Vrije Universiteit Amsterdam

Bachelor Thesis

Optimizing Metadata Handling with vkFS: A Hybrid Key-Value Store File System leveraging RocksDB

Author: Vincent Nikolas Kohm (2726735)

1st supervisor: Dr. ir. Animesh Trivedi
daily supervisor: Krijn Doekemeijer
2nd reader: Tiziano De Matteis

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

June 28, 2024

Abstract

File systems are ubiquitous and fundamental to most computer system, playing a crucial role in organizing and storing data efficiently. They are essential for a multitude of applications, particularly in the domain of High-Performance Computing (HPC) and Data Intensive Scalable Computing (DISC). As the volume of files in these systems grows, efficient metadata handling becomes increasingly important for overall system performance. However, research has highlighted challenges with metadata management in conventional file systems such ext4, btrfs, or xfs when tasked with metadata-intensive tasks (1, 2, 3). In addition to these challenges, academic research has underlined the potential of small-data storage systems, particularly key-value stores, for balancing read and write operations effectively. This research focuses on hybrid key-value store file systems, which aim to enhance metadata performance. We address two primary research questions in this work: What design options are available for hybrid key-value store file systems, and what are their respective advantages and disadvantages? Furthermore, how can such a system be designed and implemented effectively?

To address these research questions we first conduct a comprehensive literature review of highly-regarded academic papers, examining various design strategies. We then detail our own design approach for our system vkFS, as well as explaining its implementation intricacies. Finally, we evaluate the performance of vkFS using the benchmarking tool Filebench. Our literature review identifies several notable design options, including horizontal scaling, a flattened directory tree, and the decoupling of metadata from file data. These strategies offer distinct benefits, such as improved lookup performance, reduced serialization overhead, and efficient scaling in distributed environments. Nevertheless, they also present challenges, such as handling large value fields, complex renaming operations, and management intricacies associated with horizontal scaling. Despite our system not achieving the expected performance improvements, our

evaluation provides insights into potential aspects affecting the performance. Though not verified by means of quantitative data, we propose hypotheses to explain these outcomes.

The artifact can be accessed at: https://github.com/Vincent881909/vkfs

Contents

1 Introduction **1**

1.1 Context . . . 1

1.2 Problem Statement . . . 2

1.3 Research Questions . . . 3

1.4 Research Methodology . . . 3

1.5 Thesis Contributions . . . 4

1.6 Plagiarism Declaration . . . 5

2 Background on Conventional File Systems **7**

2.1 Extended File System Version 4 . . . 7

2.2 XFS . . . 11

2.3 Btrfs . . . 11

3 Exploring Hybrid File Systems: A Comparative Literature Review **13**

3.1 Methodology . . . 14

3.2 TableFS . . . 14

3.3 IndexFS . . . 20

3.4 LocoFS . . . 26

3.5 Summary . . . 32

4 System Design of vkFS **35**

4.1 vkFS Design Requirements . . . 35

4.2 Overview of vkFS System Architecture . . . 35

4.3 Inode Allocation . . . 37

4.4 vkFS Key-Value Pairs . . . 37

5 Implementation of vkFS **39**
5.1 Implementation Requirements . . . 39
5.2 Development Environment . . . 40
5.3 Project Hierarchy . . . 40
5.4 FUSE Operations . . . 41
5.5 vkFS RocksDB Class . . . 42
5.6 Inode Allocation Handler . . . 42
5.7 File System State . . . 43
5.8 Metadata Headers . . . 43
5.9 Interaction with the Underlying File System . . . 44
5.10 Limitations of vkFS . . . 45

6 Evaluation **47**
6.1 Hardware Configurations . . . 48
6.2 Metadata-Intensive Micro Benchmarks . . . 48
6.3 Data-Intensive Macro Benchmarks . . . 54
6.4 Evaluation Discussion . . . 57

7 Conclusion **59**

References **61**

A Reproducibility **65**
A.1 Abstract . . . 65
A.2 Artifact check-list (meta-information) . . . 65
A.3 Description . . . 65
A.4 Hardware Host . . . 66
A.5 Virtual Machine . . . 67
A.6 Installation . . . 67
A.7 Experiment workflow . . . 70
A.8 Evaluation and expected results . . . 72

1

Introduction

1.1 Context

File systems are integral of nearly all computer systems, serving as the primary mechanism for data storage and retrieval. They are ubiquitous, found in personal mobile devices, personal computers, embedded systems, and large-scale data centres amongst others illustrating why they are a critical component in the world of modern storage. This ubiquity necessitates that file systems must operate efficiently and accurately across a wide range of scales and use cases. For instance, personal Apple phones utilise the Apple File System (APFS)(4), Linux computers commonly employ the Extended File System version 4 (Ext4)(5), and large distributed data-intensive applications often rely on the Google File System (GFS)(6). Additionally, open-source file systems are regularly introduced (7),(8),(9),(10) (11) reflecting the ongoing advancements and contributions in this domain - and with continuous development in technology more can be expected in this field (12). Therefore, this underlines the importance of rigorous research, experimentation, and evaluation in the field of file systems.

Moreover, file systems are fundamental to operations across various industries. Businesses, governments, and society depend on file systems to run their services and manage their data. This reliance makes file systems a critical component in many sectors, driving the need for ongoing research and development. Research in file systems not only advances the technology but also educates and prepares future developers and designers in this field, ensuring that digital infrastructure remains robust and future-proof. Therefore, it is essential to address the current challenges in file systems, explore potential solutions, evaluate existing technologies, and generate innovative ideas for the future.

Conventional file systems have undergone a wide array of improvements and changes to increase performance and mitigate system crashes or data corruption (13). Due to the vast range of file system applications a diverse array of design decisions were made to optimise for specific use cases. For example, xfs was developed to handle large files, numerous files, expansive directories, and to deliver very high-performance I/O operations while the Hadoop Distributed File System (HDFS) was designed for large-scale data storage and processing across multiple machines (14, 15). Additionally, as observed in the Linux source code, numerous file systems are supported in a Linux environment to cater to various user needs, including specific use cases, workloads, performance characteristics, and reliability requirements.

The above highlights a fundamental observation - no single file system can meet all use cases. This fact in combination with ongoing technological advancements drives the continuous development of improved file systems and innovative design architectures.

1.2 Problem Statement

One specific responsibility of file systems is to handle metadata operations. These operations include but are not limited to creating directories, deleting files, altering file permissions, or retrieving file attributes such as size, ownership, or access and modification timestamps. Metadata operations can comprise up to 80% of the overall file system tasks (16). With this in mind, it is therefore not surprising that the absence of metadata optimisation techniques can create performance bottlenecks, especially in environments with large file numbers such as in High Performance Computing (HPC) or the data intensive scalable computing (DISC) world (2). Furthermore, scalable systems will soon need to handle billions of small files, challenging both large and local file systems (17).

Another use case for file systems is to access many small files, which imposes the challenge of many random disk I/Os (18). Research has shown that the median file size on personal computers is only 4KiB (19), which means that most file system operations are performed on small files and their metadata. Conventional file systems such as Ext4 managing magnetic disks have realized the importance of sequential allocation as well as large data transfer sizes (1). However, modern workloads dominated by metadata lookup and small files expose an inefficiency which occurs as a result of many random I/Os on disk even in advanced file systems such as Ext4, XFS, and Btrfs (1). These inefficiencies will be further explored in **Section 2.2**.

This leads to the opportunity to research and analyse file system designs that achieve performance improvements in the above cases. This will be the primary objective of this work.

1.3 Research Questions

In this work, our aim is to improve the performance of file systems by improving the handling of metadata workloads. Specifically, we focus on solutions that employ a hybrid file system leveraging key-value stores. To address these challenges, we identify two key research questions.

RQ1 What designs can be employed to improve metadata handling within a hybrid model of a key-value store and file system, and what advantages or limitations do they present?

Our first research question directs our attention to the design aspects within the scope of hybrid key-value store file systems. It is imperative to research and analyse existing architectures to provide an answer to **RQ1**. Striving for innovation and contribution in the domain of storage design requires a deep understanding of current architectures and their advantages and drawbacks. This study allows us to combine the design decisions of other authors and use this knowledge to introduce our own design, which further aids in the process of answering **RQ1**.

Our second research question shifts the focus to implementing such a design and performing an evaluation using benchmarking tools. Thus, to further validate our findings from **RQ1** we introduce our second research question. **RQ2** How to design and implement a Hybrid Key-Value Store File System that optimises for metadata intensive workloads?

By exploring **RQ2** we aim to gather insights that will help us to draw a conclusion about how to solve the problem at hand.

1.4 Research Methodology

To provide a meaningful answer to both research questions that address the aforementioned problem statement we employ several different research methodologies.

First, to answer **RQ1** we conduct a comprehensive literature review to analyse existing designs of hybrid key-value store file systems. In this section we touch on different design decision that were made and discuss its advantages as well as their potential drawbacks. By analysing different designs we aim to understand the complexities involved in developing

an efficient file system and identify the limitations of specific design choices. With this knowledge we aim to gain a deeper understanding of the problem and its potential solutions and utilise this to design our own solution to the problem. Once the design is finalised, we proceed to implement our solution. This implementation will serve to address **RQ2** and there are several reasons for providing a functional file system implementation as part of our research methodology.

First, it forces us to deeply understand the intricacies involved in designing a versatile file system, thereby aiding in the thorough investigation of **RQ1**. Second, it provides us with a tangible tool to evaluate the chosen design by means of benchmark tools and to make direct comparisons to existing file systems. Lastly, this rigorous approach enables us to form a meaningful conclusion regarding **RQ1**.

1.5 Thesis Contributions

With this work we are confident to provide several contributions to the industry and community.

1.5.1 Conceptual

This work provides a comprehensive overview and analysis of currently existing hybrid file systems. It also offers an examination of the limitations inherent to the design of hybrid file systems. This allows for future research to build on these insights, driving innovation and improvements in hybrid file system architectures.

1.5.2 Artifact

This research is accompanied by a fully functional implementation utilising a design that has been carefully selected after thorough research in the domain of file systems. Furthermore, the implementation serves as a source of inspiration for other related file systems and is accessible and adaptable through open-source distribution.

1.5.3 Experimental

We provide a thorough evaluation of the chosen design by means of benchmark tools. By providing the exact workloads used for this evaluation we allow future research to make direct comparisons further driving the development of modern storage solutions. We also offer guidance for reproduceability for validation purposes.

1.5.4 Societal

The complete document serves as a starting point for any future exploration in the domain of storage research. Different contributions of this paper such as conceptual, technical, or experimental, aid as valuable resources for both academic researchers and industry professionals looking to advance the field of hybrid file systems.

1.6 Plagiarism Declaration

I confirm that this thesis work is my own work, is not copied from any other source (person, Internet, or machine), and has not been submitted elsewhere for assessment.

2

Background on Conventional File Systems

Before delving into the design decisions of hybrid file systems that leverage key-value stores, we start by reviewing traditional file systems, without a hybrid design, specifically Ext4, XFS, and Btrfs. These file systems are well-established and often used as a direct comparison when developing new file system designs (1). This review lays the foundation for understanding the design of hybrid file systems and provides context on how they differ, while also highlighting the current inefficiencies in handling metadata operations. Additionally, it will emphasize key design principles of the mentioned conventional file systems, setting the stage for further analysis in **chapter 3**. Moreover, these file systems will serve as a direct comparison point for our implementation, which will be explored in Section 6. Understanding their design and functionality is essential for a meaningful evaluation of our implementation against these established standards.

2.1 Extended File System Version 4

The Extended File System Version 4 is the newest file system of the extended file system family and is considered as the default Linux File System since its introduction in 2006 (20). The transition from Ext3 to Ext4 mainly addressed scalability, performance, and reliability factors and has been integrated into Linux since version 2.6.1. Ext4 divides the drive space into a large number of blocks representing the basic unit that accommodates the further structures of the file system. These blocks are then grouped into block groups. The information held for each block group is in the descriptor table. Two other blocks are located near the start of each block group and represent the block usage bitmap and inode

usage bitmap. These bitmaps indicate which blocks and inodes are currently in use and are an essential part of the extended file system family. Another block is represented by the superblock. This block is responsible for storing file state information such as the number of free and used blocks of the entire file system. The primary copy of the superblock is stored at an offset of 1024 bytes from the start of the device but copies are present in block group 1 and powers of 3, 5, and 7. The inode table represents another block within a block group and holds a linear array of inodes allocated for its block group. Lastly, each block group contains a data block. **Figure 2.1** depicts a simplified overview of a block group. It is important to note that not every block group follows this precise pattern for example the superblock is not present in every block group. Additionally, since Ext3 this file system family uses a journal to protect from metadata inconsistencies in case of a system crash. The journal itself consists of more individual blocks taking up an entire block group. However, a more detailed description on this unit is out of scope for this work.

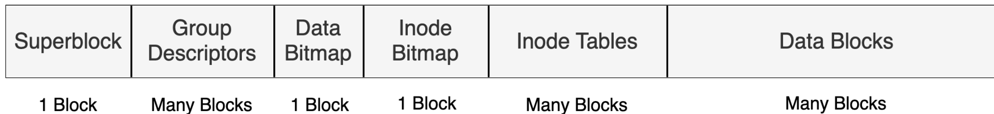

Figure 2.1: Ext4 File System Organisation divided into Block Groups

2.1.1 Locating Metadata in Ext4

Since the primary objective of this work revolves around enhancing metadata handling, it is essential to examine the specifics of the metadata stored within the Ext4 file system. As mentioned earlier, the inode table is a linear array of inodes holding metadata of all entries in the current block group. The inode is defined as a `struct ext4_inode` in its source code and holds all the metadata for a file. For readability reasons **Figure 2.2** shows only the first eleven fields of this structure. During operations such as `stat`, `chown`, or `chmod` this metadata is accessed.

While analysing the entire struct may be out of scope for this work let's examine some essential fields such as `i_mode`, `i_links_count`, `i_blocks_count`, and `i_flags`.

The `i_mode` field is a bitmask and serves two primary functions. First, it defines the file type most commonly Directory and Regular File. Second, this bitmask specifies the file's access permissions - an important part of any secure file system. This unit utilizes a bitmask to identify file permissions and file types.

```
struct ext4_inode {
  __le16  i_mode;        /* File mode */
  __le16  i_uid;         /* Low 16 bits of Owner Uid */
  __le32  i_size_lo;     /* Size in bytes */
  __le32  i_atime;       /* Access time */
  __le32  i_ctime;       /* Inode Change time */
  __le32  i_mtime;       /* Modification time */
  __le32  i_dtime;       /* Deletion Time */
  __le16  i_gid;         /* Low 16 bits of Group Id */
  __le16  i_links_count; /* Links count */
  __le32  i_blocks_lo;   /* Blocks count */
  __le32  i_flags;       /* File flags */
  ...
```

Figure 2.2: Ext4 Inode Struct

The `i_links_count` refers to the number of hard links which allows for a inode number to be present more than once as well as different names. These entries, however, all point to the same inode and thus the same data. The `i_blocks_count` field indicates how many blocks have been allocated for the contents of the file. The `i_flags` field is a another bitmask combining various flags that indicate special attributes of a file, such as requirements for secure deletion, immutability, synchronous writes, encryption, and other metadata properties.

In order to locate the metadata of a file the file system follows a systematic approach. The determination of the block group is necessary to find the necessary inode table which holds the inode and its metadata of our requested file.

The `s_inodes_per_group` field in the superblock structure specifies the number of inodes defined per group. Given that inode allocation starts at 1 (inode 0 does not exist), the following formulas can be used:

1.

$$\text{block group} = \frac{\text{inode} - 1}{\text{s_inodes_per_group}}$$

2.

$$\text{local inode index} = (\text{inode} - 1) \bmod \text{s_inodes_per_group}$$

3.

$$\text{offset} = \text{index} * \text{s_inode_size}$$

Offset	Size	Name	Description
0x0	_le32	inode	Number of the inode that this directory entry points to
0x4	_le16	rec_len	Length of this directory entry. Must be a multiple of 4
0x6	_le16	mane_len	Length of the file name
0x8	char	name[EXT_NAME_LEN]	File name

Figure 2.3: Struct to hold Information about a Directory Entry

For instance, to find the inode of "file.txt" located at path "/home/foo/file.txt", the process begins by iterating from the root directory, which has the known inode number 0. By applying formula 1. and 2. we get the local inode index 0 at block group 0. Since the index is 0 formula 3 does not need to be used since the offset will be 0. Since "/" is a directory we use the inode to allocate the directory entries in the corresponding data block and locate the entry with name "foo". In the ext4 source code, a directory entry in the data block is defined as a struct called `struct ext4_dir_entry_2` that is stored as a byte array.

Figure 2.3 shows the outline of this data structure. The filename can be read from the directory entry and compared to the entry name we are seeking. If it matches we retrieve the inode value and begin traversing for the next entry. If it does not match, we continue to the next directory entry. Once we have retrieved the inode value, we can utilise the above formulas again to locate the directory entry of "foo". Similar to the root entry, foo is a directory and its inode points to a data block containing all its directory entries. Ext4 also supports Hash Tree Directories to improve the performance of looking up a specific entry. Once the correct directory entry is located we retrieve the inode number of the file and use the formulas above to find the block group and its local inode index. Once this has been finalised we can access the files metadata.

This example serves as a demonstration to why lookup operations can be expensive in terms of calculations required to find the specified entry. In addition, deeply nested entries in a file tree aggravate the cost of a lookup operation. Moreover, as the number of files grow in a file system the need for optimisation techniques becomes evident. Key-value stores can alleviate these associated lookup costs which we discuss in detail in **chapter 3**.

2.2 XFS

XFS is a high-performing file system optimized for scalability and parallel throughput developed by SGI in October 1993 for IRIX. XFS can handle large files, many inodes, and large directories. Parallel access is optimized by dividing the storage medium into semi-autonomous allocation groups. Delayed and extend-based allocations are used to improve data contiguity and I/O Performance (21).

Similar to how Ext4 divides the disk space into block groups, xfs file ystem is internally divided into multiple equally sized areas called allocation groups (AGs). Every instance of an AG can be seen as an independent unit that is able to manage its own inodes, space usage, and other secondary metadata. This allows xfs to use multiple AGs to perform most operations in parallel without performance loss, even with multiple concurrent access patterns. Multiple b-trees are used per allocation group to manage bookkeeping data such as the locations of free blocks, allocated inodes, and free inodes.

2.3 Btrfs

Btrfs is another open source Linux files system that has been continously developed since its introduction in 2007 by organisations such as Red Hat, Intel, or Oracle. Key features included cyclic redundancy checks, for all metadata and data to achieve data integrity and security, efficient writable snapshots, and multi device support. In fact, once the objectives of btrfs is to work for a wide variety of workloads and to maintain performance as time passes. While many other storage systems are designed to optimize for specific narrow use cases, btrfs is intended to work well on system of different scales ranging from smartphones to enterprise servers. This leads to several challenges such as scalabilty, data integrity, and disk diversity.

Btrfs divides the disk layout into a forest of b-trees. Disk blocks are managed in extends, while checksums are used for integrity, and reference counting for space reclamation. The performance of a btrfs is directly influenced by the availability of long contiguous extends which becomes difficult to maintain as the system ages due to fragmentation. Because of snapshots, disk extents can be pointed to by multiple file system volumes which in turn makes defragmentation a real challenge. First, the extents can only be moved after all source pointers are updates. Second, file contiguity is preferred for all snapshots (22).

3

Exploring Hybrid File Systems: A Comparative Literature Review

In this chapter, we aim to conduct a comprehensive review of existing solutions for metadata management. By providing this review, we hope to deepen the understanding of designing a hybrid key-value store file system to optimize metadata handling and the challenges it entails. Additionally, we aim to achieve a better understanding of why existing solutions can introduce drawbacks in their design.

This work shifts the focus to a particular subset of file systems: hybrid file systems that utilize the performance benefits of key-value stores. This class of scalable, small-data storage systems emphasizes (NoSQL) interfaces and large in-memory caches. Key-value stores deploy different data structures to store data, each achieving a specific balance regarding the fundamental trade-offs between read, update, and memory amplification (23). Recent research has recognized key-value stores as an efficient way to store data, making them a promising technique for file system metadata storage.

In this work, we focus on file systems that use key-value stores to store metadata to achieve performance gains in metadata-intensive tasks. We review existing solutions, outlining their key design decisions and discussing the trade-offs associated with these decisions. We do this by examining three case studies in depth: TableFS, IndexFS, and LocoFS. Finally, we provide a short summary of our findings in the last section of this chapter.

3.1 Methodology

In order to conduct this survey, we collected relevant academic papers through various approaches. First, we examined the most cited paper in this domain, TableFS, and considered the papers cited by its authors. This provided us with a solid list of papers relevant to our problem statement. Additionally, since the TableFS paper is highly regarded, we examined other publications from the same set of authors and discovered IndexFS, a continuation of TableFS that tackles the same problem with different design choices. Here, we looked at all the papers referenced by IndexFS to expand our literature base for research. Lastly, we utilized Google Scholar to browse all publications that cited either IndexFS or TableFS, further increasing our collection of related papers.

3.2 TableFS

3.2.1 Introduction

TableFS is a hybrid key-value store file system utilising the LevelDB key-value store to store metadata and files smaller than a pre-defined threshold while storing larger files in the underlying file system. The authors underline the rise of data-store systems called key-value stores, LevelDB in particular. LevelDB is a fast key-value store written at Google mapping keys to values represented as arbitrary byte arrays (24). The primary objective of TableFS is to improve performance for workloads dominated by metadata operations and small files. The authors of this work published their paper "TABLEFS: Enhancing Metadata Efficiency in the Local File System", in 2013 at the USENIX Annual Technical Conference (USENIX ATC '13) (1). The motivation for this work came from the fact that scaling for metadata and small files has not seen the same kind of development as for high bandwidth and large file transfers. Thus, this work aims to tackle this problem in the form of an implementation followed by an evaluation of their design, making direct comparisons to established file systems using metadata and data-intensive workloads.

In the following section we aim to provide details on the components of TableFS's design prior to presenting a overview of its overarching architecture. TableFS uses FUSE, a userspace file system framework to implement TableFS. We provide a short review of this framework discussing its advantages and why it fits in this context. We continue by delving into an analysis of the underlying data structure of LevelDB, an Log Structured Merge Tree (LSM-tree) exploring how it facilitates fast lookups while ensuring quick writes and

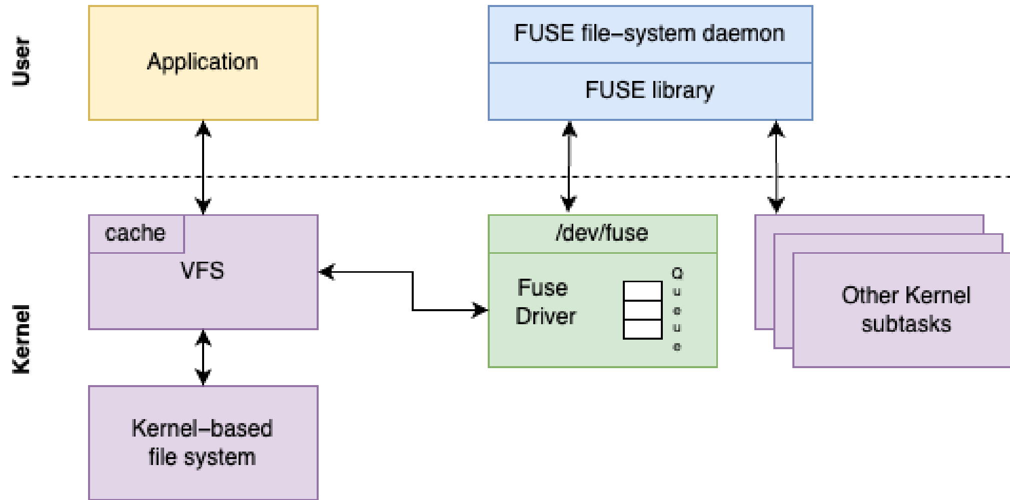

Figure 3.1: FUSE high-level architecture

updates. Lastly, we touch on the table schema employed to store metadata and small files in the key-value store.

3.2.2 FUSE

Most file systems offer a standardized interface for applications to read, write, and update data. While mico-kernels implement file systems in user space, most file systems are part of monolithic kernels (25). By operating at the kernel level, implementations sidestep the expensive message-passing overheads found in micro-kernels and user-space daemons. However, developing file systems in the kernel is challenging as bugs can crash whole systems. Thus, file systems developed at userspace level allows developers to easily prototype new stackable file systems adding specialized functionality to the common file system interface. Therefore, deciding whether to develop a file system in user space or at the kernel level involves balancing the ease of development against the performance inefficiencies introduced by user-space implementations. FUSE or file system in user space is the most popular user-space file system framework and consists of a kernel and a user-level daemon part. The kernel component is implemented as a Linux kernel module that registers a FUSE file system driver with Linux's Virtual File System (VFS). This FUSE driver works as a bridge for different types of file systems, each run by its own user-level program. A new block device is also registered by the kernel module named "/dev/fuse" from which daemons can read FUSE requests, processes them, and write their response back to the block device. **Figure 3.1** depicts the high level architecture of FUSE.

When the application performs an operation the request gets directed to the VFS which adds this request to the queue at the FUSE Driver. At this point of time the application process is in a waiting state. Once the operations has been added to the FUSE queue the FUSE user-level daemon picks up the operation from the block device and processes it using custom implementations of the FUSE library operations. Once processed, the result is written back to the block device "/dev/fuse" at the kernel. FUSE Kernel Driver then marks the process as finalised and wakes up the original application process. If an application performs a read request for a file whose contents are cached by the kernel the enqueuing process is omitted and the result is returned using the cache instead completely bypassing communication with the user-level FUSE daemon (25). This illustrates the overhead produced from using FUSE due to the message passing between modules.

3.2.3 LevelDB and its LSM Tree

TableFS uses the persistent key-value store LevelDB to store metadata as well as well as files below 4KiB. To understand why this particular key-value store is a good fit we must examine its design. A key-value store must define basic operations such as put and get to insert or retrieve data. Other operations are optional and include delete to remove a certain entry or scan to retrieves a range of key-value pairs. To store data in a key-value store several data structures can be utilised. LevelDB uses a Log Structured Merge Tree (LSM-tree) to store data in memory as well as on disk. An LSM-tree consists of different components where data can be considered to be carried on from one component to the next one where each subsequent component is larger than the previous one. To illustrate how data is stored in an LSM-tree we provide **Figure 3.2** based on a figure by Lu et al. (26).

If the client triggers a put operation it is first written to an in-memory table (Memtable) and optionally to a WAL (Write Ahead Log). This can be seen in **Figure 3.2** as step 1 and 2. All changes to LevelDB are first written to the mutable memtable residing in memory. The WAL resides on persistent storage and allows to restore any lost changes after shutdown. All operations are also appended to the WAL so that in case of system failure all data in the memtable can be recovered which is necessary to restore the database to its original state. Once the memtable fills up or certain conditions (such as time passed) are met the mutable memtable as well as the WAL will be marked as dirty and the mutable memtable will be made immutable. This implies that the immutable memtable can only be read at this point and not altered as it is prepared to being stored on persistent storage. To achieve this, the immutable memtable is converted into a Sorted String Table (SST).

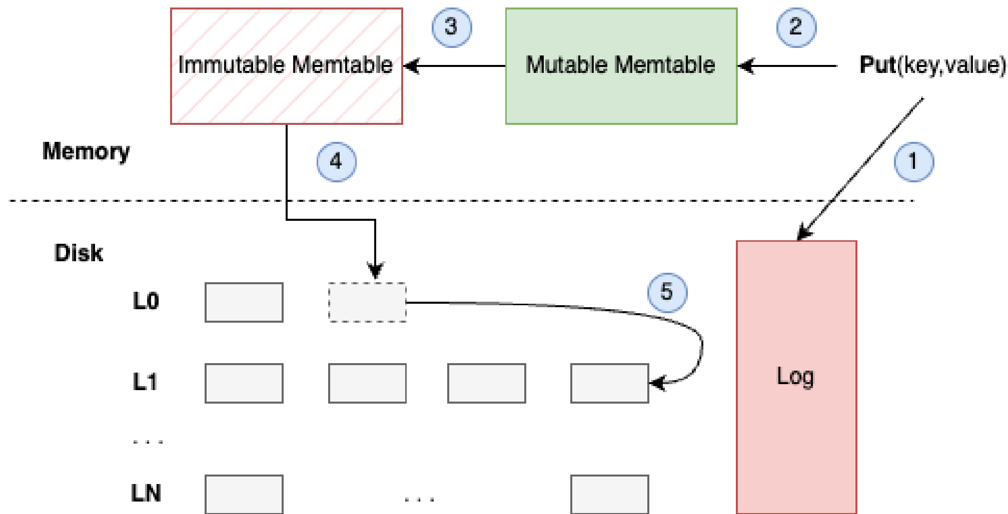

Figure 3.2: LSM-tree Design of LevelDB

This structure holds sorted key-value pairs. The SST is then appended to the first level of the tree also known as L0. This can be seen as step 4 in **Figure 3.2**.

Once the top level of the tree reaches a pre-defined size some of the SSTables from the top level must move to the next level of the tree - a process known as compaction depict in the final step of **Figure 3.2**. The key difference between L0 and L1 is that L1 only contains unique key ranges while L0 may still hold overlapping key ranges. To achieve this property merge sort is applied during the compaction process. The newly formed non-overlapping SSTables are appended to L1 while the original SSTables used for this compaction process are removed. This idea continues to deeper levels of the tree where each level is smaller than the next level. The depth of this tree is configurable (27). The details of compaction are relevant since compaction operations are inherently resource-intensive and improper tuning of the LSM-tree can lead to frequent compactions, resulting in many costly operations. This can lead to degrading the performance of the file system.

Moreover, during a lookup operation we must traverse all components at the worst case ranging from the mutable memtable all the way to the last level LN. However, since LSM-trees always append data we can safely return the first entry that matches our key in this traversal as it is guaranteed to be the most up to date entry of the key we are interested in.

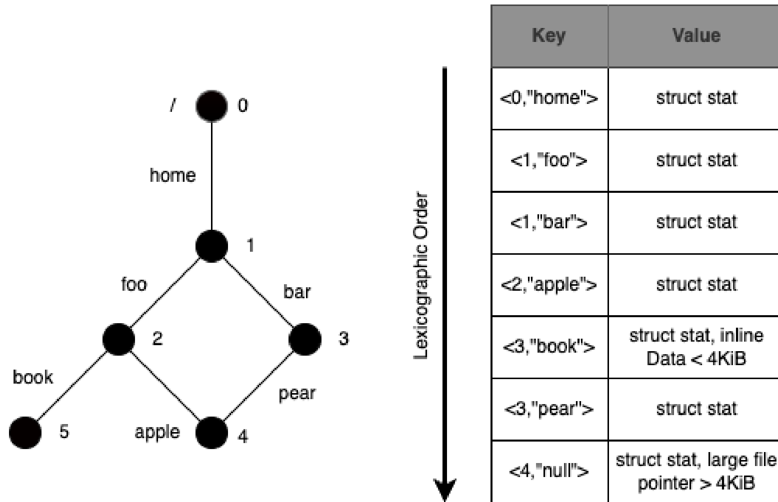

Figure 3.3: Table Scheme of TableFS

3.2.4 LevelDB Table Schema

Now that we have covered the essentials of how the underlying data structure (LSM-tree) of LevelDB functions we can shift the focus to more fine-grained design choices, the LevelDB Table Schema. This includes how the key pointing to the value is composed and what exact data the value field holds. The key is composed of the entries parent directory 64-bit inode number and the final component of its path (filename). The entries in the LSM-tree are ordered by the variable-length key. The value contains the metadata or inode attributes as seen in **Figure 2.2**. If the corresponding data does not exceed a predefined threshold T the data is appended to the inode attributes and stored inside the LSM-tree. The authors of TableFS determined this threshold to be 4KiB. This scheme can be seen in **Figure 3.3** and has been adapted from the original TableFS paper (1).

3.2.5 High Level Overview

The authors of TableFS saw opportunity for performance improvement with regard to scaling metadata intensive workloads and decided on several design choices to approach this improvement as listed in this section. In this section, we present a short review of how these individual components work together to form the overall architecture of their file system. **Figure 3.4** depicts the above-mentioned components and shows how they work together to form a fully functional file system. Similar to **Figure 3.1**, this figure divides the architecture into two distinct areas: user space and kernel space. In the user space, the Benchmark Application Process, shown in red, initiates an operation targeting the file

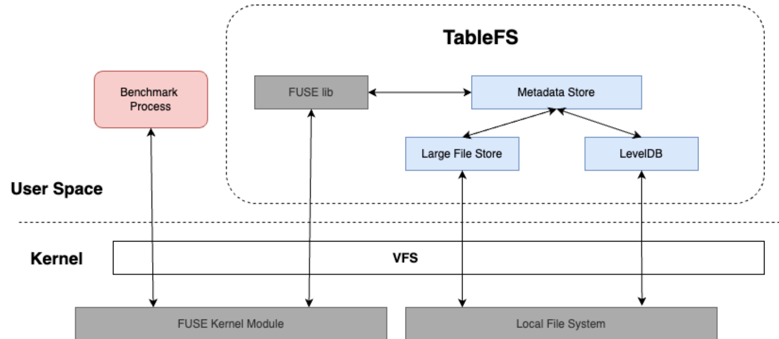

Figure 3.4: High Level Overview of TableFS Architecture

system and sends this request to the virtual file system (VFS). While this process is in a waiting state, the VFS writes the request to the FUSE Kernel Module at "/dev/fuse," where it is subsequently processed by the Fuse library. TableFS then defines this operation and uses its customized file system logic to decide whether the operation needs to be handled by LevelDB or the Large File Store. As mentioned previously, LevelDB holds all metadata and small inline data. The Large File Store refers to all data that exceed the threshold for storing this data as an inline file. Thus, if data is requested to be written or read and exceeds a predefined threshold, the Large File Store performs a system call back into the kernel to perform the requested operation. In this design, the authors of TableFS have chosen ext4 as the underlying local file system to store large objects.

The figure also illustrates an arrow between the Local File System and LevelDB, which is due to the nature of the LSM-tree data structure. In the design of TableFS this is ext4. As discussed in the previous section, once the in-memory memtables reach a certain size or specific conditions are met, or any write to the WAL, their contents are flushed to disk, requiring kernel access. Once either the Large File Store or LevelDB finishes its operations, it will return to the FUSE library, which writes the results to the Fuse Driver in the kernel. The benchmark process, which awaits these results, is reactivated and the results are delivered to the application.

3.2.6 Analysis of Advantages and Drawbacks

In the preceding section, we provided a comprehensive review of the design implemented by TableFS. Here, we aim to briefly analyze the benefits and limitations of this design.

The design entails storing metadata, file names, and inline data as the value field for a given file. This can lead to value fields reaching sizes up to the maximum file name length plus the size of the inode and 4KiB of inline data. A drawback is that key-value stores

can experience performance degradation as the value field size increases (3). Additionally, the (de)serialization process can become costly. During this process, the value for a key is fetched from the key-value store and stored in memory. Most operations will only modify a specific field of the inode, such as access or modification time. In this case, only a few bytes are altered, but the entire value field must be written back to the key-value store (3). With smaller value fields, this overhead could be kept at a minimum.

The use of an LSM-tree enables TableFS to attain high write performance due to the data buffer provided by the memory table (1). For instance, this design permits for quick bulk insertions without the need to immediately write all new entries to disk. However, this can also result in data loss during failure scenarios such as a system crash. To mitigate this, a well-tuned WAL is essential, along with mechanisms to recover lost data, adding further complexity to the design. Another drawback is that the performance of TableFS depends on the tuning of the key-value store in use, specifically LevelDB in this case. Improper initialization of LevelDB settings can lead to system slowdowns. However, this can be seen as an advantage since careful tuning allows developers to optimize the system for specific access patterns, albeit adding additional complexity (28).

One significant drawback of the current TableFS implementation is the overhead introduced by FUSE. As mentioned in the FUSE section, developing in user space results in a storm of messages between modules. Consequently, it is difficult to derive meaningful insights about TableFS's overall performance. Nonetheless, the authors sought to reduce this overhead by creating a TableFS library version, which runs all benchmarks within the application, thus bypassing nearly all kernel calls (1, 25).

Lastly, one more disadvantage stems from utilizing a single LSM-tree as the chosen data structure for metadata storage. As the file system tree expands, the underlying LSM-tree also increases in size. Consequently, lookup operations become more expensive. In contrast, as discussed in the background section, the lookup operation and its complexity in ext4 are not dependent on the size of the file system tree (3).

3.3 IndexFS

3.3.1 Introduction

The authors Kai Ren, Qing Zheng, Swapnil Patil, and Harth Gibson from Carnegie Mellon University introduced their paper "IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion" in 2014 and addresses the issue of metadata scalability. In their paper, they authors claim that distributed file systems such as PVFS,

Lustree, or HDFS focus on fast and parallel access to large files, while metadata scalability often lacks performance optimization techniques. Their prototype implementation named IndexFS attempts to tackle this issue by using a middleware design that can be integrated on top of existing file systems, providing a flexible and adaptable solution for metadata management.

The IndexFS architecture is table-based, allowing partitioning of the file system namespace. This namespace partitioning is performed on a per-directory basis, preserving disk locality as small directories stay on a single server, whereas larger directories are partitioned and distributed across multiple servers. Similarly to TableFS, IndexFS stores metadata in a log-structured manner, minimising disk seeks and improving I/O efficiency. The paper introduces two client-side caching techniques: bulk namespace insertion and stateless consistent metadata caching. Bulk namespace insertion optimizes for workloads that create many entries, while stateless consistent metadata caching helps mitigating hot spots in the system, reducing the load on metadata servers. In this section, we will conduct an in-depth review of this architecture followed by a discussion about the advantages and shortcomings of this design (2).

3.3.2 IndexFS Client

The IndexFS Client is a subcomponent of the whole design and is responsible for directing file system operations to the appropriate destination. IndexFS deploys a FUSE user-level file system to implement the logic behind their hybrid design. This means that the operations performed by the IndexFS client are directed to the FUSE user-level file system, which decides whether the operation is of type metadata or a data request. This distinction allows to direct data request to the underlying clustered file system making use of parallel I/O bandwidth. If the operation is classified as metadata, the IndexFS client directs this operation to the metadata indexing module sending the requests to the appropriate IndexFS server. Metadata requests can also consist of data requests if the read or write size does not exceed a predefined threshold of 64KiB. If a file in the underlying file system undergoes changes such as access time, size, or permissions, the IndexFS server will capture those changes on file close and apply the necessary changes on the metadata server. IndexFS clients make use of caching mechanisms to allow fast access to frequently requested metadata such as directory entries or directory server mappings. More details of this caching technique are discussed later in this design section.

3.3.3 IndexFS Server

Each individual IndexFS server is responsible for a non-overlapping subset of the file system metadata. The architecture is layered and consists of multiple components such as the Metadata cache, LevelDB key-value store, WAL, and SSTables. Similarly to TableFS an IndexFS server stores metadata and small files on the underlying file system using a LSM-tree employed by LevelDB. When data is written, it first resides in memory in a memory table before being flushed to disk in the form of a SSTable. The details of this mechanism are already discussed in the Design section of TableFS. The metadata of the file system is distributed among multiple IndexFS servers, each responsible for owning and managing a subset of the entire file systems metadata. The distribution occurs at the granularity of a subset of a directories entries. When the number of entries in a directory exceeds a threshold, IndexFS incrementally partitions its entries and metadata across multiple IndexFS servers. The underlying cluster file system is responsible for replication and encoding the LSM-tree's SSTable files and WALs.

3.3.4 Dynamic Namespace Partitioning

Index deploys a fine-level dynamic namespace partitioning to distribute both directories and directory entries across all IndexFS servers. The authors highlight that while many solutions partition the file system namespace based on a collection of directories that form a subtree, their design works at the directory subset granularity. This means that the file system is partitioned into directories and stored across multiple IndexFS servers if the directory has fewer than 128 entries. The advantage of this is that it preserves locality, and thus allowing for efficient reading of directory entries. The initial server assignment is executed by adapting the "power of two choices" load-balancing technique. This approach assigns a directory by choosing two random servers and selecting the one with fewer stored directory entries. This reduces the variance in the number of directory entries stored on the metadata servers.

When a single directory grows past a threshold of 128 entries, IndexFS makes use of the GIGA+ binary splitting technique. This technique hashes an individual directory entry to uniformly map the entire hash-range, which is then range-partitioned. Once the directory exceeds the 128 entry threshold, GIGA+ splits the hash-range in half and assigns one half to another IndexFS server. If the directory continues to grow, the hash-space can be further split to distribute the entries across multiple IndexFS servers. This idea continues until all servers have a partition of the directory. IndexFS clients cache partition-to-server

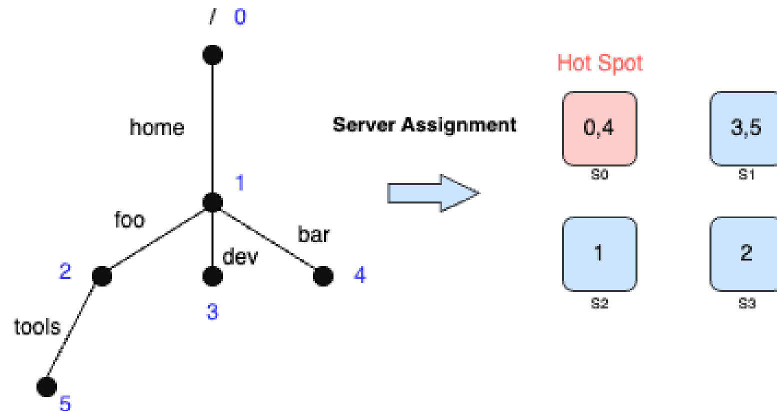

Figure 3.5: Distribution of file system directory tree among IndexFS servers

mappings to locate the entries of distributed directories. These mappings are cached at inconsistent rates to avoid cache consistency traffic while stale mappings are corrected by any server inappropriately accessed.

Figure 2.5 shows how a simple file tree and how it could be partitioned in a IndexFS environment. The hot spot indicated in red refers to frequently accessed paths like the root directory for example. Using the load balancing techniques mentioned above, IndexFS attempts to balance file and directories across all servers aiming for balanced and low variance distribution. In order to mitigate the hot spots IndexFS deploys a stateless directory caching mechanism.

3.3.5 Stateless Directory Caching

To support POSIX file I/O semantics many metadata operations are needed for parent directories resulting in a large amount of remote procedure calls (RPCs). Individual servers on a cluster use RPC protocol to communicate with each other. This introduces a messaging overhead for each message sent, and thus the number of messages should be minimized. GIGA+ uses a cache to map server entries for directories reducing the number of messages required to locate the correct IndexFS server holding the requested data. However, this approach could also result in stale mappings when a directory is partitioned or newly created. The overhead for this scenario is small though, since the server that was wrongly addressed can correct some or all of the clients cache. This eliminates a storm of cache updates or invalidation messages that occur in large-scale systems with consistent caching.

Nevertheless, once the correct IndexFS server has been identified there is still a need

for RPCs to test existence and permission checking. The authors of IndexFS highlight that exactly this access pattern is not well balanced across multiple servers, as directories further up in the tree are accessed far more frequently than directories at the bottom of a file tree. This scenario is shown in **Figure 2.5** and is indicated as a hot spot. To mitigate this, the authors implemented a lease for a given pathname offered to a client, causing any modification to that pathname to be delayed until the longest lease has expired. If a client requests to modify a pathname, it has to wait until the longest lease on that pathname has expired. Moreover, any requests for a lease on this pathname are blocked until the modifications have been completed. This is especially useful to mitigate a storm of invalidation messages in case a client attempts to write changes to a file that is currently being read by another client. For this to work, IndexFS assumes that the clocks on all machines in the cluster are synchronized. Finally, instead of assigning fix-sized lease duration, IndexFS adapts the duration of each lease based on two deciding factors. First, entries higher in the directory tree get a longer lease, since they are more likely to be access frequently than entries located towards the bottom of a file tree. Second, IndexFS monitors how often each directory is read compared to its write requests. Directories that are read often but rarely modified get longer leases.

3.3.6 Metadata Table Schema

Similar to TableFS, IndexFS stores metadata in LSM-tree. The mechanisms of this approach are covered in **section 2.2.2.2**. The table schema used for IndexFS is also almost identical with the small difference that inline data can be as large as 64KiB as opposed to 4KiB in TableFS.

3.3.7 Column-Style Table For Faster Insertions

The authors of IndexFS have identified that compaction of two levels of the LSM-tree is a resource intensive operation. Additionally, as the LSM-tree grows in size, lookup operations become more costly. For these reasons, IndexFS deploys a second LSM-tree table schema, called column-style. This table improves the throughput of insertion, modification, and single-entry lookup. With this much smaller table, IndexFS can disable compaction of the full metadata table. The column-style table only stores the file name, permissions, and a pointer to the most recent corresponding record in the full metadata table. This table is still sorted based on the same key. Due to its smaller size, its ability to cache is better compared to the full metadata table. Additionally, reduced size also implies far less

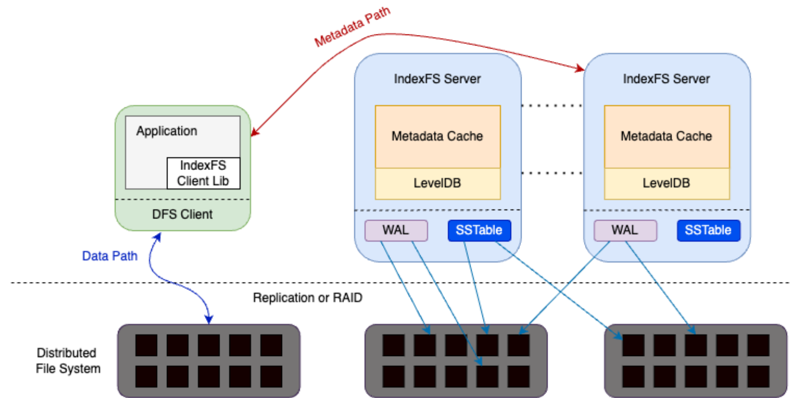

Figure 3.6: IndexFS architecture acting as middleware on top of an existing clustered file system

frequent compaction. The metadata it holds is sufficient to satisfy lookup operations and operations involving reading the contents of a directory. In the case the column-style table cannot satisfy the requested operation, it can utilize the stored pointer to access the full metadata table to retrieve the required metadata. In this scenario, only one additional disk read is required. The authors claim that this is still more efficient than a search on the large metadata table, especially on large LSM-trees.

3.3.8 Overall Architecture

Now that each individual component in the IndexFS is reviewed, we can provide an abstract overview of the entire system. This is depict in **Figure 2.6**.

Once an IndexFS client makes a request for a file operation, the request is categorized either as a metadata or data operation. Data operations traverse the blue data path and are directed to the underlying distributed file system. However, metadata operations are directed along the red line to the appropriate IndexFS server. The server utilizes the caching methods discussed above to retrieve metadata or make insertions into the metadata table using LevelDB. New entries are appended to the WAL and eventually written to an SSTable and flushed to the underlying clustered file system.

3.3.9 Analysis of Advantages and Drawbacks

IndexFS directly addresses some of the shortcomings of the TableFS design by partitioning the metadata store into dual components, significantly reducing the frequency of compaction operations. This architectural change mitigates latency variability in lookup operations, which is a noted issue in the TableFS approach. Furthermore, the adoption of a

Column-Style Table in IndexFS enhances cache performance by reducing the size of stored entries (1, 2).

Despite these advancements, the IndexFS design presents multiple drawbacks inherent to its distributed architecture. The concurrent operation of multiple IndexFS servers necessitates sophisticated data recovery mechanisms to maintain system integrity in the event of server failures. Typically, these mechanisms involve maintaining replicas of server states to enable seamless recovery (29). The intricate details of these mechanisms, however, are beyond the scope of this discussion.

Another significant challenge is that cache misses may force clients to interact with multiple servers to retrieve file information, introducing additional latency. Properly balancing these trade-offs is crucial for optimizing performance (2).

Moreover, the horizontal scalability of metadata servers, while offering potential for improved parallel access, introduces complexity in load balancing. The authors propose the "power of two choices" technique for load balancing, wherein two servers are randomly selected, and the server with fewer entries is chosen for metadata storage. Although this method aims to distribute load evenly, it can, in the worst case, lead to one server becoming disproportionately loaded, thus utilizing available resources inefficiently (2).

Scaling to hundreds of IndexFS servers also compounds management complexities. In scenarios where a directory grows significantly, it may lead to a situation where all servers hold entries for that directory. Consequently, a client would need to communicate with each server, resulting in high latencies due to increased messaging and networking demands (3).

In summary, while IndexFS introduces some improvements over TableFS, particularly in reducing compaction frequency and enhancing cache performance, it also presents significant challenges in terms of data recovery, load balancing, and management of large directories primarily inherent to its distributed environment.

3.4 LocoFS

3.4.1 Introduction

The final case study discussed in this paper is derived from the 2017 publication "LocoFS: A Loosely-Coupled Metadata Service for Distributed File Systems". The authors emphasize that data centers are transitioning from Petabyte-scale storage to Exabyte-scale storage, leading to scalability challenges for metadata services. They criticize the solution proposed by IndexFS, which involves deploying hundreds of IndexFS servers, as it leads to

management complexity and guarantee issues. They argue that most solutions only use 1 to 4 metadata servers, thereby avoiding these complexities. Moreover, they also emphasize the appropriateness of key-value stores for metadata storage due to their straightforward interface and excellent scalability. However, because of the directory tree semantics, there is a performance gap between key-value stores and file system metadata. For example, the authors state that IndexFS, which uses LevelDB as a metadata store, only achieves 1.7% of LevelDB's full performance. Moreover, the authors critizise that the performance of key-value stores like LevelDB and Kyoto Cabinet decreases with larger value sizes because handling bigger chunks of data takes more time during serialization and deserialization. In systems like IndexFS, where all metadata for a file is stored in a single value, any small change requires rewriting the entire value, leading to unnecessary overhead and reduced efficiency.

To address these issues, the authors developed LocoFS. The primary objective of LocoFS is to reduce dependencies among file system metadata, ensuring that essential file system operations engage with only a few metadata servers during their lifecycle. They aim to accomplish this by implementing a flattened directory tree structure to separate file metadata from directory metadata. Additionally, they further divide the file metadata into two components: Access Metadata and Content Metadata, to enhance the performance of the key-value store in use (3)

In the following section, we provide an in-depth review of the design of LocoFS followed by an analysis of its advantages and drawbacks.

3.4.2 Separation of File and Directory Metadata in LocoFS

LocoFS separates file and directory metadata to minimize the traversal path during lookup operations and to decrease latency. LocoFS employs a single Directory Metadata Server (DMS). The decision to use only one DMS is based on two factors. Firstly, the flattened directory structure, discussed later in this section, supports a large number of directories (approximately 10^8 for a DMS with 32GB memory). Secondly, a single DMS simplifies the Access Control List (ACL) verification process. During this process, permissions are verified to ensure data security. A DMS stores directory metadata in key-value pairs, where the full path name is used as the key and the metadata is stored in the value field. Kyto Cabinet is chosen as the key-value store in this design.

To manage file metadata, LocoFS employs several File Metadata Servers (FMS). It utilizes a consistent hashing method to allocate the file metadata among various servers. Consistent hashing is a strategy designed to distribute data uniformly across servers, thereby

reducing the need for data reallocation when servers are added or removed. To generate a key, a universally unique identifier (UUID) is combined with the file's name. The UUID originates from the file's parent directory. Similarly to the DMS, an FMS uses Kyoto Cabinet for storing key-value pairs.

3.4.3 Flattened Directory Tree

As mentioned in the introduction, the authors of LocoFS underline performance issues that occur in traditional file systems due to the complex dependencies between directory tree metadata and file metadata. This results in the fact that solutions that implement a hybrid key-value-store file system do not effectively exploit the advantages inherent to key-value stores. One of the advantages is that data are independently stored and accessed in key-value stores. While certain file systems achieve benefits by storing metadata in a key-value format, the dependencies within directory tree metadata still hinder file systems from fully leveraging the advantages of the key-value approach.

The authors address this in LocoFS by using a method called the Flattened Directory Tree to simplify the way metadata is organized. The primary objective is to reduce the complex links between directory and file information, arranging them in a way such that each component is independent of each other. **Figure 3.7** shows how this mechanism flattens a directory tree.

Traditionally, directories store their directory entries in their corresponding data blocks that hold information about each entry. Such directory entry is shown in **Figure 2.4**. LocoFS avoids storing directory entries within the directory data blocks, opting instead to restructure this format in a backward way.

In the flattened directory tree, directory entries, depicted as a dir entry in **Figure 3.7**, are separated from the directory metadata. The crosses in red in **Figure 3.7** represent the point of this decoupling mechanism. As a result, directories are stored with their metadata on DMS, files are stored with their metadata on FMS, while directory entries are stored on DMS if they belong to a subdirectory, and on the FMS if they represent a file in a directory.

These directory entries are stored using key-value pairs, where the key is composed using the hash value of the directory UUID. The corresponding value is the concatenation of all directory entries that are also stored on the same server. This means that a directory entry stored on a file server holds all file entries of a directory that are also present on the same server. Similarly, on the DMS a directory entry key-value pair holds concatenated directory entries consisting only of subdirectories.

For example, the figure above shows directory 2 and 5 in the flattened directory tree. These are directory entries from directory 1 and to be stored on the DMS. Here, the key will be the UUID of directory 1. Similarly, the green entries represent files, so the directory entries stored separately holds the UUID of the parent directory, and the value consists of a normal directory entry field as shown earlier in **Figure 2.3**. These entries point to data blocks indicated in purple.

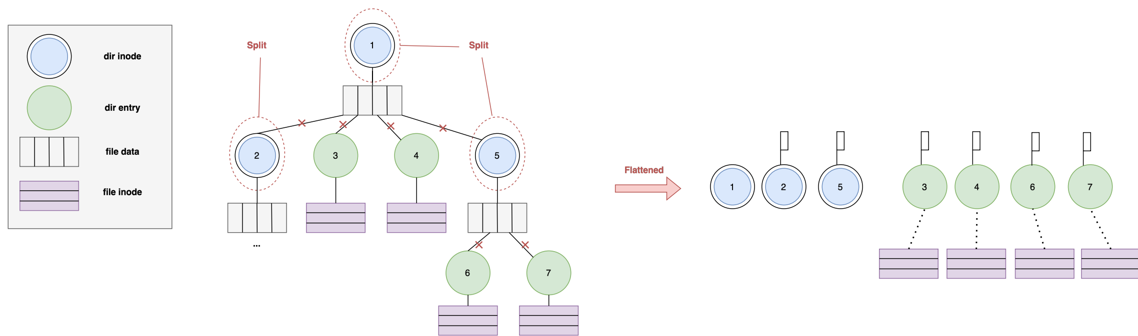

Figure 3.7: Flattened Directory Tree Mechanism in LocoFS

3.4.4 Client Directory Metadata Cache

LocoFS deploys a cache mechanism to store the metadata of directories. The objective of this is to reduce latency and improve scalability. The cache works on the client side and is especially useful for operations such as create, remove, and open since the metadata of the parent directory is required for these operations.

When a client creates a file within a directory, LocoFS stores the directory's inode from the DMS to the client. As a result, if another file is created in the same directory, the metadata of the parent directory can be retrieved locally. Additionally, during path traversal operations, LocoFS caches the inodes of directories along the path. This approach reduces latency and favors scalability. Similarly to IndexFS, LocoFS uses a lease mechanism, which allows the client to cache metadata for a default duration of 30 seconds.

3.4.5 Decoupled File Metadata

The authors of LocoFS highlight that the performance of key-value stores significantly drops as the size of the value field increases. Consequently, they suggest a different design approach to minimize the size of the value field. Moreover, modifying a single field in the inode structure necessitates rewriting the entire value field back to the key-value store, causing additional overhead. During this process, the value field must first be deserialized

	Dirrectory	File		Directory Entry
Key	Full Path	UUID + File Name		UUID
Value	ctime mode uid gid uuid	ctime mode uid gid	mtime atime size bsize suuid sid	entry
mkdir	x			x
rmdir	x			x
readdir	x			x
getattr	x	x	x	
remove		x	x	x
chmod	x	x		
chown	x	x		
create		x		x
open		x	x*	
read			x	
write			x	
truncate			x	

*stands for optional field updating in an operation (different file system have different implementations)

Figure 3.8: Key-Value Pairs in LocoFS and File System Operations and their Access Pattern

before it can be accessed. This procedure can reduce efficiency if the value field size grows. To address these overheads, LocoFS employs a decoupled file metadata mechanism that divides the metadata into two categories: Access and Content Fields. The access metadata section includes the fields atime, mode, uid, and gid. These fields define the access permissions of the files. The content metadata section includes the fields mtime, atime, block size, and file size. Each part is stored as one value and different file operations access different metadata values. **Figure 3.8** lists the different key-value pair stored across the whole system and which operations need access which pair. A cross in the table indicates that the corresponding operation needs to access or modify the corresponding metadata. It also displays the different key-value pairs in the previous section and which value they hold.

3.4.6 Overall Architecture

Now that we have covered all the essential components of the LocoFS design, we can provide an overview of the entire system. This is depicted in **Figure 3.9**. Most notably in this design are the two different types of metadata servers, one for file metadata and the other for directories. In addition, the FMS is further divided using a Metadata interface which distinguishes operations between Access and Content Metadata. The figure also illustrates

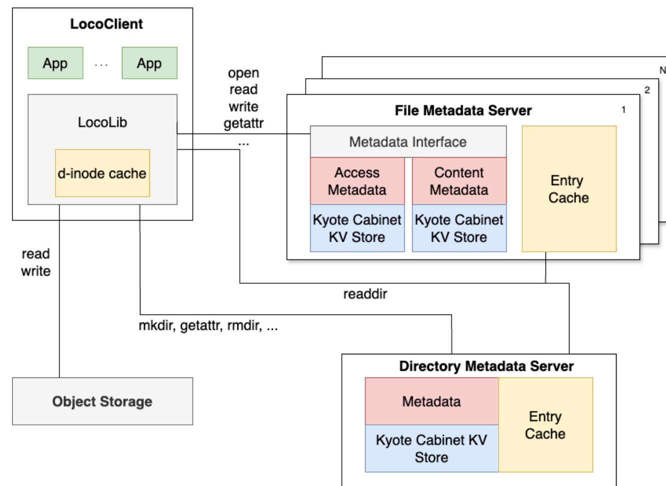

Figure 3.9: Overall Architecture of LocoFS

how different operations are redirected to different locations where they are optimized to be executed. This concludes the design section of LocoFS.

3.4.7 Analysis of Advantages and Drawbacks

LocoFS introduces a sophisticated approach to metadata management with both notable advantages and some drawbacks.

One drawback is related to rename operations. These operations can become complex because the flattened directory structure of the system requires meticulous metadata handling, which can lead to performance inefficiencies. This occurs because directory entries use their full path as the key. If a directory is renamed, all entries containing the old directory name in their path need to be updated, causing significant overhead. However, the percentage of rename operations in real-world applications is generally low (3).

The design decision to decouple file metadata into access and content metadata is effective, creating small value fields that allow the key-value store to perform well. However, some operations, such as "getattr" or "remove", need to access both access and content metadata, requiring two database queries. Specifically, "getattr" could cause contention as it is one of the most frequently used file operations in real-world applications (3, 25).

LocoFS uses a 30-second lease to cache entries on the client side. Although this improves lookup operations, it can also lead to stale data, which is undesirable (3).

Lastly, by reorganizing the traditional hierarchical directory structure into a flattened format, LocoFS reduces the complexity of metadata operations. This design minimizes

File System	Advantages	Drawbacks
TableFS	Optimized for writes using LSM	Large value fields
	Comprehensive tuning options with LevelDB	Costly (de)serialization
		Degraded lookup performance as LSM-tree grows

File System	Advantages	Drawbacks
IndexFS	Dual-component metadata store improves lookups	Complex load balancing for horizontal scaling
	Cache mechanisms avoid hot spots	Complex management due to horizontal scaling
	Scalable distributed design	

File System	Advantages	Drawbacks
LocoFS	Small value fields through metadata decoupling	Complex rename operations
	Flattened directory tree simplifies metadata operations	Some file operations require multiple get operations

Figure 3.10: Summary of Key Advantages and Drawbacks for our three Case Studies, TablesFS, IndexFS, and LocoFS

dependencies between directory and file metadata, enabling more independent and efficient handling (3).

In summary, despite some issues such as stale cache entries and intricate rename processes, LocoFS's novel method of handling metadata and its scalable framework make it ideal for high-performance computing settings. These environments can leverage this efficiency and simplified operational complexity when managing metadata-heavy tasks. This architecture delivers smart solutions in the domain of hybrid key-value-store file systems, laying a solid foundation for any practical application that demands high throughput and minimal latency in metadata operations.

3.5 Summary

In this chapter, we explored three different file system implementations that utilize key-value stores to store metadata in order to achieve performance gains with regard to metadata management. Our review has shown that many design options exist such as deploying multiple metadata servers, segregating metadata into smaller subsets, or flattening the file

system tree. For each design decision we analysed what the benefit is according to the authors. Subsequently, we listed some key advantages and drawbacks of the overall architecture. **chapter 3** summarises our findings.

Our analysis indicates that hybrid key-value store file systems present multiple design choices to tackle metadata management challenges. Each method has its own set of benefits and limitations. These designs emphasize the trade-offs among complexity, performance, and scalability. Selecting a design for a key-value store hybrid file system should be based on the specific needs of the application, highlighting the difficulty in creating a file system that fulfils all requirements. In addressing our research question **RQ1**, we determine that the existing design choices for hybrid key-value store file systems each provide unique benefits for efficient metadata handling. Nonetheless, these choices also introduce inherent complexities, whether in managing distributed systems, optimizing key-value stores, or dealing with the costs associated with (de)serialization of value fields. Therefore, it is difficult to identify the most suitable design choice.

4

System Design of vkFS

In this section we present the design of vkFS and deploys a hybrid file system storing metadata in a key-value store while using the underlying file system to store larger files. In this section we aim to provide a high level overview of the system and then gradually diving into more detailed design decisions.

4.1 vkFS Design Requirements

Prior to presenting our design we want to highlight the design requirements we set ourselves. Given the problem at hand we set the following Design Requirements (DR):

- **DR1**: Optimize Metadata Handling

  The main objective of the system should be to optimize the system for metadata-intensive tasks

- **DR2:** Utilize a Key-Value Store

  The design should fit into the realm of hybrid key-value store file systems. Therefore, the design needs to make use of a key-value store.

- **DR3:** Exploit Underlying File System

  The design should make use of very high-performance I/O operations of established file systems to store large file data.

4.2 Overview of vkFS System Architecture

This section provides a high-level overview of our file system design. **Figure 4.1** depicts all the essential components of our design.

Figure 4.1: vkFS Sytem Architecture

In this figure, we can observe arrows with three different colors. The green arrow indicates a bidirectional communication between the Application, in this case the Filebench Benchmark, the Fuse Driver, and vkFS. The arrow is bidirectional, since vkFS will return the appropriate result and writes it back to the FUSE Driver (covered in **subsection 3.2.2**) before its result is returned to the waiting process via the Virtual File System (VFS). Similarly, the blue arrow also indicates bidirectional communication. If vkFS receives a request to read or write from or to a file that exceeds the inline data threshold of 4KiB, the system performs the appropriate system call via the VFS to access the underlying file system ext4. This addresses **DR3**. Its response will cycle through the blue arrow followed by the green arrow back to the application process. Finally, the red arrows handle all metadata and requests for small files. vkFS makes the appropriate request to RocksDB to retrieve, put, or delete an entry that contains the metadata or inline file data. This addresses **DR2**. If the requested data is not in in-memory table, the system has to pass through the kernel to access the underlying storage hardware and read the contents from the LSM-tree. Its contents are then returned to the file system and a response can be formed for the waiting application process. The design as a whole addresses **DR1**.

4.3 Inode Allocation

Inode allocation plays an intrinsic part in any file system as it provides every entry with an inode number, which is the primary information to retrieve the entry. vkFS uses a single allocation mechanism to handle all inodes in the file tree starting at the root directory with special inode number 1. When a request is made to create a new file or directory, a new inode number is required. vkFS uses a simple mechanism that increments the current inode number after every creation of a file or directory. Additionally, it recycles inodes that belonged to entries that were requested to be deleted. vkFS allows for 2^{32-1} inodes to exist simultaneously.

4.4 vkFS Key-Value Pairs

In vkFS all metadata is stored in RocksDB, a persistent key-value store. RocksDB uses an LSM-tree as the underlying data structure to store key-value pairs. The details of this data structure are already covered in section **subsection 3.2.3**. The inherent design of an LSM-tree allows for fast updates and insertion. This write optimized performance becomes especially useful when inserting many new directories or files into the file system tree or when making small changes on metadata during modification operations such as “utimens”, “chmod”, or “chown”.

In a manner akin to LocoFS, we have chosen to separate dependencies between directory tree metadata and file metadata. Consequently, vkFS records one entry for each file and directory in the metadata store. The insertion key is composed of the inode number of the specific file. The corresponding value varies for files and directories. For a file entry in the metadata store, the value comprises five elements: a flag indicating that the value pertains to a file, the length of the file name, a flag indicating whether the data is stored in the underlying file system or as inline data in the metadata store, the inode itself, and the file name. Additionally, similar to TableFS, vkFS also stores small files in the metadata store by appending the data to the value field of a file entry. The threshold for this is 4KiB, meaning that file data below this threshold will be stored alongside the metadata, whereas larger files will be stored in the underlying ext4 file system.

The corresponding value field for directories holds values similar to those of files. The value of directories consists of a flag indicating that the value belongs to a directory, the length of the directory name, the inode, and the name of the directory. Additionally, the inode number of each entry in the directory is appended to the value field. The layout

vkFS File Key-Value Pair

Inode Number	File Flag	Filename Length	Inline Data Flag	Inode	Filename	Inline Data ...
Key	Value					

vkFS Directory Key-Value Pair

Inode Number	Dir Flag	Directory Name Length	Inode	Directory Name	Directory Entries ...
Key	Value				

Figure 4.2: vkFS Key-Value Pairs for Files and Directories

of both key-value pairs is shown in **Figure 4.2**. Note that the lengths of the individual blocks are not scaled.

5

Implementation of vkFS

In this section we will provide a detailed description of how we implemented vkFS. We will cover requirements, development environment, review the project hierarchy and some of its classes and their functionality. Finally, we will touch on some challenges that we faced during the implementation phase as well as the current limitations of our system.

5.1 Implementation Requirements

After finalizing our design reviewed in **chapter 4**, we commenced with the implementation phase. To begin with, we set out a number of implementation requirements that the system must fulfill. We define the following **Implementation Requirements** (IR):

- **IR1:** The system should initialize the file system and setup internal structures as required by FUSE.
- **IR2:** The system should be able to retrieve file metadata and perform modifications to it such as timestamps.
- **IR3:** The system should be able to create empty files and directories.
- **IR4:** The system should be able to read from files stored as inline data or in the underlying file system.
- **IR5:** The system should be able to write to files stored as inline data or in the underlying file system.
- **IR6:** The system should be able to delete files or directories.
- **IR7:** The system should be able to read the contents of a directory.

- **IR8:** The system should be able to truncate a file.
- **IR9:** The system should be able to modify timestamps in the metadata of a given file or directory.

5.2 Development Environment

In this section, we provide a brief summary of the development environment used for our project. The host machine was an Apple MacBook Pro with the following hardware specifications: an Apple M2 Pro processor, 16 GB of LPDDR5 memory, and a 1 TB SSD. To ensure compatibility and isolation, we deployed a virtual machine using Parallels Desktop. The virtual environment was configured to run Ubuntu 22.04 ARM 64 with 8 GB of memory. This configuration allowed us to maintain a consistent development environment.

Our programming language of choice was C++, utilizing the g++ compiler version 11.4.0 for code compilation. To efficiently handle the build process, we used CMake, which managed include paths, compiler flags, and the setup of custom benchmark runs. CMake also facilitated handling program arguments, making it a valuable tool in our development phase.

For version control, we used Git for local repository management and synchronized our changes with a remote repository on GitHub. We adhered to common conventions for repository setup and commit messages. Lastly, the development was conducted using Visual Studio Code (VSC) IDE, version 1.88.0, which provided a comprehensive environment for code editing and project management. This setup enabled us to maintain a high standard of code quality and project organization.

5.3 Project Hierarchy

During medium sized projects like vkFS, it is essential to maintain a clear project hierarchy and segregate files based on functionality and purpose. In addition, a clear project layout with good naming conventions makes it easier to locate specific files. The project hierarchy of vkFS is depicted in **Figure 5.1**.

The project includes four subdirectories. These include benchmark workloads further divided into data-intensive and metatdata-intensive, our header files, rocksdb as a submodule, and the source code. When following the installation guide of our system more empty directories will be created such as a directory to store the procuded binaries, the

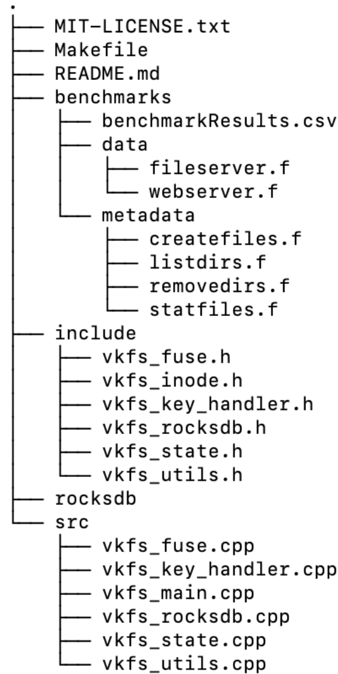

Figure 5.1: Project Hierarchy of vkFS

mount directory, the directory to store the metadata from RocksDB, and a directory to store large files. In our project structure, RocksDB is marked as a submodule using git. By doing this, we can assure compatibility as we linked the exact commit hash that we have been using for our development.

5.4 FUSE Operations

During our implementation, we made extensive use of the Filesystem in Userspace (FUSE) framework, specifically focusing on FUSE 3. The framework consists of three main packages. First, FUSE3 provides the core utility, allowing to mount the file system and managing the lifecycle of the file system within the user space. Second, libfuse3-3 contains the shared library including the core functionalities such as linking the FUSE libraries at runtime. Third, libfuse3-dev provides the headers and static libraries needed for compiling applications that use FUSE.

The FUSE library defines callback functions for each FUSE operation. In our main program we linked these to our own custom implementation of these operations. We did this by defining a FUSE Operation structure and changed the function pointer of the specific function definition to match the corresponding function implementation of

our system. This results in the FUSE library to use our own custom implementation when a FUSE operation is called. Understanding the entire FUSE workflow required some comprehensive testing. For instance, some file system requests result in multiple fuse operations to be triggered. If an empty file is created using the program "touch", fuse creates a series of operations to be sent. These include "getattr" to check if the file exists, "create" to create the file, "getattr" to check if the file exists now, "flush" to flush the contents to disk. Immediately after that, FUSE triggers more operations such as "utimens" to modify timestamps of that file, and also "getattr" to check if these changes have been made, and "flush" again to write these changes to disk. It is evident that this workflow takes time to fully grasp and was a challenge during the development process.

5.5 vkFS RocksDB Class

At several scopes of our implementation we needed access to the metadata store, or in other words, RocksDB. To start with, we passed the pointer for the database initialzed at the begining of our main program as a parameter to multiple functions throughout our program. However, as the complexity of the file system grew this became inefficent. At this point, we decided to deploy a Singleton Design Pattern to make sure different files and different scopes in our program have access to the same database pointer. In addition, it was also essential to make sure that there was only a single instance of the database. For these reasons, the Singleton Design Pattern fit well as it makes sure that only exactly one copy of the database pointer exists at any time. Furthermore, throughout the program multiple functions needed to perform basic database operations such as get, put, or delete. Therefore, we decided to create this separate class handling all operations made to the database while ensuring that only one instance exists.

5.6 Inode Allocation Handler

vkFS employs a dedicated class to allocate inode numbers during the creation of files and directories. The root directory is initialized with the well-known inode number 1 upon system start. Subsequent allocations are assigned new inode numbers in an incremental fashion. Inode numbers are represented as unsigned 32-bit integers, allowing for up to $2^{32} - 1$ total inode numbers to be managed simultaneously.

In addition to incremental assignment, vkFS also recycles inodes that are no longer in use due to file deletions or directory removals. When recycling an inode, the responsible

class inserts the inode number into a queue. If a new inode is requested, the system first checks this queue for available inode numbers to reuse before incrementing a global counter for the next inode number to be assigned.

When a new inode number is assigned, the class stores it in a hash map where the key is the full path of the entry, and the value is the inode number. This implementation enables fast retrieval of inode numbers for a given path and, subsequently, allows for quick access to the metadata store using the inode number.

The class responsible for assigning inode numbers uses two types of locking mechanisms to ensure thread safety. First, a shared lock is used whenever an entry is requested to be read, such as when the system checks if an entry already exists. Secondly, a unique lock is utilized when creating new entries or deleting existing ones. This locking mechanism is implemented using the mutex class.

5.7 File System State

vkFS uses a class to globally store information about the file system state. Upon entering the main program, the system declares an instance of this class to hold various pieces of information, such as the paths for the metadata, mount directory, and the directory used to store data in the underlying file system. Additionally, it stores a threshold value that determines whether data is stored inline or on the underlying file system.

Furthermore, this class contains a boolean flag and a pointer to the class responsible for inode allocation. The flag indicates whether the root directory has already been initialized. The pointer to this file system state class is passed as an argument to the main FUSE function, allowing FUSE operations to access this instance. The following code snippet illustrates this:

```
vkfsState fsState = static_cast<vkfsState>(fuse_get_context()->private_data);
```

Once this pointer is fetched, functions can access critical data about the file system state.

5.8 Metadata Headers

vkFS extensively utilizes a header file that defines the structures used to store metadata for files and directories. As described in **chapter 4**, vkFS stores metadata in the value field of a key-value store. This metadata is encapsulated within a header, defined using a struct, which includes the length of the file name, a boolean flag indicating whether the

data is stored as inline data, and the actual metadata. The metadata itself is represented using the struct stat from the Standard C Library. The structure for vkFS files is shown in the following code snippet.

```
struct VKFSFileMetaData {
    size_t filename_len;
    bool has_external_data;
    struct stat file_structure;
};
```

Upon file creation, a new structure is declared and initialized with the appropriate values. To write the data to the value field, vkFS uses another defined structure. This structure holds a char pointer and the corresponding size in bytes. Its declaration is depicted in the following code snippet.

```
struct VKFSHeaderSerialized {
    uint16_t size;
    char* data;
};
```

The size field is calculated as the sum of several components. First, a special flag, one byte in length, indicates that the value pertains to a file. Second, the size of the VKFSFileMetaData structure, as shown above, is included. Lastly, the length of the file name plus an additional byte for the null-terminating character `'\0'` is added. Thus, upon file creation, the size field of the VKFSHeaderSerialized structure is initialized to the sum of the flag, the size of the VKFSFileMetaData structure, the file name length, and an extra byte for the null-terminating character.

The system then requests the appropriate size of memory on the heap, and the char pointer of VKFSHeaderSerialized is set to the newly allocated memory returned by the new keyword. The system writes the data to this location using the memcpy function from the cstring header file. The contents of VKFSHeaderSerialized are then used as the value field for a given key and inserted into the metadata store. The process for directories is almost identical using different values as discussed in **chapter 4**.

5.9 Interaction with the Underlying File System

vkFS manages data exceeding a pre-defined threshold of 4KiB by storing it in the underlying ext4 file system. When the FUSE library receives requests to truncate or write data,

vkFS checks if the new file size surpasses this threshold. If so, several implementation techniques are employed to store the data in ext4.

First, vkFS determines the full path for the file within the ext4 system. This is achieved using a method inspired by the TableFS paper, which is illustrated in the following code snippet:

```
std::string getLocalFilePath(VKFSFS_KEY key,std::string datadir){
    std::string J = std::to_string(key / 1000);
    std::string I = std::to_string(key);
    std::string projectPath = getProjectRoot();
    std::string fullPath = projectPath + "/" + datadir + "/" + J + "/" + I;

    return fullPath;
}
```

This code maintains a directory structure where each directory contains up to 1,000 files, determined by their inode number (represented as `VKFS_KEY`). The datadir parameter specifies the directory within the project structure where these files are stored. To create a file, vkFS uses the "open" system call with the `O_CREAT` flag. Data is written to the file using the "pwrite" system call. Similarly, for read requests, vkFS employs the "read" system call to retrieve data from the ext4 file system.

5.10 Limitations of vkFS

The current version of vkFS has several limitations due to two primary reasons. First, the limited time frame for this project allowed for only a subset of functionalities to be implemented. Second, file systems are inherently complex and require extensive planning and phased implementation, which were beyond the scope of this work. The key **System Limitations** (SL) identified are as follows:

- **SL1: Lack of Failure Handling**

  The current version of vkFS does not include mechanisms to handle system crashes or data recovery from a Write-Ahead Log (WAL) or other recovery systems. Consequently, any data in the file system is lost upon system shutdown or crash.

- **SL2: Incomplete Thread Safety**

While some locking mechanisms have been implemented, the system's behavior under multi-threaded conditions remains undefined. Full thread safety has not been achieved at this stage.

- **SL3: Limited FUSE Operations Support**

  vkFS currently supports only a subset of FUSE operations, limiting its functionality and compatibility with various use cases.

- **SL4: Absence of Permission Checking**

  The system does not perform permission checks when files are opened, written to, or executed. This absence of security measures could lead to unauthorized access and operations.

- **SL5: No Support for Hard Links**

  Currently, vkFS does not support the creation of hard links between files, limiting its ability to create references to the same data from multiple locations.

6

Evaluation

In this section, we conduct a comprehensive evaluation of vkFS using both micro and macro benchmarks utilizing the Filebench benchmarking tool. Filebench provides predefined workloads designed to test specific aspects of a file system's performance, including metadata-intensive operations and data-intensive workloads that simulate real-world application workloads.

We employ micro benchmarks to test the performance of particular components of the system, focusing on file system operations that are heavily reliant on metadata. Detailed descriptions of these workloads and their methodologies will be presented in the subsequent sections of this chapter. The primary aim of these benchmarks is to assess how our design choices discussed in chapter 4 influence vkFS's ability to enhance performance for metadata-intensive tasks, thus directly addressing **RQ1**.

In addition to micro benchmarks, we also employ macro benchmarks to evaluate vkFS in real-world application workloads, such as those typical for file servers or web servers. These tests are meaningful in understanding how the system performs in scenarios that extend beyond the primary focus of our optimizations. Our objective is to show that vkFS can achieve satisfactory performance with conventional workloads commonly used to evaluate local file systems.

For a robust and meaningful comparison, all tests are performed on established conventional file systems, including ext4, btrfs, and xfs, as discussed in chapter 2. We describe each workload used in our tests, present the performance results, and offer a summary of our findings at the end of each section. The chapter concludes with a summary of all results and a discussion addressing the research questions posed at the outset.

6.1 Hardware Configurations

In this section, we detail the hardware and operating system configurations used for our experiments. All tests were consistently executed on the same physical machine to ensure comparability and reliability of the results. The hardware setup includes a 20-core Intel Xeon Silver 4210 CPU running at 2.20 GHz, with two sockets connected in NUMA mode, complemented by 256 GiB of RAM. The hardware storage is a Samsung NVMe (SAMSUNG MZQL2960HCJR-00A07) with 960GiB.

For the benchmarks, we utilized a virtual machine (VM) configured with the QEMU tool, employing only a portion of the host's available resources. The VM was set up with a QCOW2 formatted virtual disk image and allocated 2 GiB of RAM. To enhance performance, Kernel-based Virtual Machine (KVM) acceleration was enabled, providing the VM direct access to the host's hardware features. Moreover, the VM used the host's CPU configuration to ensure compatibility and performance closely aligned with the physical host. These configurations allowed us to create a controlled and consistent environment for our performance benchmarks.

6.2 Metadata-Intensive Micro Benchmarks

In this section, we assess the outcomes of the micro benchmarks we conducted. These benchmarks are chosen to evaluate specific components of a file system, focusing particularly on metadata management. Each subsection includes a description of the workload, a graph displaying our results, an outline of the results, and a summary of the findings. The reason for evaluating the following metadata-intensive tasks is to evaluate whether or not our design introduced in chapter 4 displays a performance benefit in comparison to the file systems explored in chapter 2. This helps us formulate a conclusion to both our **RQs**. If we can observe significant performance benefits throughout the next section, we can claim that our introduced design has advantages with regard to handling metadata-intensive tasks, thus addressing **RQ1**. At the same time, positive results will serve as a justification for our own design and its implementation in particular, thus also addressing **RQ2**. However, if we observe that vkFS performs significantly worse than the traditional file systems we can discuss whether this occurred as a result of our implementation or design.

6.2.1 Setup for Metadata-Intensive Micro Benchmarks

The configuration for all subsequent micro benchmarks follows a similar pattern. Each workload is executed five times to assess the consistency of the performance achieved. This repetition helps to ensure that anomalies do not significantly affect our evaluation. Additionally, each figure will include an error bar showing the standard deviation of our results, providing a clear indication of each system's consistency. Four different file systems will be tested with each workload, including the conventional file systems mentioned in chapter 2, namely ext4, xfs, and brfs, as well as our own system vkFS. The files systems will be represented along the x-axis. For each micro benchmark, we will report the average throughput along the y-axis in Kilo Operations Per Second (KOPS) to simplify large numbers. This implies that higher values along the y-axis are considered better. All results will be displayed using a bar chart, allowing for an easy interpretation of our findings.

6.2.2 Create Files

Our first micro benchmark tests the performance of creating empty files in a file system tree. The workload tasks the file system to create 100,000 empty files. This is the default number set by Filebench for this micro benchmark. At the start of the workload, Filebench prepares a file system tree consisting of only directories that are not part of the performance measurement. Once the file system tree is prepared, the benchmarking process commences, and 100,000 empty files are requested to be inserted while maintaining an average directory width of 100 files. The workload involves a significant amount of metadata operations, including updating directory entries and allocating inode structures. **Figure 6.1** shows our results.

While the conventional file systems btrfs and xfs performed consistently achieving 49.99 KOPS, ext4 performed 6.66% worse than xfs and btrfs achieving an average throughput of 46.66 KOPS. Additionally, a higher standard deviation can be observed for the results of ext4 in comparison to btrfs and xfs. Upon further inspection of the single runs, our data indicate that one run performed significantly worse than the other four, producing a high value for the standard deviation for ext4. The cause of this has not been verified. Our implementation vkFS performs significantly worse, averaging only 13,52 KOPS. However, the standard deviation indicates consistent performance across all five runs. To be precise vkFS performed 72.96% worse than both btrfs and xfs, and 71.02% worse than ext4.

Based on the micro benchmark results, it is evident that while btrfs and xfs consistently delivered average throughput at 49.99 KOPS, ext4 exhibited performance at only 46.66

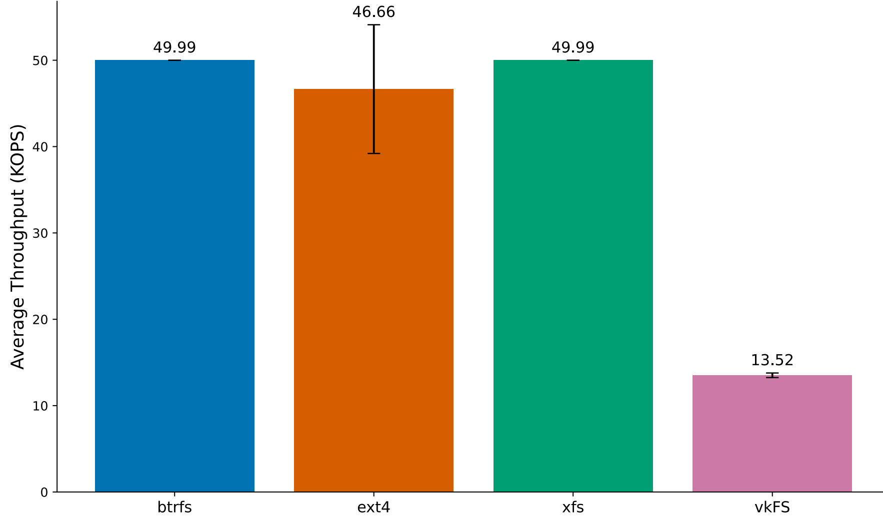

Figure 6.1: Average Throughput (KOPS) for File Creation Across Different File Systems

KOPS on average with notable variability in one of the runs. In contrast, our vkFS file system, although consistently performing across multiple runs, lagged significantly behind the others in terms of average throughput.

6.2.3 Stat Files

Our second micro benchmark involved retrieving metadata from existing files. The Filebench workload first prepares a file system tree with 100,000 empty files, keeping an average directory width of 100 entries. Once completed, the benchmark starts by executing "getattr" operations on the files for 60 seconds. These are the default settings set by Filebench for this micro benchmark. During the getattr operation, the metadata is accessed of each file, and the modification and access time is set to the current time. The updated metadata is then returned to the application. This benchmark is especially important to evaluate as the getattr operation is one of the most common file system operations when using FUSE (25). Here, we can provide meaningful data on how well the system accesses the metadata, performs the modifications, and stores the updated metadata. **Figure 6.2** shows our results.

The **Figure 6.2** illustrates that conventional file systems consistently achieve nearly 60 KOPS with a small standard deviation. Ext4's performance is slightly less consistent than that of btrfs and xfs. In contrast, our vkFS file system only achieves an average throughput of 28.97 KOPS. The standard deviation also indicates greater inconsistencies across runs

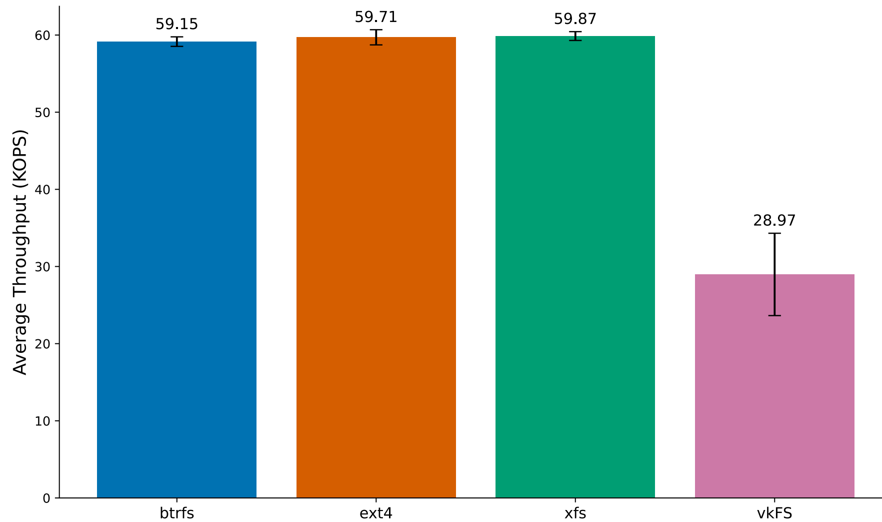

Figure 6.2: Average Throughput (KOPS) for Retrieving Metadata Across Different File Systems

compared to the other file systems.

The results from the stat files benchmark indicate that conventional file systems like btrfs and xfs handle metadata retrieval operations consistently achieving around 60 KOPS with standard deviations ranging from 0.57 KOPS to 0.98 KOPS. On the other hand, vkFS showed a standard deviation of 5.33 KOPS indicating high variation in comparison to the other file systems. In addition, the performance of vkFS is significantly lower than any other of the other file systems.

6.2.4 Remove Directories

The third micro benchmark tasks the file system to delete directories. Before performance measurement begins, Filebench creates a large file tree with an average of 100 empty files or subdirectories per directory. The benchmark then issues 100,000 requests to delete directories. When removing a directory, the corresponding entry in the parent directory must be deleted. This implies that the metadata of the parent directory need to be retrieved, modified, and written back to where it is stored. The storage location for this metadata is contingent upon the specific file system in use. Once the metadata of the parent directory has been updated, the contents of the directory to be deleted are recursively removed, including files and subdirectories. Since each directory contains an average of 100 entries the workload is considered to be the same for each individual run. **Figure 6.3** shows our results.

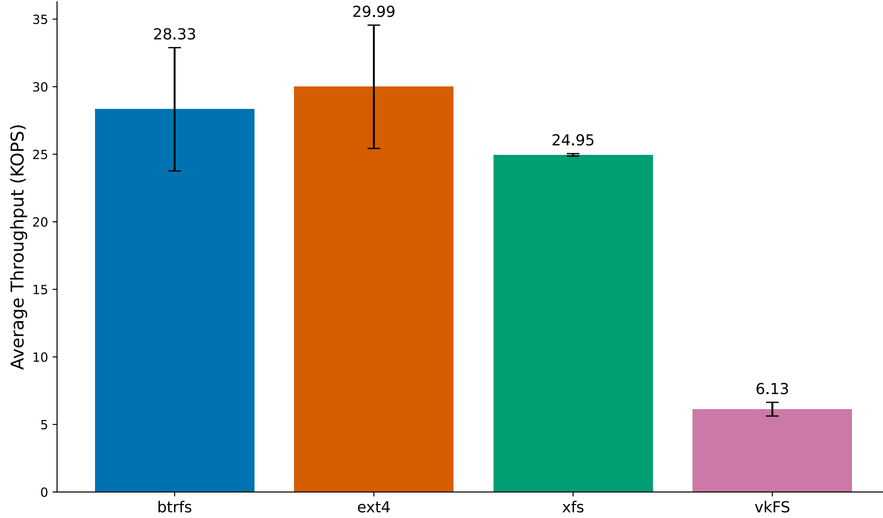

Figure 6.3: Average Throughput (KOPS) for Deleting Directories Across Different File Systems

Figures 6.3 shows that ext4 achieves the highest average throughput with 29.99 KOPS. Btrfs performs similar to ext4 but both file systems produced a standard deviation of 4.56 KOPS. Both of their lowest run is comparable to the average run of xfs. Our data reports that btrfs achieves 28.33 KOPS on average across five runs, while xfs only achieved an average throughput of 24.95 KOPS. However, across five runs our data indicates that xfs achieved the highest consistency during this benchmark with a standard deviation of 0.09 KOPS. The vkFS file system performs poorly in comparison to the other file systems. vkFS achieved 6.13 KOPS on average. However, its standard deviation reports 0.5 KOPS indicating less variability in comparisons to btrfs, and ext4 producing consistent results.

The results from this benchmark indicate that among the conventional file systems, ext4 and btrfs achieve the highest average operations per second, with ext4 slightly outperforming btrfs. However, both exhibit variability in their performance. In contrast, vkFS consistently performs at a much lower rate but shows less variability compared to btrfs and ext4, suggesting that while vkFS is slower, its performance is more predictable.

6.2.5 List Directories

The last micro benchmark we tested is designed to evaluate the performance of listing directories within a file system. Filebench pre-allocates a large fileset of 100,000 files using an average directory width of 5. The benchmark lists directories for a total of 60 seconds. During one operation, the file system checks the metadata of the directory and lists all

entries and their metadata. This operation can be compared to running a `ls -l` command in the terminal. During this workload, directory entries will be retrieved in a sequential manner. This specific operation is key to performing well on this benchmark. **Figure 6.4** shows our results.

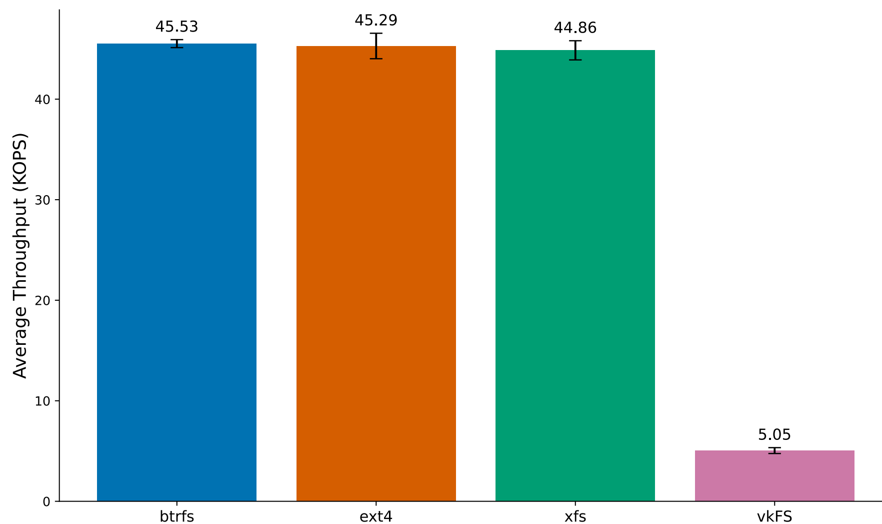

Figure 6.4: Average Throughput (KOPS) for Listing Directory Contents Across Different File Systems

The figure shows similar performance between conventional file systems. Btrfs performed best with 45.53 KOPS average throughput and a low standard deviation of 0.39 KOPS. Ext4 and xfs produced a slightly higher standard deviation with 1.26 KOPS and 0.95 KOPS respectively, while still performing similar as btrfs. However, vkFS performs poorly compared to the other file systems. Here we can observe a significant drop in performance as vkFS only achieved 5.05 KOPS. This is a performance drop of 89.34% in comparison to the next best performing file system xfs. Although performance is poor, the file system produces consistent results throughout the five runs reporting a standard deviation of 0.29 KOPS.

The directory listing benchmark assesses the efficiency of directory listings by pre-allocating a set of 100,000 files and measuring KOPS over a 60-second period. The findings, shown in **Figure 6.4**, reveal that btrfs, ext4, and xfs perform similarly, with btrfs leading at 45,53 KOPS and all show low standard deviations. In contrast, vkFS falls significantly behind, achieving only 5.05 KOPS average throughput although it shows consistent results across the tests.

6.2.6 Micro Benchmarks Summary

In the above section, we explored the results of various metadata intensive workloads and measured their average KOPS results and compared it among the different file systems. Evidently, all conventional file systems produced higher average throughput than our system vkFS. vkFS performs poorer on all tests, showcasing consistent runs, nonetheless. These micro benchmarks are important to conduct in order to draw conclusions about the performance of a file system with respect to metadata intensive optimization techniques, and therefore addressing our **RQs**. **Figure 6.5** summaries our micro benchmark results.

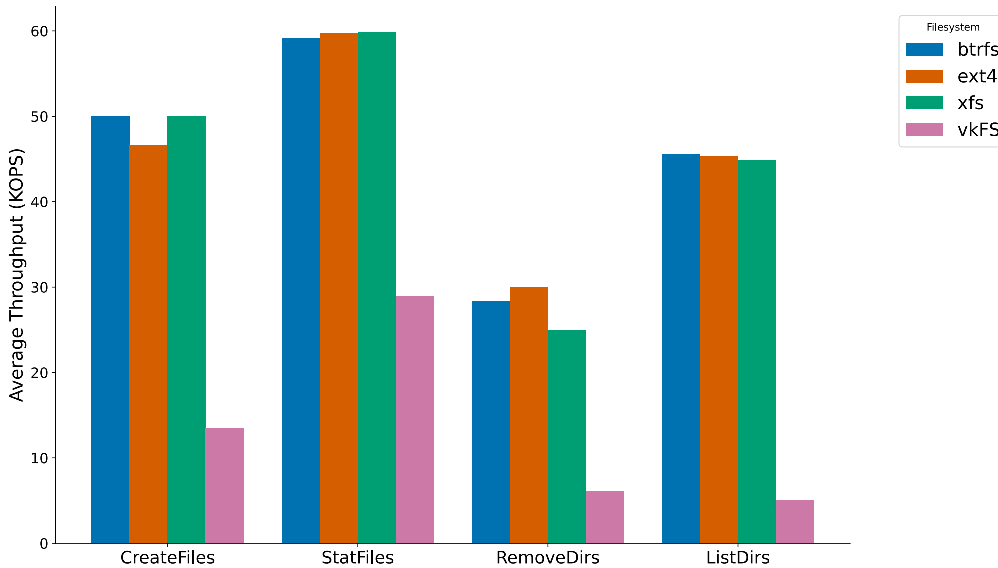

Figure 6.5: Summary of Metadata-Intensive Micro Benchmark Results

The results show that while btrfs, ext4 and xfs consistently excel in these operations, vkFS lags significantly, particularly when listing directories.

6.3 Data-Intensive Macro Benchmarks

In this section, we explore two default macro benchmarks from the Filebench tool. These workloads give insight into how a system performs under real-world application requirements. For example, we run a file server, measure the performance of each file system, and discuss how they handle the workload. We chose to run a file server and a webserver as these are common applications to test a file systems performance when tasked with a real-worl workload (1). This section serves to identify how our system performs when it is

assigned workloads for which it has not been optimized. We explore two different workloads in total, describing each result, followed by a summary of all macro benchmarks.

6.3.1 File Server

The file server workload is our first macro benchmark and simulates the typical operations of a file server environment. This includes a mix of access patterns common for this environment. In particular, this workload combines operations such as creating files, opening files, writing to files, reading from files, opening files, closing files, appending data to a file, deleting a file, and retrieving metadata from a file. We run the workload a total of five times for a duration of 60 seconds and report the average throughput in KOPS. We repeat this process for every file system discussed in this evaluation. **Figure 6.6** shows our results.

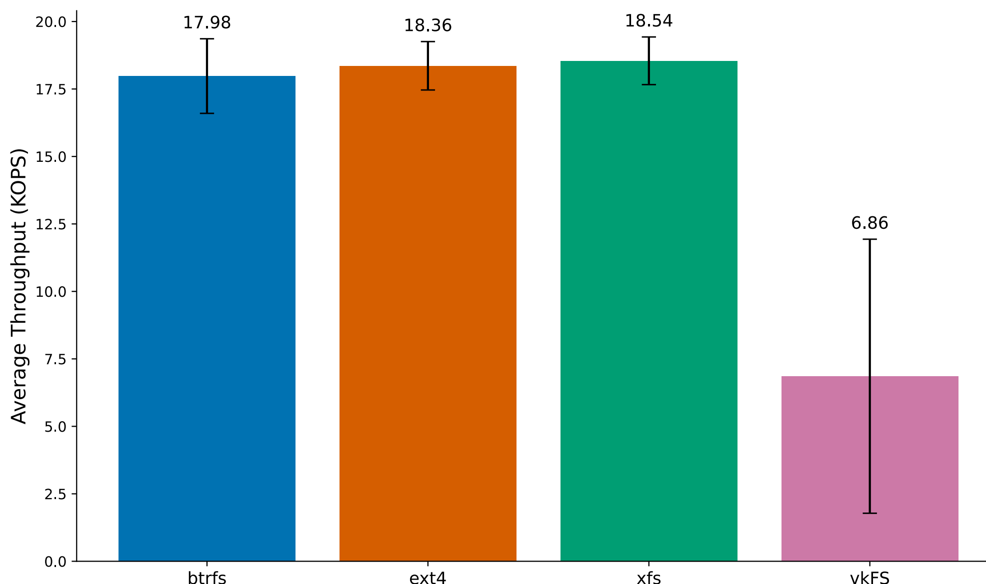

Figure 6.6: Average Throughput (KPOS) for Macro Benchmark File Server Across Different File Systems

Our results show that conventional file systems perform almost identically across all runs on average. Meanwhile, our vkFS system does not achieve similar performance results. While xfs achieves 18.54 KOPS average throughput, our system vkFS only achieves 6.86 KOPS. The standard deviation for vkFS also indicates inconsistent results.

The results of our file server macro benchmark reveal that conventional file systems perform consistently with high average throughput in comparison to our system. vkFS

shows significantly lower performance and greater variability, highlighting its struggle to match the performance of established systems.

6.3.2 Web Server

Our last macro benchmark simulates a web server environment. This data-intensive workload involves a mix of operations that include opening, writing, reading, and closing files. The primary focus of this benchmark is to evaluate the capabilities of the file system to handle these frequent and varied file read and write operations. The data written to the files is 4KiB on average. We run this workload five times for each file system for a duration of 60 seconds and measure average throughput in KOPS. **Figure 6.7** shows our results.

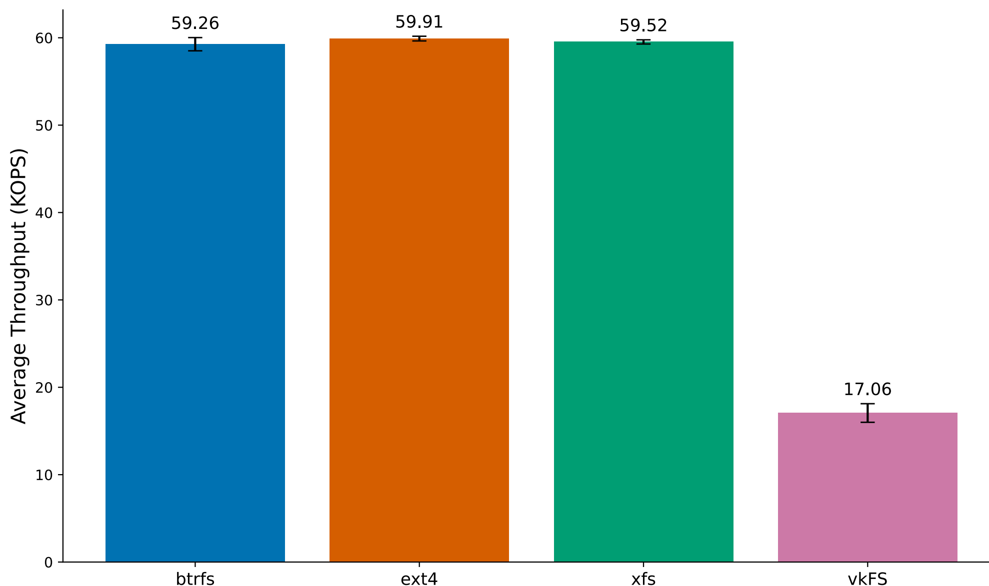

Figure 6.7: Average Throughput (KPOS) for Macro Benchmark Web Server Across Different File Systems

Although all conventional file systems perform similar, ext4 performed best at an average of 59.91 KOPS. In contrast, our system performed the worst at an average of 17.06 KOPS. During this benchmark, standard deviation across all file systems was low, indicating consistent performance.

6.3.3 Macro Benchmarks Summary

In this section, we provide a summary of the data-intensive macro benchmarks. We run two different workloads that simulate different real-world application requirements such

as a file server and a web server. While the micro benchmarks tested the file systems at one specific operations, the macro benchmarks included a multitude of operations, shifting the focus to realistic workloads and access patterns. It is useful to look at both types of benchmarks to get an idea of how a system performs in a very specific area such as metadata handling, but also to see how this affects overall performance when dealing with real-world examples. In addition this evaluation is useful to address potential downsides of our design and hence addressing **RQ1**. **Figure 6.8** summarizes the macro benchmarks in one figure.

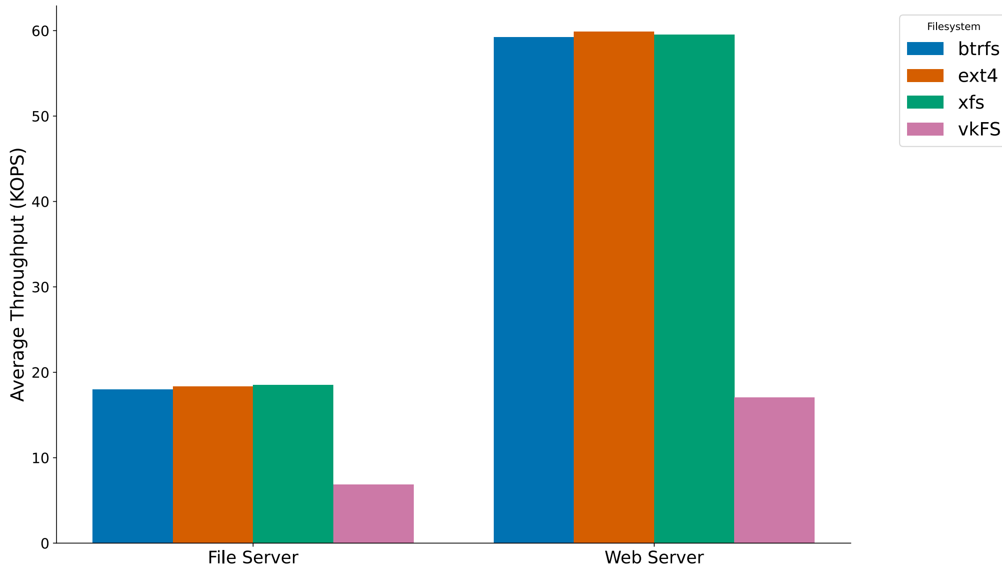

Figure 6.8: Summary of Macro Benchmarks

6.4 Evaluation Discussion

In this section, we provide a brief discussion of our benchmarking results. From the summaries on both micro and macro benchmark results we can observe that our system vkFS perfromed the worst overall. The reasons for this have not been verified by means of experimentation. However, there are some aspects to consider when interpreting the results. First, the FUSE module introduces a significant messaging overhead between user space and kernel as discussed in **subsection 3.2.2**. Moreover, our implementation introduces additional overhead. However, this has also not been verified by experimentation. For example, the benchmark Stat Files in **subsection 6.2.3** tests the performance of the file system to retrieve metadata, make small modifications to it, return it, and subsequently

store the modified metadata. In our implementation, we fetch the metadata from the key-value store, make the adjustments to the metadata structure, and then initiate the process of mirroring these changes in the key-value store. However, since the value field stores more than just the metadata, we initiate another get request to the key-value store to retrieve additional metadata such as file name, but also its potential inline data. We write the modified metadata into the corresponding value field and insert it back into the key value store. This can lead to inefficencies, as the get request is called twice, where other implementation approaches can reduce this complexity and handle the whole request with a single get and put request. This complexity in addition with the FUSE overhead could cause vkFS to perform poor in contrast to other file systems in this specific benchmark (subsection 6.2.3).

Moreover, the benchmarks have revealed that the List Directories benchmark performed particularly poorly. Although not verified, we hypothesize that our design to store directory entries causes inefficiencies. Large value fields are known to throttle the performance of key value stores, and by adding additional fields to the value field, we allow this performance drop to occur (3). Especially when reading directory entries, our implementation lacks efficiency as we perform two get operations for each directory entry, one to retrieve the file name and one to retrieve the metadata structure. This could be re-factored to aim for minimizing get requests.

Overall, this evaluation could benefit from approximating the overhead caused by FUSE. Then, a further analysis is required to determine the overhead caused by our implementation. This would allow for a more precise analysis of the performance of our design. For example, two approaches could be considered to approximate the performance of our system given the FUSE overhead. First, a FUSE passthrough file system could be developed that forwards all FUSE operations to the underlying file system using system calls. This way the FUSE overhead is included in the performance evaluation for the conventional file systems. Second, a library version of vkFS could be considered. In this version, the benchmarking process occurs inside the application omitting the overhead of communication between benchmark application, VFS, and FUSE driver.

With this in mind, future approaches could be utilized to gauge the performance of our system in more comparable way. However, with our current implementation is is difficult to claim if our design improved the performance of metadata-intensive workloads.

7

Conclusion

In this work, we have outlined our process to develop our own hybrid key-value store file system to improve metadata handling and achieve performance benefits when tasked with metadata-intensive operations. We detailed key design decisions, such as the use of the LSM-tree, storing small files alongside metadata, and allocating inode numbers based on full paths. We proceeded by providing a brief review of key implementation aspects, including different classes, the development environment, header files, and the interaction with the underlying file system.

We then carefully selected multiple benchmark workloads, divided into metadata-intensive and data-intensive tasks, to test the performance of our system against established standards like Ext4, XFS, and Btrfs. Our evaluation revealed that the expected performance benefits were not observed. However, we provided a discussion on the factors that may have influenced the performance of our system. Although these influences have not been verified by quantitative data, we offered a hypothesis explaining why the expected performance was not achieved. This array of methodologies directly addresses our **RQ2**. In this question, we asked how we can design and implement a hybrid key-value store file system. We are confident that **chapter 4** and **chapter 5** provide comprehensive answers to this question.

Additionally, we explored what design options are available in a hybrid key-value store file system to optimize for metadata-intensive tasks and discussed their advantages and drawbacks. This exploration represents our **RQ1**. **chapter 3**, we provided a comprehensive overview of three highly regarded academic papers focusing on metadata handling using a hybrid key-value store design. We analyzed each design individually, outlining the intentions behind each design decision from the authors' perspectives. Subsequently, we outlined the benefits of the designs as a whole, as well as some drawbacks introduced by

specific designs. Lastly, we summarized the key takeaways and presented them in **Figure 3.10**.

Our literature review revealed multiple design options, most notably horizontal scaling, a flattened directory tree, and the decoupling of metadata. Our analysis showed that these design choices offer several benefits, such as improved lookup performance, reduced overhead during serialization, and effective scaling using a distributed architecture. However, we also identified some drawbacks, such as large value fields, complex operations like renaming, and the challenges of managing horizontal scaling. These points apply to all the designs collectively rather than a single one. Our work shows that many design options exist and claiming that one is superior over the other is challenging since every design comes with some drawbacks as exemplified in **Figure 3.10**(**RQ1**). For each use case, one has to consider which drawbacks are acceptable in relation to performance. Furthermore, our work has shown that designing and implementing a file system remains challenging (**RQ2**). This underlines the necessity for continuous research in the domain of file systems ensuring that digital infrastructure remains robust and future-proof.

References

[1] Kai Ren and Garth Gibson. **{TABLEFS}: Enhancing Metadata {Efficiency} in the Local File System**. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 145–156, 2013. iii, 2, 7, 14, 18, 20, 26, 54

[2] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. **IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion**. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 237–248, 2014. iii, 2, 21, 26

[3] Siyang Li, Youyou Lu, Jiwu Shu, Yang Hu, and Tao Li. **LocoFS: A loosely-coupled metadata service for distributed file systems**. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2017. iii, 20, 26, 27, 31, 32, 58

[4] Göbel T, Türr J, and Baier H. **Revisiting Data Hiding Techniques for Apple File System**. *ARES '19: Proceedings of the 14th International Conference on Availability, Reliability and Security*, August 2019. 1

[5] HJ. Kim and JS. Kim. **Tuning the Ext4 Filesystem Performance for Android-Based Smartphones**. *Sambath, S., Zhu, E. (eds) Frontiers in Computer Education. Advances in Intelligent and Soft Computing, vol 133*, 2012. 1

[6] S. Ghemawat, H. Gobioff, and S.-T. Leung. **The Google file system**. *Proc. 19th ACM Symp. Operating Syst. Princ.*, 2003. 1

[7] Ohad Rodeh, Josef Bacik, and Chris Mason. **BTRFS: The Linux B-tree filesystem**. *ACM Transactions on Storage 9, 3*, 2013. 1

[8] et al. McKusick, Marshall K. **A fast file system for UNIX.** *ACM Transactions on Computer Systems (TOCS) 2.3*, pages 81–197, 1984. 1

[9] Rosenblum Mendel and John K. Ousterhout. **The design and implementation of a log-structured file system.** *ACM Transactions on Computer Systems (TOCS) 10.1*, 1992. 1

[10] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. **Scale and performance in a filesystem semi-microkernel**. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 819–835, 2021. 1

[11] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. **The zettabyte file system**. In *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, **215**, 2003. 1

[12] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. **Performance analysis of NVMe SSDs and their implication on real world databases**. In *Proceedings of the 8th ACM International Systems and Storage Conference*, pages 1–11, 2015. 1

[13] Lanyue Lu, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Shan Lu. **A study of Linux file system evolution**. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 31–44, 2013. 2

[14] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. **Scalability in the XFS File System.** In *USENIX Annual Technical Conference*, **15**, 1996. 2

[15] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. **The hadoop distributed file system**. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010. 2

[16] Hao Dai, Yang Wang, Kenneth B. Kent, Lingfang Zeng, and Chengzhong Xu. **The State of the Art of Metadata Managements in Large-Scale Distributed File Systems — Scalability, Performance and Availability**. *IEEE Transactions on Parallel and Distributed Systems*, **33**(12):3850–3869, 2022. 2

[17] Ric Wheeler. **One billion files: pushing scalability limits of linux filesystem**. *Linux Foudation Events*, 2010. 2

[18] Mendel Rosenblum and John K Ousterhout. **The design and implementation of a log-structured file system**. *ACM Transactions on Computer Systems (TOCS)*, **10**(1):26–52, 1992. 2

[19] Dutch T Meyer and William J Bolosky. **A study of practical deduplication**. *ACM Transactions on Storage (ToS)*, **7**(4):1–20, 2012. 2

[20] 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*, **2**, pages 21–33. Citeseer, 2007. 7

[21] Darrick Wong. **XFS AsciiDoc Documentation tree**. 11

[22] Rodeh Ohad, Bacik Josef, and Mason Chris. **BTRFS: The Linux B-tree filesystem**. *ACM Transactions on Storage (TOS)*, **9**(3):1–32, 2013. 11

[23] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, et al. **Learning key-value store design**. *arXiv preprint arXiv:1907.05443*, 2019. 13

[24] Sanjay Ghemawat and Jeff Dean. **Google/leveldb: Leveldb is a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values.** 14

[25] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. **To {FUSE} or not to {FUSE}: Performance of {User-Space} file systems**. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 59–72, 2017. 15, 16, 20, 31, 50

[26] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. **Wisckey: Separating keys from values in ssd-conscious storage**. *ACM Transactions On Storage (TOS)*, **13**(1):1–28, 2017. 16

[27] Krijn Doekemeijer. *TropoDB: Design, Implementation and Evaluation of an Optimised KV-Store for NVMe Zoned Namespace Devices*. PhD thesis, Universiteit van Amsterdam, 2022. 17

[28] Google. **LevelDB Documentation**. https://github.com/google/leveldb/blob/main/doc/index.md, 2021. Accessed: 2024-06-16. 20

[29] Qin Xin, Ethan L Miller, and SJ Thomas JE Schwarz. **Evaluation of distributed recovery in large-scale storage systems**. In *Proceedings. 13th IEEE International Symposium on High performance Distributed Computing, 2004.*, pages 172–181. IEEE, 2004. 26

Appendix A

Reproducibility

A.1 Abstract

This artifact descriptions provides an overview on how to access our artifact vkFS , run the file system, and reproduce the results we have obtained in our evaluation section. We provide information to any dependencies used as well as guides on how to install them correctly.

A.2 Artifact check-list (meta-information)

- **Program: vkFS (https://github.com/Vincent881909/vkfs)**
- **Compilation: C++ 17, g++ 11.4.0, CMake 4.3**
- **Benchmark Tool: Filebench (https://github.com/filebench/filebench/archive/refs/tags/1.4.9.1.**
- **How much time is needed to prepare workflow (approximately)?: 1h**
- **How much time is needed to complete experiments (approximately)?: 4h**
- **Publicly available?: Source code publicly available**
- **Code licenses (if publicly available)?: MIT LICENSE**

A.3 Description

A.3.1 How to access

In order to access our artifact please visit GitHub to locate our remote repository. The artifcat can be accessed here: https://github.com/Vincent881909/vkfs

A.3.2 Software Dependencies

The following software dependencies are required for this project:

- `g++`
- `build-essential`
- `fuse3`
- `libfuse3-3`
- `libfuse3-dev`
- `zlib1g-dev`
- `libbz2-dev`
- `liblz4-dev`
- `libsnappy-dev`
- `libzstd-dev`
- `CMake`
- `bison (For Benchmarking only)`
- `flex (For Benchmarking only)`
- `libtool (For Benchmarking only)`
- `automake (For Benchmarking only)`
- `Filebench (For Benchmarking only)`

A.3.3 Hardware dependencies

The evaluation section was performed on the following hardware:

A.4 Hardware Host

- **CPU**: 20-core Intel Xeon Silver 4210, 2.20 GHz, NUMA mode, 2 sockets.
- **RAM**: 256 GiB.
- **Storage**: Samsung NVMe (960 GiB, SAMSUNG MZQL2960HCJR-00A07).

A.5 Virtual Machine

- **VM Tool**: QEMU.
- **Resource Allocation**: 2 GiB RAM, QCOW2 virtual disk.
- **Performance Boost**: KVM acceleration for direct hardware access.
- **CPU Configuration**: Aligned with the host's CPU for compatibility and performance.

A.6 Installation

In order to install our program vkFS please follow the subsequent steps.

1. Clone the repository from GitHub

```
$ git clone https://github.com/Vincent881909/vkfs.git
$ cd vkfs
```

2. Install all dependencies with the following command

```
$ sudo apt-get update
$ sudo apt-get install -y \
  g++ \
  build-essential \
  fuse3 \
  libfuse3-3 \
  libfuse3-dev \
  zlib1g-dev \
  libbz2-dev \
  liblz4-dev \
  libsnappy-dev \
  libzstd-dev \
  bison \
  flex \
  libtool \
  automake
```

3. Install CMake from source

```
$ wget https://ftp.gnu.org/gnu/make/make-4.3.tar.gz
$ tar -xzvf make-4.3.tar.gz
$ cd make-4.3
$ ./configure
$ make
$ sudo make install
```

4. Initialize Git Submodule

```
$ git submodule update --init --recursive
```

5. Install RocksDB

```
$ cd rocksdb/
$ make static_lib
$ cd ../
```

6. Configure FUSE

   By default our system uses the `-oallow_other` flag to allow other users that did not mount the file system to interact with it. This is required to run the benchmarks and is also enabled by default in our program. Run the following command to open the file "/etc/fuse.conf".

```
$ sudo nano /etc/fuse.conf
```

   Then remove the "#" before `user_allow_other`. This ensures that the `-oallow_other` flag can be used.

7. Prepare Directories

   Our system assumes the following directories exist and will use them for storing binaries, metadata, local files, and mounting the file system. Create these with the following command:

```
$ mkdir bin mntdir metadir datadir
```

8. Compile the Program

```
$ make
```

9. Run the Program

```
$ make run
```

Alternatively, you can run the program in debug mode. This will print the specific operations called and their arguments. You can run the program in debug mode with the following command:

```
$ make run_debug
```

In order to run the benchmark workloads provided by Filebench we need to install Filebench.

1. Install Filebench

```
$ wget https://github.com/filebench/filebench/archive/refs/tags/1.4.9.1.tar.gz
$ tar -xzvf 1.4.9.1.tar.gz
$ cd filebench-1.4.9.1
$ libtoolize
$ aclocal
$ autoheader
$ automake --add-missing
$ autoconf
$ ./configure
$ make
$ sudo make install
$ cd ..
$ rm -rf 1.4.9.1.tar.gz
```

2. Disable Virtualization

   Filebench requires virtualization to be turned off in order to properly run the workloads. Run the following command to disable virtualization:

```
$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

A.7 Experiment workflow

In this section we describe how to reproduce our evaluation. After proper installation of our program and Filebench we can commence with the benchmarking process. First run the file system with the following command:

```
$ make run
```

Then, in a new terminal window, navigate to the "vkfs/benchmarks" directory. Here, you have the choice between metadata or data benchmarks. Details of the difference are established in **chapter 6**. You can inspect the different workloads for each benchmark. To run a specific benchmark use the following command. For instance, to run the "createfiles.f" workload we can use the following command:

```
$ sudo filebench -f createfiles.f
```

This will start the "createfiles.f" workload on our system vkFS.

Alternatively, we have provided quick access to all benchmarks using CMake. If you direct to the "vkfs" directory you can run all micro benchmarks using the following command:

```
$ make micro_benchmarks
```

Similarly, you can run the macro benchmarks using:

```
$ make macro_benchmarks
```

In each case, this command will run all micro or macro benchmarks in sequence.

A.7.1 Running the Benchmarks on Conventional File Systems

In order to run the benchmarks on ext4, xfs, or btrfs we have to follow a couple of steps. First we need to make sure we have the required packages to create the specified file systems. For that we need:

```
$ sudo apt install e2fsprogs xfsprogs btrfs-progs util-linux
```

Then we need to create disk images at a size of 10GB. To do this run:

```
$ dd if=/dev/zero of=ext4.img bs=1G count=10
$ dd if=/dev/zero of=xfs.img bs=1G count=10
$ dd if=/dev/zero of=btrfs.img bs=1G count=10
```

Then we need to format these images to the specified file systems like so:

```
$ sudo mkfs.ext4 ext4.img
$ sudo mkfs.xfs xfs.img
$ sudo mkfs.btrfs btrfs.img
```

Create mount points for each file system:

```
$ sudo mkdir -p /mnt/ext4
$ sudo mkdir -p /mnt/xfs
$ sudo mkdir -p /mnt/btrfs
```

And lastly, mount the file systems.

```
$ sudo mount -o loop ext4.img /mnt/ext4
$ sudo mount -o loop xfs.img /mnt/xfs
$ sudo mount -o loop btrfs.img /mnt/btrfs
```

To run the benchmarks, we need to navigate to the single Filebench files located at either "vkfs/benchmarks/metadata" and "vkfs/benchmarks/data". For each file we need to adjust the directory to the previously specified mount points. For example, for xfs this would be "/mnt/xfs". We can do this altering the following line:

```
$ set $dir=MOUNT_DIRECTORY
```

Replace MOUNT_DIRECTORY with the directory that is mounted for the individual fie system. To execute the benchmarks, navigate back to the "vkfs" directory and run the following to execute all workloads:

```
$ make micro_benchmarks && make macro_benchmarks
```

This will run all benchmarks in sequence.

A.8 Evaluation and expected results

The above section outlines how we prepared our evalutation section. Once everything has been set up we started with the actual benchmrarking process. First, we tested the three conventional file systems ext4, btrfs, and xfs. We followed the instructions above and then run the following command five times:

```
$ make micro_benchmarks && make macro_benchmarks
```

This will run all our tests once. We repeated this five times. We manually added the results to a .csv file. This file can be inspected at "vkfs/benchmarks/benchmarkResults.csv". We repeated this for all file systems.

A.8.1 Expected Results

We expected our system vkFS achieve higher average throughput than the conventional file systems. This was the main objective of developing vkFS. In contrast, we expected that the conventional file systems would yield varied performance results, with disparities in throughput. This is due to their different designs, explored in **chapter 2**. However, the results for the conventional file systems ended up being very close to each other. Perhaps, the scale at which we run the workloads was not sufficient to observe disparities. More concrete, it would have been beneficial to run the workloads at large scale for example instead of creating 100,000 files, create 1,000,000 files. Moreover, our system did not perform as expected. However, as mentioned in our **chapter 6** we hypothesize that this was due to the large overhead caused by FUSE. However, this has not been verified.