A User-Space File-system based on Key-Value stores

Afshin Sabahi Khosroshahi

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2019

Abstract

Recent storage device hardware trends enables a new approach to the design of file systems. Advances in Non-Volatile Memory(NVM) technologies, are blurring the line between storage and memory. These NVM devices support load/store operations and provide new APIs. Emerging storage technologies such as KV SSDs [10] opens a new path way to how file systems can be implemented. User space drivers for these NVMe devices allows applications to directly access without the kernel overhead [3]. This paper introduces the design and implementation of a user space POSIX compliant file system using key-value stores. The system is designed to provide efficient and scalable access to both large and small files.

Decleration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

–Afshin Sabahi Khosroshahi

Acknowledgements

I would like to thank my parents, who have supported me throughout entire process, by keeping me harmonious and providing their invaluable guidance and kindness. I will be grateful forever for your love.
I would like to also express my gratitude to my supervisor Pramod Bhatotia for the useful comments, remarks through the learning process of this project.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Users with ever-increasing data storage requirements use two solutions: file systems and databases. File systems are designed to store large files such as movies and VM images. Users expect to access files sequentially at close to disk throughput, requiring file data to be stored on disk with minimal fragmentation. Databases, how-ever, store large numbers of comparatively small tuples, all indexed so that lookups and scans are efficient. Additionally, transactional semantics are a must for applications to maintain consistent relationships between the large numbers of tuples that databases support.Today, users must choose between file systems and databases. If program data does not precisely fit one of these two options — or requires both — then developers have to carefully consider the workload to decide which trade-off will hurt performance the least. Applications that manage both kinds of data can do no better than to manage data in two separate stores. For example, training deep NNs with millions of images, would require to cache large data sets into RAM for fast access. RAM is volatile and non persistent. An approach to provide these specialised complex workloads with direct access to a fast persistent memory storage device such as NVMe, can accelerate and simplify their code complexity. [1]

The project overview and design goals are described in section 3.

1.2 Key contributions

This project was initially titled as implementing a user space file system using Intel SPDK[3]. After much research and discussing possible approaches to tackling the problem with my supervisor, I decided on implementing a file system based on key value stores. These are the key contributions I have done to finish this project:

- Research emerging storage devices, such as NVMe[13].
- Research emerging new software APIs, such as SPDK[3], PMDK[2]

- Research possible storage engines such as Key value stores which are optimised for persistent memory devices. e.g. RocksDB[5], LevelDB[6], Samsung KVSSD[10].
- Learn building a C++ shared library, CMake build system.
- Learn and research POSIX[7] file API standards and specifications.
- Learn, understand user space file systems and their limitations.
- Design a file system layout which is efficient for a key value store.
- Develop solutions for limitations enforced by key value storage engines, e.g. storing binary data, appending to stored data and serialisation.
- Develop the file system, while complying with POSIX standard requirements.
- Evaluate the overhead produced by my file system against native key value store performance.

Chapter 2

Background

2.1 Essential Background on Persistent memory

The terms persistent memory and storage class memory are synonymous, describing media with byte-addressable, load/store memory access, but with the persistence properties of storage. The article Persistent Memory Programming [9] explains in details what are persistent memories and describes available software APIs to access these storage devices.

2.2 Essential background on User space NVMe drivers

User space NVM Express drivers provide 6x performance gains over kernel NVM Express drivers. The article "A Development Kit to Build High Performance Storage Applications" [14] explains how this is achieved. The result is a ongoing project called Intel SPDK[3], a user space, polled-mode, asynchronous, lockless NVMe driver. This provides zero-copy, highly parallel access directly to an SSD from a user space application.
After careful research into this project, in order to find a way to build a file system, I found out that key value stores such as RocksDB[5] can be configured to run in this environment. SPDK provides a flat file system called BlobFS(Blobstore Filesystem). This file system has many limitations and still under development. It has been tested to support RocksDB, but still very immature. However more research into available persistent storage engine APIs, I found experimental tools such as pmemkv[1] and Samsung's new KVSSDs [10]. This convinced me to design my project based on key value stores.
However since these software APIs require actual hardware, and are currently experimental, I wouldn't be able to use them for my development. So I used actual stable key value store databases. e.g. LevelDB and RocksDB.
The core interface that these databases provide, are mostly identical. Put and Get operations are for loads and stores. Iterators allow for range scans over key value pairs. Hence I have designed my system based on this basic core functionalities for easy transition between key value store engines.

2.3 Essential background on Key value databases

I mentioned LevelDB[6] and RocksDB[5] in the earlier section. These are high performance embedded databases for key value data, based on Log structured merge trees. [8] RocksDB is a fork of LevelDB by Facebook optimized to exploit many central processing unit (CPU) cores, and make efficient use of fast storage, such as solid-state drives (SSD), for input/output (I/O) bound workloads.

Chapter 3

Overview

The design goal of this project is to provide user space implementation of POSIX file APIs using key value stores. The project aims at accelerating single application workloads, to fully utilise emerging storage devices capabilities.
I present an API named **kvfs** (short for key-value file-system) with modularised design and capable of using any key value store implementation as storage engine.
kvfs is a C++ shared library developed using ISO C++ standard template library and C POSIX library. The target supported operating system is currently Linux. Under current implementation of **kvfs** applications can directly link to the library to perform storage operations.
The library is intended for applications that want to create and manage a file system on persistent memory without the kernel overhead. The interfaces in this library are modelled after the corresponding POSIX interfaces for file management. Using interfaces modelled on POSIX allows for easier transition for application developers.

3.1 kvfs storage engines

kvfs provides two storage engines that conform to the same common API 4.3. Most key value store engines can be used with **kvfs** implementing the common API 4.3.
RocksDB and LevelDB are currently available for use.

3.2 Requirements

Currently **kvfs** only supports a debian based linux, such as Ubuntu. I have developed this project on a Ubuntu 18.04 system. An script at `kvfs/install_dependecies.sh` can be used to install the requirements, for debian system only.

- `build-essentials`
- `cmake >= v3.9`
- `gcc, g++ >= v8.0`
- `libsnappy-dev # optional, used for LevelDB`

3.3 Building

kvfs-specific cmake variables:

- `BuildWithTests = ON # enables kvfs tests builds`
- `BuildWithRocksDB = OFF # changes kvfs key value store engine to use RocksDB`
- `BuildWithLevelDB = ON # By default kvfs uses LevelDB as storage engine. Much more compact library and builds faster than RocksDB`
- `BuildWithThreadSafety = OFF # options to enable synchronisation using mutexs`

Sample steps to build **kvfs** with LevelDB are included in the `kvfs/Readme.md` file:

- `mkdir build`
- `cd build`
- `cmake DCMAKE_BUILD_TYPE=Release ../`
- `cmake --build . -- -j 4`

3.4 kvfs API

kvfs API provides a generic interface for users to perform file system operations on different types of key value stores. The main entry is at `kvfs/include/kvfs/`. The interface implementation details and file system limitations are described in section 4.2.

3.4.1 Supported interfaces

The following interfaces are provided by **kvfs** through `FS` interface class.

3.4.1.1 Access Management

```
int Access(const char *filename, int how)
int ChMod(const char *filename, mode_t mode)
int ChDir(const char *path)
```

3.4.1.2 File Creation and Deletion

```
int Open(const char *filename, int flags, mode_t mode)
int Close(int filedes)
int Link(const char *oldname, const char *newname)
int UnLink(const char *filename)
int Remove(const char *filename)
```

3.4.1.3 open/creat Flags Support

```
O_APPEND
         This flag is supported.  Makes writes to files append only.
O_ACCMODE
         This flag is checked to be a valid of types:
                  (O_RDONLY, O_WRONLY, O_RDWR)
O_CREAT
         Supported, either creates a new file or opens and existing file
O_EXCL
         Supported, creates a new file if and only if it does not exist,
otherwise throws error.
O_ASYNC
         This flag is ignored.
O_CLOEXEC
         This flag is always set.
O_DIRECT
         This flag is ignored.
O_NOATIME
         Is supported.  Does not modify file's time stamps.
O_NONBLOCK or O_NDELAY
         These flags are ignored.
O_NOCTTY
         Not supported.
O_PATH
         Sockets are not supported.
O_SYNC, O_DSYNC
         These flags are ignored.  Writes depend on the storage engine.
Writes to persistent memories however are always synchronous.
```

3.4.1.4 File Naming

kvfs does not support renaming files between different instances of kvfs file systems.

```
int Rename(const char *oldname, const char *newname)
```

3.4.1.5 File I/O

```
ssize_t Read(int filedes, void *buffer, size_t size)
ssize_t PRead(int filedes, void *buffer, size_t size, off_t offset)
ssize_t Write(int filedes, const void *buffer, size_t size)
ssize_t PWrite(int filedes, const void *buffer, size_t size, off_t offset)
```

3.4.1.6 Offset management

```
off_t LSeek(int filedes, off_t offset, int whence)
int Truncate(const char *filename, off_t length)
```

3.4.1.7 File Status

```
int Stat(const char *filename, kvfs_stat *buf)
```

3.4.1.8 Directory Management

```
int MkDir(const char *filename, mode_t mode)
kvfsDIR *OpenDir(const char *path)
kvfs_dirent *ReadDir(kvfsDIR *dirstream)
int CloseDir(kvfsDIR *dirstream)
int RmDir(const char *filename)
char *GetCWD(char *buffer, size_t size)
std::string GetCurrentDirName()
```

3.4.1.9 Symbolic Link Management

```
int SymLink(const char *path1, const char *path2)
ssize_t ReadLink(const char *filename, char *buffer, size_t size)
```

3.4.1.10 Timestamp Management

```
int UTime(const char *filename, const struct utimbuf *times)
```

3.4.1.11 Special files

For now only S_IFREG file type is supported.

```
int Mknod(const char *filename, mode_t mode, dev_t dev)
```

3.4.1.12 Sync management

For invoking sync on a key value store engine, whose writes are asynchronous. These two functions are equals.

```
void Sync()
int FSync(int filedes)
```

3.4.1.13 kvfs special functions

```
int UnMount() override;
```

Chapter 4

Design and implementation

4.1 kvfs software architecture overview

Figure 4.1 shows the current software system's architecture of **kvfs**. **kvfs** provides a POSIX-compliant file system API for applications. A user application will link to **kvfs** API and file system requests are went through this API. **kvfs** translates the requests into key value store operations.

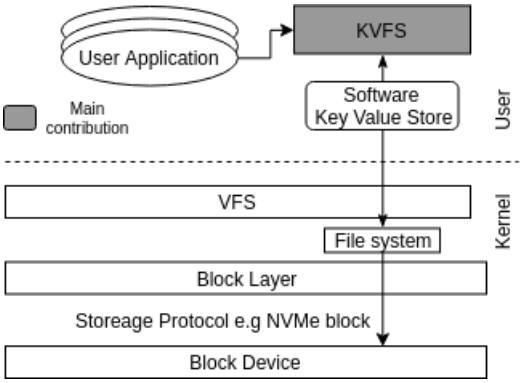

Figure 4.1: **kvfs** architecture

4.2 File system library

The main public library is called **kvfs**. There are two public header files, `fs.h` and `kvfs.h`.

- `fs.h` contains an abstract class called `FS`.
- `kvfs.h` contains an implementation of this interface, called `KVFS`.

The reason for this approach is to be able to protect actual methods implementations and also provide the ability to have a shared instance of **kvfs** within an application. This allows for multi threaded usage of **kvfs**. Thread safety is available as an option

in the CMake configurations of the the project, mutex locking overhead should be expected when this option is turned on.
Currently **kvfs** can only be opened/initialised by one process only, possibly multi threaded, as this is a limitation enforced by most key-value stores.
Figure 4.2 shows the available operations to the user. These methods are mostly identical or in some cases similar to the POSIX.1-2017 [7] specifications. I have used the POSIX specification extensively for implementing this file system. **kvfs** is compliant with the standard and users can expect functions to perform as they do defined by the standard. Please take note:
kvfs is a user space file system, hence it lacks protective features of a kernel file system. Consequently POSIX ACL(Access control list) is not supported.
In the following sub sections, implementation details and library modules are described.

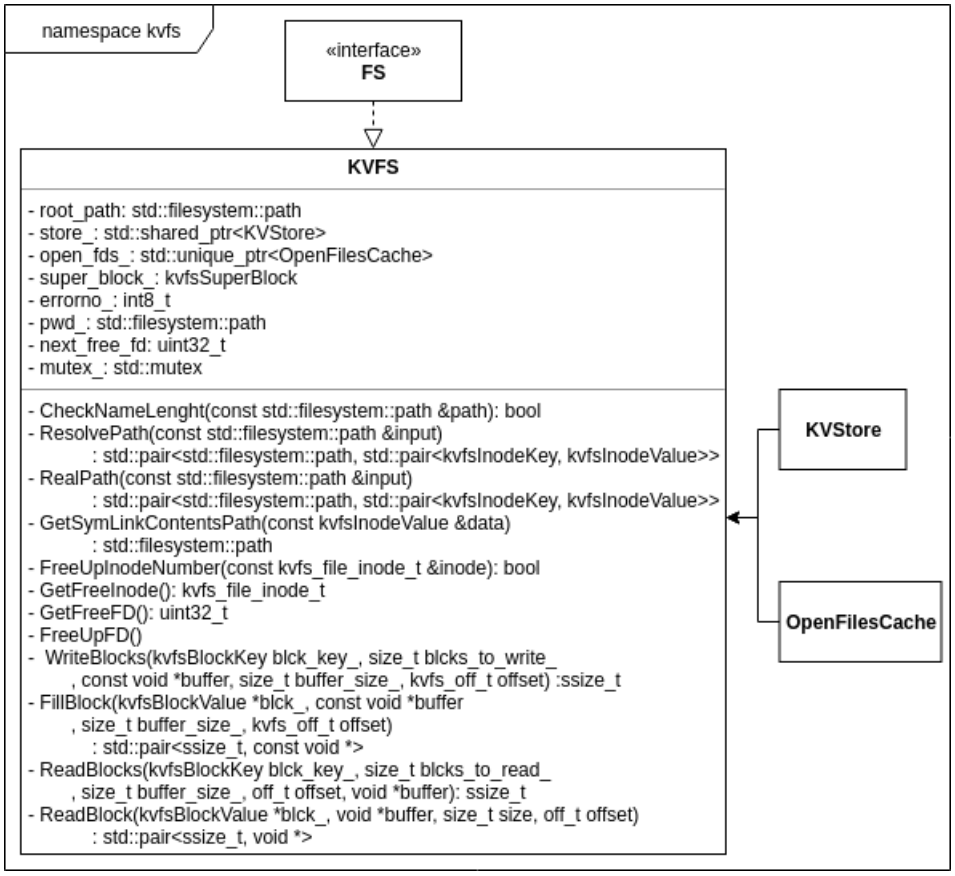

Figure 4.2: **kvfs** implementation

4.2.1 kvfs file system implementation

4.2.1.1 Terminology

kvfs unlike traditional file systems, doesn't need to directly communicate with a storage device. It relies on the underlying key-value store to communicate with a storage device. The key-value store would use its own efficient own data structures to achieve

high performance block allocation. Generally the block size for these key-value stores will be 4KiB. However since **kvfs** interfaces with only key-value pairs, LBAs (logical block allocations) are overshadowed in term of these key-value pairs. For the sake of convenience, the value sizes will be referred to as block_size of **kvfs** throughout the rest of this chapter.

4.2.1.2 Type definitions

There are several types defined in **kvfs**, which are used in the implementation and will be referred to respectively in this document. These are listed in the table 4.1 .

Table 4.1: **kvfs** typedefs

Real type	Name	Description
`size_t`	`kvfs_file_hash_t`	represents a type for hashes
`unsigned char`	`byte`	represents arbitrary bytes
`struct dirent64` or `struct dirent`	`kvfs_dirent`	dynamically represent dirent based on 64bit support
`struct stat64` or `struct stat`	`kvfs_stat`	dynamically represent stat based on 64bit support
`ino64_t` or `ino_t`	`kvfs_file_inode_t`	dynamically represent inode numbers based on 64bit support
`off64_t` or `off_t`	`kvfs_off_t`	dynamically represent offsets based on 64bit support
`__kvfs_dir_stream`	`kvfs_DIR`	Opaque structure for directory streams

4.2.2 Overview

A **kvfs** file system is split into a series of key-value groups. These groups each have their own key structure and have distinguished prefixes. This reduces performance difficulties and reduces seek times, prefixes can act as bloom filters for the underlying key-value store. In addition the distinction, helps for identification when performing iterations and de-serializations. All fields in **kvfs** are written to the key-value store relative to host's endianness.

4.2.3 Block

kvfs uses structures to store informations about files, file's data blocks, super block and other essential infos. There are several different structures defined for **kvfs**. Each of these structures have their distinct key structure. These are described in the following sub sections 4.2.4. **kvfs** file system maximums are described in 4.2. A definition called `KVFS_BLOCK_SIZE` can be set in the project's CMake configurations. A single typical file in **kvfs** when the kvfs block size is configured as 4KiB on a 64bit host can be upto 32 ZettaBytes, since a single file can address 2^{63} blocks. Maximum number of files in the file-system is limited by the inode number.

Table 4.2: kvfs File System Maximums

Item	Mode	Size
Inodes	64bit	2^{64}
	32bit	2^{32}
Blocks per file	64bit	2^{63}
	32bit	2^{31}

4.2.4 Layout

4.2.4.1 kvfs Superblock

The superblock is a structure which records various information about the enclosing filesystem, inode counts, maintenance information, and more. The kvfs superblock is laid out in 4.3 as follows in `struct kvfsSuperBlock`. The key for super block is a constant std::string `"superblock"`.

Table 4.3: kvfs superblock

Type	Name	Description
`uint64_t`	`total_inode_count`	Total number of used inodes
`uint64_t`	`next_free_inode`	First non-reserved inode
`size_t`	`freed_inodes_count`	Total number of freed inodes
`uint64_t`	`fs_number_of_mounts`	Total count of mounts
`time_t`	`fs_creation_time`	Creation time in seconds
`time_t`	`fs_mount_time`	Mount time in seconds, since epoch

4.2.4.2 kvfs Inode table

The inode is a unique number associated to a file in **kvfs**. As described in super block, free inode are tracked globally for allocations. **kvfs** also handles freeing inodes when files are removed from the file system. Freed inodes are checked first for allocation before using the global free inode counter from superblock. The allocation policy will be described later in this section. Inode value structure in **kvfs** is laid out in table 4.5. The definition can be found in `struct kvfsInodeValue`. The inode key is defined in `struct kvfsInodeKey`, which is described in table 4.4.

Table 4.4: kvfs inode key structure

Type	Name	Description
`kvfs_file_inode_t`	`inode_`	The inode number of this file's parent
`kvfs_file_hash_t`	`hash_`	This file's name hash value

4.2.4.3 kvfs Block

kvfs can store a file's data in fixed size blocks defined by `KVFS_BLOCK_SIZE`. To allow for more efficient and simpler design of blocks allocation for files, **kvfs** utilises the

Table 4.5: kvfs inode value structure

Type	Name	Description
kvfs_dirent	dirent_	The standard POSIX structure defined in dirent.h on Linux systems which consists of the file's name, name length
kvfs_stat	fstat_	The standard POSIX structure defined in stat.h on Linux systems which consits of file attributes, permission bits and more
kvfsInodeKey	real_key_	A pointer to the original file, if this file is a hard link

key-value interface and implements file's blocks differently. Block key structure and value structure are laid out in the tables 4.6, 4.7 respectively.

Table 4.6: kvfs block key structure

Type	Name	Description
kvfs_file_inode_t	inode_	The unique inode number of the owner of this block
kvfs_off_t	block_number_	This block's number

Table 4.7: kvfs block value structure

Type	Name	Description
kvfsBlockKey	next_block_	The kvfsBlockKey for the subsequent block
size_t	size_	This block's data size
unsigned char	data_	This blocks data, an array of arbitrary bytes

4.2.4.4 kvfs Freed Inodes

kvfs handles freeing inodes efficiently and stores freed inodes in arrays which are then stored in the key-value store. As described in superblock, freed inodes count are globally tracked and this counter is used to calculate the key. The structure used for keys and values of freed inodes are laid out in the tables 4.8, 4.9 respectively.

Table 4.8: kvfs freed inodes key structure

Type	Name	Description
char	name_	Name prefix of freed inodes key, constant "freedinodes"
uint64_t	number_	This freed inodes unique number

4.2.5 Inode Allocation policy

kvfs design recognises that data locality is a desirable quality of a filesystem. In order to reduce the total number of requests and speed up I/O in the underlying key-value

Table 4.9: kvfs freed inodes value structure

Type	Name	Description
kvfsFreedInodesKey	next_key_	The subsequent kvfsFreedInodesKey
uint32_t	count_	Counter for freed inodes in this block
kvfs_inode_t	inodes_	An array of inode numbers

store, **kvfs** key structure are designed such that each each key has a specific prefix. The prefix in keys allows the underlying key-value store to sort together similar keys, adjacent keys (according to the sort order) will usually be placed in the same block.

As described in table 4.3, the super block contains a global counter for next available inode number and freed inodes. When a file is created **kvfs** checks the freed inodes counter to see if there are any reusable inode numbers, an inode number is taken from the last freed inodes block and the respective counters decremented. Otherwise an inode number is taken from the global counter and next available inode is incremented.

4.2.6 kvfs Special inodes

Under the current version of **kvfs** the inode number ”0” is a fixed allocated inode for root directory key and inode number ”1” is allocated for root inode’s attributes. Entries in root directory will start from 1. These inode numbers cannot be freed nor allocated to another file. The root directory name in **kvfs** is defined as a single `"/"` which will therefore provide a known kvfsInodeKey for **kvfs** to function with.

4.2.7 Directory entries

In **kvfs** file system, a directory is basically a file which mode type is of `S_IFDIR`. To create hierarchical structure of directories, each new entry is stored by a kvfsInodeKey as in table 4.4, consisting of the file’s parent inode number and the file’s name hash value. A sample hierarchy is shown in figure 4.3.

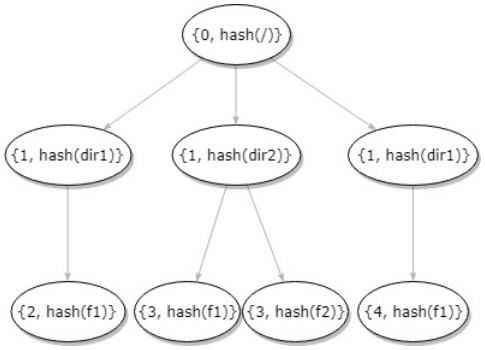

Figure 4.3: Sample directory heirarchy

4.2.8 Reading directory entries

When a directory is opened by function `OpenDir`, a new directory stream is generated. This directory stream is an Opaque structure which holds file descriptor, and a pointer to `KVStore::Iterator`. Contents of this structure are protected and cannot be queried via the user application. Upon creation of the iterator, an implicit snapshot of the key

value store is taken, hence any newly added file after the directory was opened will not be retrieved.
kvfs utilises the scan operation provided by the key value store to implement `ReadDir()` system call. For example, when using LevelDB:
The scan operation is designed to support iterations over arbitrary key ranges, which may require searching SSTables at each level. In such a case, Bloom filters cannot help to reduce the number of SSTables to search. However in **kvfs** only needs to scan keys sharing the common prefix – the inode number of the search directory. This can be achieved using iterator to seek to the first key containing the prefix. All other keys which are not in the desired range are ignored.
This is the fastest possible approach to scanning a directory in **kvfs**.

4.2.9 File access and permission bits

In **kvfs** whenever a file is created, the access mode of the file shall be set to the according value of the argument in respective function(e.g. `Open`, `MkNod`). The file's gid (group id) shall be set to the parent's gid and uid(user id) set to the process's uid. The file's access modes can be checked via `Access` function, also updated using `ChMod`.

4.2.10 File hard links

Creating a hard link has the effect of giving one file multiple names (e.g. different names in different directories) all of which independently connect to the same data on the disk, none of which depends on any of the others. [12]
In **kvfs** whenever a hard link file is created, the file's stat inode number is set to the original owner of the file. Refer to table 4.5. The rest of the file's attributes are same as the original file. Hence this link can access the original file's data blocks or directory entries. The link count of both the original and new file are incremented.

4.2.11 kvfs Cache

In order to track opened files, a cache has been implemented, using an unordered map data structure. This cache is indexed by allocated file descriptors. File descriptors are abstract handlers, a non negative integer, used to refer to any UNIX file type. The cache records the opened file's inode key-value, flags which the file has been opened with as well as the current offset of this handle.
Figure 4.4 shows the software architecture of this cache.

4.2.12 kvfs Error handling

To handle errors and inform the user application of critical exceptions, a class has been developed using `std::exceptions`. This class provides detailed error messages and corresponding error codes. The error numbers in **kvfs** comply with standard POSIX system error numbers defined in `error.h`. A typical usage of `kvfs` shall protect function calls in try-catch blocks for correct exceptions handling. The error codes for **kvfs** can be found in `kvfs/fs/kvfs/fs_error.h` for reference.

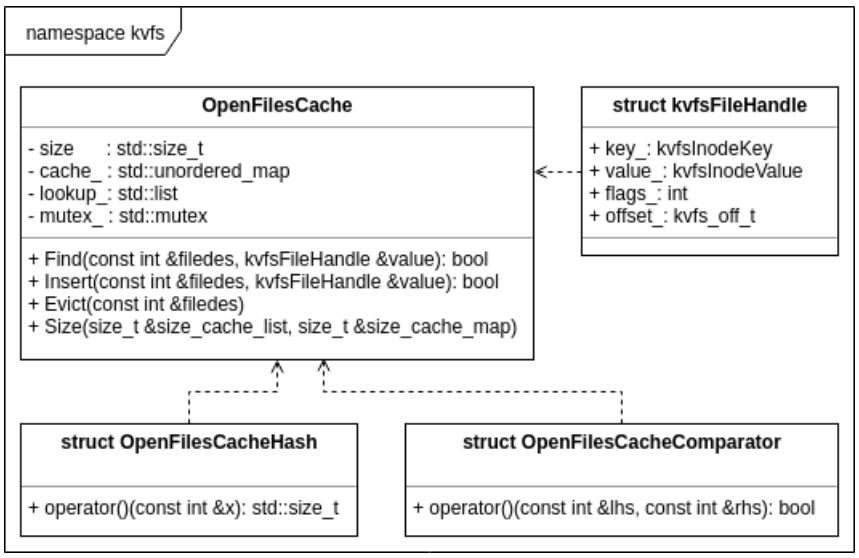

Figure 4.4: **kvfs** open files cache

For functions returning `int`
`0 = success`
`-1 = failure`
Otherwise as defined by function return type.
Error number is set in all cases of failure, Exceptions are raised to a standard Errno, values are noted.

4.2.13 Limitations

kvfs does not support the following features:

- Permissions: I only target single applications, permission control and POSIX ACL are redundant. Access and Chmod methods are provided but only if the application needs it.
- Extended attributes: These are not implemented since most linux applications do not rely on it. Also its out of the scope of this project.
- Directory change monitoring: This is very difficult to implement correctly.Imagine writing an implementation which scales up to 10M item directories and never misrepresents a change?The demands on handling race conditions correctly are very detailed and tricky to get right in a performant and portable way. User applications should carefully change directories, to avoid errors.

4.3 Key-Value store abstraction library

This library is used to generalise and abstract **kvfs**'s backing storage engine used to run **kvfs** on top of. It abstracts common and necessary operations needed for **kvfs**. Common functions such as but not limited to Put and Get, Iterator API and Write-batch are implemented in this interface. The library follows Zero copy principle, hence no

overhead is produced from this abstraction. Figure 4.5 displays this interface.

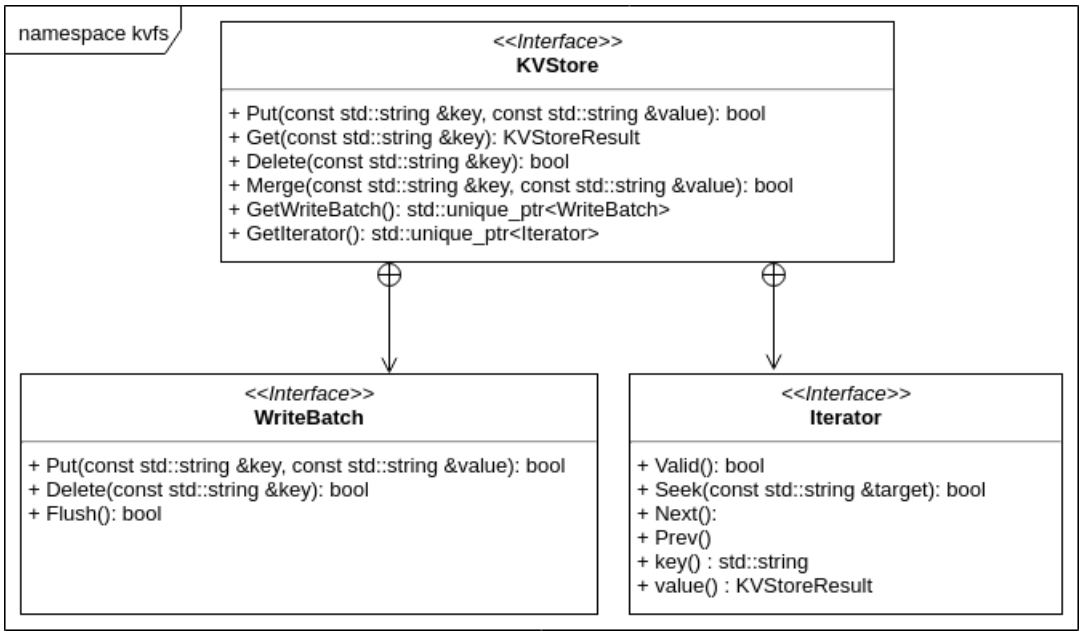

Figure 4.5: **kvfs** KVStore interface

There are two modules implemented for this library using LevelDB and RocksDB. Any other key value stores can be implemented following the same format. Figure 4.6 shows a sample implementation format.

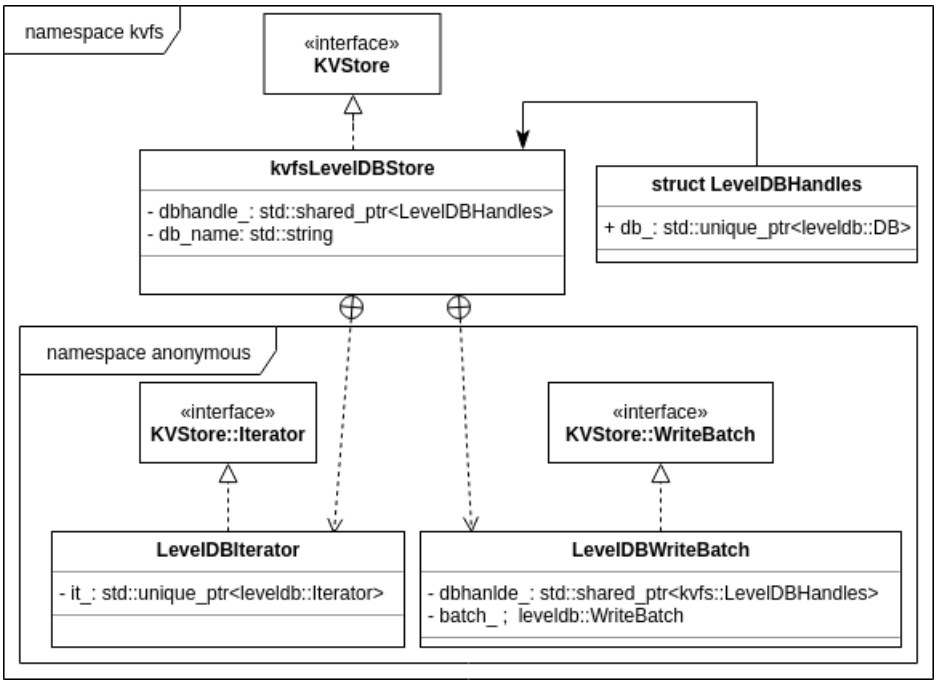

Figure 4.6: Sample **kvfs** KVStore implementation using LevelDB

Each module which implements the interface is composed of the following format:

- **Handler**: An immutable struct which encapsulates the store pointer. This struct is non copyable and takes a path name as it's argument. It handles store options and opening.
- **KVStore**: A class which implements the **kvfs**'s key-value store interface using the third party library. This class will also implement Iterator and Write-batch interfaces inside an anonymous name-space within this class.
- **Exception**: A class which implements error messages and exception raising, which is used for this module.

4.3.1 KVStore result

The return type of values from key value stores are std::string which unfortunately gives a relatively poor memory management control. A wrapper class 4.7 has been implemented around the returned string, with a few benefits:

- It can represent a "not found" result, this improves efficiency and avoids throwing exceptions when key lookups are not present.
- This class is move only, prevents unintentional copy of the data.

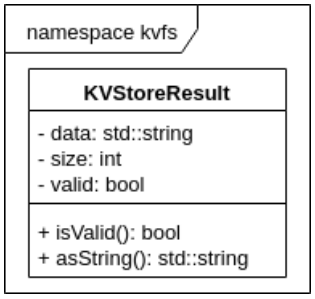

Figure 4.7: KVStoreResult

Chapter 5

Evaluation

5.1 Test system configuration

I have developed my project in a virtual machine. The virtual machine is configured with 4 cores, 4 GiB ram. The virtual machines configuration and details are as follows:

Linux	Ubuntu 18.04.1, Kernel 4.18.0-17-generic
CPU	Intel Core i7-7700HQ CPU @ 2.80GHz, 4cores - 8threads
DRAM	4GiB DDR4 2400MHz
HardDisk	SCSI, 32GiB
File system	ext4

My PC's host operating system is Windows 10 pro. The virtual machine is running on a NTFS file system. Windows is running on a Toshiba NVMe ssd. Intel premium rapid storage technology is enabled in BIOS. The drive's sequential read performance reach about 3 GB/s and writes for 1.5 GB/s.

There are three different evaluations I have implemented. The main focus is to evaluate the project based on the design goals given in section 3.

5.2 Sequential and random read/write

I tested **kvfs** against native LevelDB and ext4 file system. The overhead produced by **kvfs** is very low and almost negligible. The visible overhead, is due to limitations enforced by LevelDB. This library does not allow for appending to an existing key value pair in the database, every time the value needs to be read for appends. Furthermore LevelDB only allows for writing std::strings or C style strings, which complicates when writing binary data. The data size needs to be a fixed size. C style strings end when it reaches a null value. std::string does allow for storing null bytes but converting between std::string and data structures produces some small overhead caused by memcpy. Figures 5.1, 5.2, 5.3 and 5.4 shows these benchmarks. These tests create 512 files in either random or sequential order, write binary data in those files, reads and then deletes all created files. **kvfs** performs very well against native LevelDB considering that not only it r/ws the file's data content, but also has to update the file's timestamps

and file descriptor offsets. LevelDB in the other hand is only writing either randomly or sequentially generated key value pairs. ext4 of-course has the best performance since it is running in Kernel. It is added here only for reference on how much overhead from running kv databases on a file system is produced.
Ongoing projects such as Samsung's KVSSDs [10] are trying to tackle this very problem, by developing specialised firmwares for NVMe that support key value store operations natively. This would mean, there are no overheads of kernel drivers or file systems. Their benchmarks shows 10x performance gains over RocksDB instances running on a linux file system. Their API also allows for a Zero-Copy design pattern. This suggests that my implementation could be even further enhanced, to achieve even lower overhead. However, as I mentioned in earlier chapter, these projects are still under development and were not suitable or in some cases impossible for me to use.

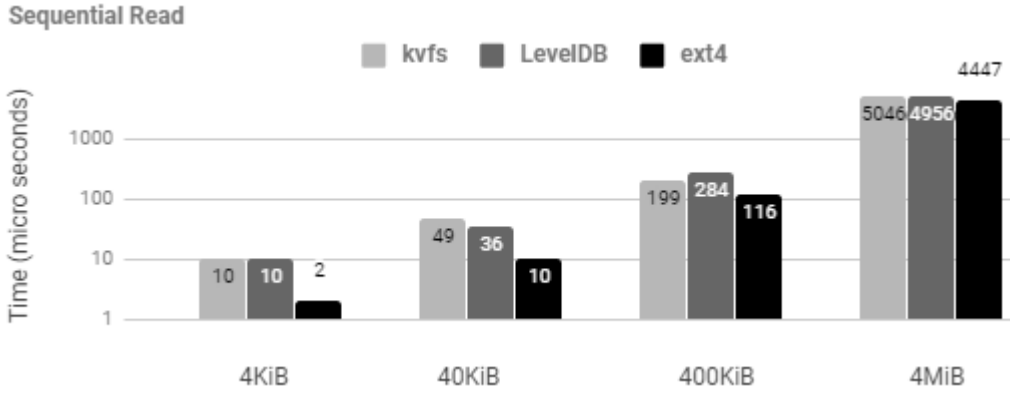

Figure 5.1: Sequential read performance of 512 files

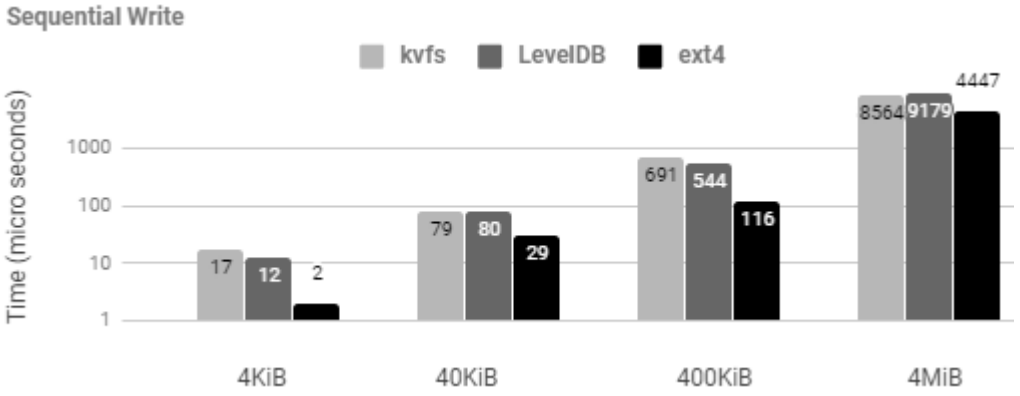

Figure 5.2: Sequential write performance of 512 files

5.3 Scan queries

To test the performance of metadata intensive workload in **kvfs** against ext4, I created a test to generate nested directories of 10000 files, and ran readdir on each direcory. After calculating an average, I developed the following graph 5.5. Intresingely, bf kvfs

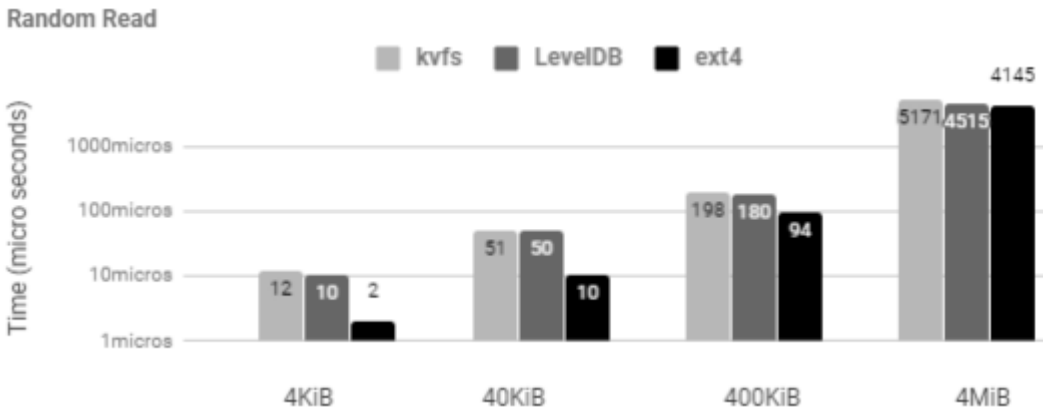

Figure 5.3: Random write performance of 512 files

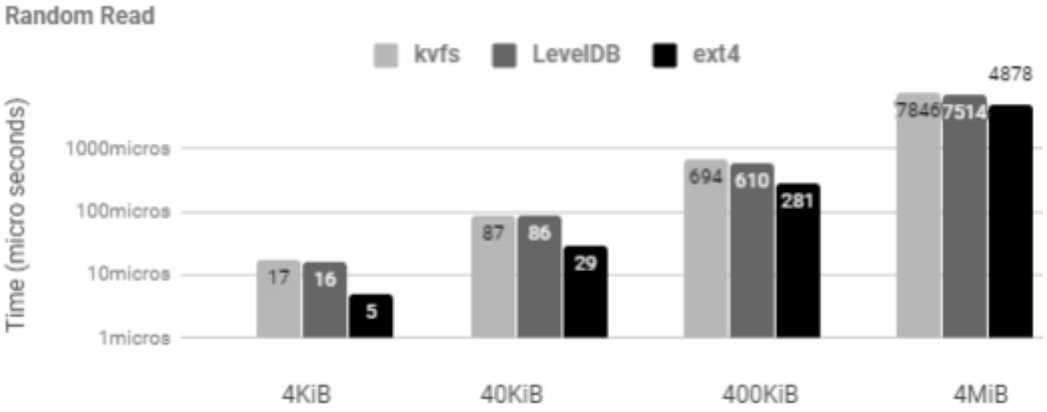

Figure 5.4: Random write performance of 512 files

perform very similarly in mkdir operations. Perhaps because in **kvfs** the file's metadata is stored with one operation, but for ext4 the creation of new file's is more costly. Moreover, readdir is slower in **kvfs** since, entire file's attributes are returned from the kv store. However the differenece can be argued to be the overhead of running LevelDB on ext4, as seen in previous benchmarks. Delete operations, seem very expensive, enforced by LevelDB delete performance. When a key value pair is deleted, in regards to LevelDB, entire sorted table files need to be rewritten. This would be more expensive consequently, than single deletion that takes place for ext4 files.

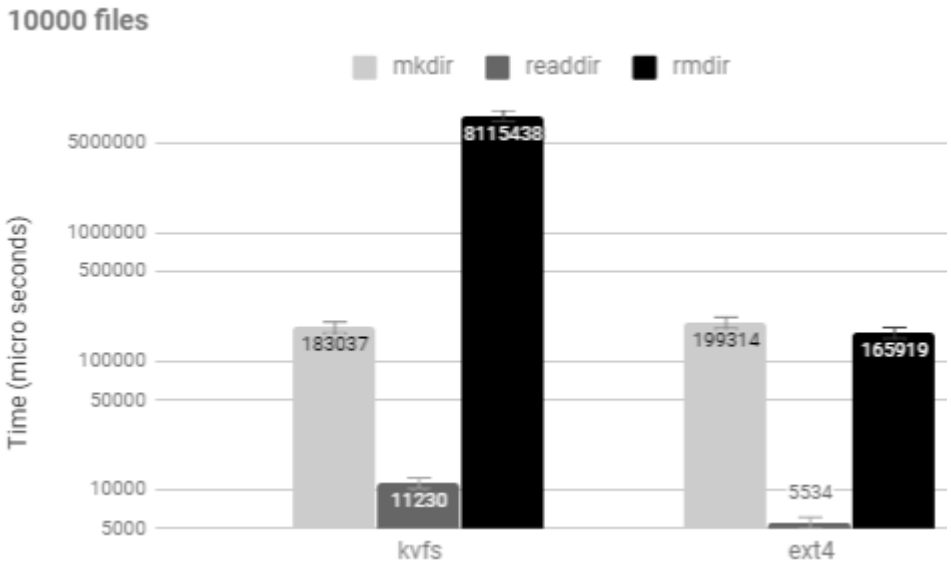

Figure 5.5: Creating, scanning and removing 10000 directories

Chapter 6

Related work

Pmemfile [4] is project with the goal to provide low-overhead userspace implementation of file APIs using persistent memory. This has similar goals to my project, achieves the no kernel overhead design and current applications can run with this file system using LD_PRELOAD. However the project does not use a key value store engine, rather directly uses persistent memory programming kit [2].
kvfs on the other hand, links directly with the user application, but that would mean limited compatibility. However, **kvfs** can be used with self encrypting key value store engines, to support encryption. Furthermore, with the upcoming KVSSDs, **kvfs** could possibly perform similarly to pmemfile project, in terms of performance.
Next C++20 standard [11], will standardise file APIs, therefore **kvfs** could be implemented in a standard way, to support more operating system and easily integrate into user applications.

Direct access File systems Kernel library modules provided by pmdk[2], have been patched into most operating systems, accelerating their file system operations, by avoiding extra copies of page-cache for persistent memory devices. These prove to be very practical improvements for current most widely used file systems. However the kernel overhead, would still become a bottleneck for low latency access needs in some user application workloads. Hence, these applications could benefit from user space file system. Intel SPDK[3] shows how kernel can become a bottleneck through their benchmarks.

Chapter 7

Conclusion and future work

In this paper I presented **kvfs** a user space file system based on key value stores. I developed this file system complying with POSIX standards and evaluated the file system's overhead against a native key value store engine(LevelDB).

kvfs with the current implementation, can have many interesting use cases. Applications that sandbox particular softwares could use **kvfs** to store applications data changes.

kvfs current implementation, updates file's attributes whenever the file is accessed. This produces an overhead which could potentially be done in the background, such that read/write operations take less even less time. A thread pool can be designed to push low priority updates to background. This thread poll would be an internal feature of **kvfs**.

A POSIX compliance benchmark test for **kvfs** is for future contribution, for correct expected behaviour. This is a large project on its own and was out of scope of this project.

A system call intercept library can also be implemented for **kvfs**, to seamlessly run user application on **kvfs** without application modification. Another difficult task which was not possible to achieve in the time frame of this project.

Bibliography

[1] Intel Corporation. Key/Value Datastore for Persistent Memory. https://github.com/pmem/pmemkv, 2019. pmemkv is a local/embedded key-value datastore optimized for persistent memory.

[2] Intel Corporation. Persistent Memory Development Kit. https://github.com/pmem/pmdk, 2019. PMDK is a collection of libraries and tools for System Administrators and Application Developers to simplify managing and accessing persistent memory devices.

[3] Intel Corporation. Storage Performance Development Kit. https://spdk.io/, 2019. SPDK provides a set of tools and libraries for writing high performance, scalable, user-mode storage applications.

[4] Intel Corporation. Userspace implementation of file APIs using persistent memory. https://github.com/pmem/pmemfile, 2019. Pmemfile project's goal is to provide low-overhead userspace implementation of file APIs using persistent memory.

[5] Facebook. A Persistent Key-Value Store for Flash and RAM Storage. https://github.com/facebook/rocksdb/, 2019. RocksDB library provides a persistent key value store.

[6] Google. LevelDB is a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values. https://github.com/google/leveldb, 2019. The leveldb library provides a persistent key value store. Keys and values are arbitrary byte arrays. The keys are ordered within the key value store according to a user-specified comparator function.

[7] The Open Group. POSIX 1-2017. http://pubs.opengroup.org/onlinepubs/9699919799/, 2019. The Open Group Base Specifications Issue 7, 2018 edition.

[8] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, 1996.

[9] Andy Rudoff. Persistent memory programming. *;login:*, 42(2), 2017.

[10] Ltd. Samsung Electronics Co. KV SSD host software package. https://github.com/OpenMPDK/KVSSD, 2019. KV SSD host software package includes the host software that operates with KV SSD.

[11] Programming Language C++ Library Evolution Working Group SG14 Low Latency study group. P1031R1 Lowlevel io library. www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1031r1.pdfa, 2019. Aproposal for a lowlevel io library very thinly wrapping kernel syscalls in to a portable standard library API, preserving all of the time and space complexities of the host platform.

[12] Wikipedia. Hard link — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Hard%20link&oldid=862682029, "2019". "[Online; accessed 02-April-2019]".

[13] Wikipedia. Non-volatile memory — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Non-volatile%20memory&oldid=887034647, 2019. [Online; accessed 04-April-2019].

[14] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. SPDK: A development kit to build high performance storage applications. In *IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2017, Hong Kong, December 11-14, 2017*, pages 154–161, 2017.