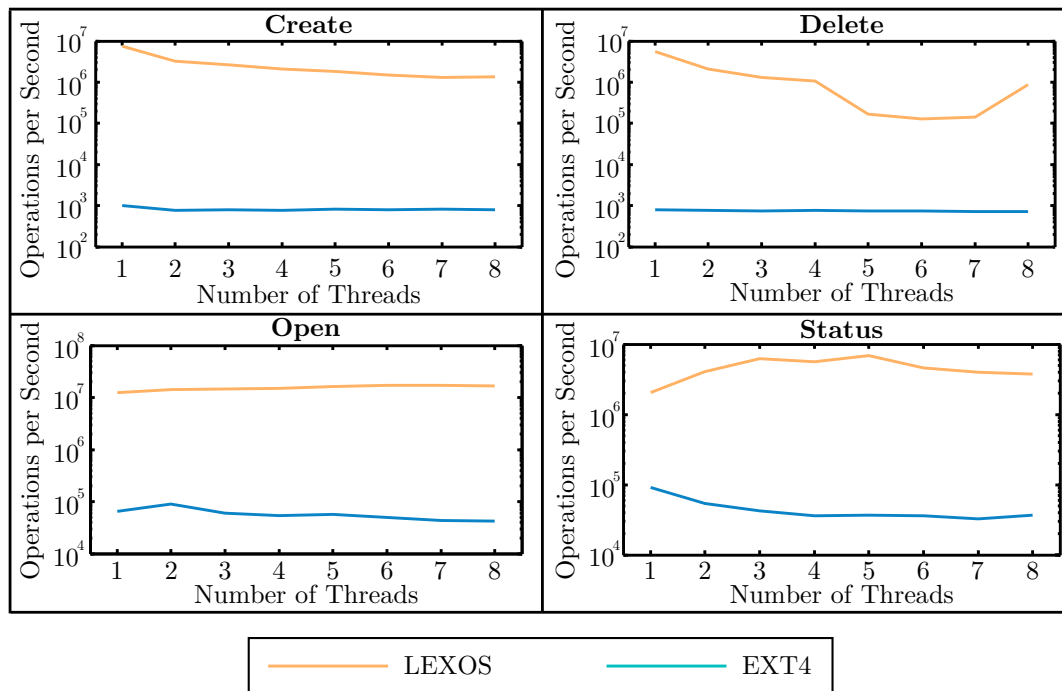
This expects the path `PKG_CONFIG_PATH` to be set in order to find the LEXOS library and GLib if one of them is not located in the standard paths.



## Appendix D.

### Metadata Performance

As mentioned in chapter 5 the number of objects that was tested may be too small. All objects could be held in the main memory during the benchmark execution. That is why a higher number of objects should be tested for LEXOS and the file systems. In the first steps of the evaluation of the metadata benchmark, an appropriate number of objects were determined. For this, the benchmark was started with 10 000 000 objects. Only LEXOS and ext4 were able to produce results in an adequate time.



**Figure D.1.:** Metadata performance measured with 10 000 000 objects.

Figure D.1 shows the resulting metadata performance of LEXOS and ext4 when 10 000 000 objects are processed. It is noticeable that there is no major difference to the metadata performance shown in Section 5.2. LEXOS outperforms ext4 in terms of metadata performance. Concerning the status operation ext4 has a slightly lower performance than with 500 000 objects. For further investigations, ZFS and btrfs also have to be tested with 10 000 000 objects.





# Danksagung

Ich möchte mich herzlich bedanken bei

- Prof. Dr. Thomas Ludwig für die Möglichkeit meine Masterarbeit in der Arbeitsgruppe Wissenschaftliches Rechnen durchführen zu können und für die Übernahme des Erstgutachtens,
- Prof. Dr.-Ing. Ritter für die Übernahme des Zweitgutachtens,
- Michael Kuhn für die hervorragende Betreuung und Unterstützung,
- den wissenschaftlichen Mitarbeitern und Studenten der Gruppe Wissenschaftliches Rechnen,
- meinen Kommilitonen für die gemeinsame Studienzeit,
- Christian Adolff und
- Johanna Hennig-Schröder und Holger Schröder.



## **Erklärung**

Ich versichere, dass ich die Arbeit selbstständig verfasst und keine anderen, als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internetquellen – benutzt habe, die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ich bin mit der Einstellung der Masterarbeit in den Bestand der Bibliothek des Fachbereichs Informatik einverstanden.

Hamburg, den 19.12.2013 .....