Design and Implementation of Database Support in nDiskFS

A Report submitted in partial fulfillment of the requirements

for the degree of Master of Technology

by

Nuka Siva Kumar

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

August 2020

Certificate

It is certified that the work contained in this report entitled "Design and Implementation of Database Support in nDiskFS" by "Nuka Siva Kumar" has been carried out under my supervision and that it has not been submitted elsewhere for a degree.

[illegible] 14-Aug-2020

Dr. Debadatta Mishra

August 2020

Assistant Professor

Department of Computer Science and Engineering

Indian Institute of Technology Kanpur

Declaration

This is to certify that the thesis titled **Design and Implementation of Database Support in nDiskFS** has been authored by me. It presents the research conducted by me under the supervision of **Dr. Debadatta Mishra** To the best of my knowledge, it is an original work, both in terms of research content and narrative, and has not been submitted elsewhere, in part or in full, for a degree. Further, due credit has been attributed to the relevant state-of-the-art and collaborations (if any) with appropriate citations and acknowledgments, in line with established norms and practices.

[illegible]

Signature
Name: Nuka Siva Kumar
Programme: M.Tech

Department: Computer Science and Engineering
Indian Institute of Technology Kanpur
Kanpur 208016

Abstract

Name of the student: **Nuka Siva Kumar** Roll No: **18111069**

Degree for which submitted: **M.Tech.** Department: **Computer Science and Engineering**

Thesis title: **Design and Implementation of Database Support in nDiskFS**

Thesis supervisor: **Dr. Debadatta Mishra**

Month and year of thesis submission: **August 2020**

Traditional storage devices like hard disk drives (HDDs) are capable of providing large storage capacity while solid state drives (SSDs) offer very efficient read and write performance. Hybrid storage systems consisting of HDDs and SSDs can provide combined benefits of both the storage technologies i.e., provide high volumes of storage and fast access speeds. Traditional file systems like Ext2, Ext4 are not designed for hybrid storage. Therefore, there has been a lot of focus to design file systems suited for the changing storage technologies. However, current hybrid storage file systems do not provide mechanisms to support custom policies for efficient disk cache management and data placement.

nDiskFS, an in-house hybrid storage file system under development, is a basic hybrid storage file system. However, nDiskFS does not support arbitrary user space applications, specifically support for data base servers is yet to be implemented. Further, nDiskfs does not have a modular design for better resource accounting and a pluggable policy framework. In this project, we design and implement different file system functionalities like file mappings, support for vectored reads and writes, file synchronization, direct I/O mechanisms in nDiskFS. Moreover, we propose a pluggable policy framework for data placement, page-cache management along with modular resource accounting support. We demonstrate the database support by executing OLTP workload with several configurations on MySQL database server hosted on nDiskFS filesystem and comparing the performance against Ext4 filesystem.

Acknowledgements

I would like to thank my supervisor Dr. Debadatta Mishra for his dedicated support on my project. He kept motivating me and he always guided me in the right direction whenever I needed them. His timely inputs helped me speed-up the work and he also made sure I finish my project on time.

I would also like to thank Mr. Pallav Agarwal and Mr. S V Shanmuga Sundar for his contributions to this project. They started this work project the guidance of Dr. Debadatta Mishra, made a working file system before handing over the same to me.

Contents

List of Figures

List of Tables

Abbreviations

HDD	**H**ard **D**isk **D**rive
SSD	**S**olid-**S**tate **D**rive
CPU	**C**entral **P**rocessing **U**nit
FS	**F**ile **S**ystem
Ext2	Second **Ext**ended File System
Ext3	Third **Ext**ended File System
Ext4	Fourth **Ext**ended File System
API	**A**pplication **P**rogramming **I**nterface
VMA	**V**irtual **M**emory **A**rea
VFS	**V**irtual **F**ile **S**ystem
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
I/O or **IO**	**I**nput/**O**utput
KB	**K**ilo-**B**yte
MB	**M**ega-**B**yte
GB	**G**iga-**B**yte
LRU	**L**east **R**ecently **U**sed
BIO	**Block I/O**
POSIX	**P**ortable **O**perating **S**ystem *I*nterface

Dedicated to Indian Institute of Technology, Kanpur

Chapter 1

Introduction

Due to the recent technology advancements, heterogeneous storage systems are more prominent in modern-day computing. Systems equipped with multiple storage devices like traditional Hard Disk Drive (HDD) combined with fast Solid State Disks (SSDs) provide better storage efficiency and cost benefits. HDDs are meant for storing high volumes of data. However, due to the mechanical movements associated with the HDDs, I/O performance bottleneck is a major concern. On the other hand, processors built with high speed transitions and gates operate at a speed magnitude higher than the storage devices. The speed mismatch always leads to poor application performance as the CPU has to wait for the slower devices like HDDs. In the last decade, SSDs are more prominent because of its characteristics of high read/write speeds along with low latency compared to HDDs [1, 2]. However, SSDs are not very cost effective and present unique challenges like write endurance. To get the benefits of both SSDs and HDDs, many systems employ both the HDDs and the SSDs. In this context of hybrid storage systems, the community is faced with a unique challenge to design efficient system software layers like file systems. Note that, the file system layer solution is one of the potential solutions and there can be solutions at other layers as shown previously [3, 4, 5].

Traditional filesystems like Ext2, Ext4, XFS etc. are primarily designed considering the storage characteristics of HDDs [6]. As shown by many previous research works, existing filesystems are not very efficient to manage characteristics of hybrid storage systems consisting of the SSDs and HDDs [3, 4, 5]. An ideal hybrid storage file systems should take advantage of large storage capacity in HDDs and fast access speeds from SSDs. Other hybrid storage filesystems lack support for file data and meta-data management at a block granularity [3, 4, 5]. Further, most OSes use the DRAM pages to implement page cache—a disk cache to store frequently accessed file system content. However, the OS kernel doesn't know the storage characteristics of the blocks. Therefore, in heterogeneous storage systems, single page cache eviction policy may not fully leverage the page cache benefits. To tackle these issues, nDiskFS was conceptualized and being developed as a full fledged file system. nDiskFS is hybrid storage filesystem to manage SSDs and HDDs in an efficient manner [3]. Some of the important design elements of nDiskFS are—it manages the page-cache in the file system layer and, it decides data and meta-data placement at a finer granularity.

1.1 Scope and Motivation

nDiskFS was initially developed by IITK graduates Mr. Pallav Agarwal and S V Shanmuga Sundar under supervision of Dr. Debadatta Mishra. The existing nDiskFS supports basic directory operations like `create`, `rename`, `delete` and supports file operations like `create`, `copy`, `delete`, list, move and read/write etc. However, there are many features unimplemented in nDiskFS which becomes an issue to support arbitrary user space applications. Moreover, there are some design modularity concerns which impedes further improvement of policies related to page cache management and dynamic data placement using schemes like data migration. In this section, we highlight these issues as follows,

Support for databases: The current nDiskFS lacks many features necessary to run arbitrary user space applications. Specifically applications like a database engines (e.g., MySQL) require extensive file system support which are not implemented in the current version of nDiskFS. Applications like data base engines require APIs to map files into the process address space (file `mmap`), synchronize the file content using system calls like `fsync` and perform direct I/O bypassing the file system caches (using `O_DIRECT` flag).

Modular resource accounting support: Resource accounting plays a crucial role in many ways—*(i)* Designing better policies, *(ii)* Studying different application behavior and, *(iii)* Debugging issues like memory leaks. nDiskFS maintains statistics to employ policies related to placement of blocks, page cache management and respond to events like memory pressure. Memory resource accounting is non-trivial as memory resource is not only used for page caches and caching meta-data (unit of 4KB), it is also used to maintain several file system data structures (variable sized allocations). The current version of nDiskFS does not have a modular approach with respect to the memory resources usage for different purposes.

Pluggable policy framework: There are two primary policy aspects in nDiskFS. First, nDiskFS implements its own page cache and therefore, page cache eviction policy becomes a necessity. Second, nDiskFS implements policies related to the block placement depending on the access and other characteristics by employing block migration (e.g., from HDD to SSD). In current nDiskFS, the mechanisms for both page cache management and block migration are tightly coupled with the policies. Therefore, any further research on policy design would require understanding the complex mechanisms which will be very inefficient in terms of time and efforts.

This project tries to address the above limitations of nDiskFS by designing new features and augmenting existing design as explained below.

1.2 Project Contributions

This project aims to build nDiskFS support to execute arbitrary user space applications on nDiskFS. Mainly, we focus on the database engines (e.g., MySQL) to execute correctly on nDiskFS. Moreover, this project tries to eliminate the modularity design issues like memory resource accounting and maintaining statistics for policy enforcement. This project tries to creates a general framework for the policies underlying nDiskFS and, debugging support for locks and memory usage.

Support for databases: To support arbitrary user space applications like database engines (e.g., MySQL), the Linux virtual file system (VFS) operations like memory mapped file I/O and synchronous file I/O operations are required to be implemented. As part of this project, we enable the direct I/O bypassing the nDiskFS page-cache and designing new dirty syncing extensions to the existing page cache design. Moreover, nDiskFS creates a new flag in the Linux kernel (`O_FSDIRECT`) to bypass OS-layer caching and transfers the file data between the disk and application memory without any intermediate caching. Further, iterative read/write calls originating from the VFS layer to support vectored I/O operations from the application layer is required to be implemented. To implement the `mmap` system call, this project enables the mapping of nDiskFS file to the process address space by integrating the file system with the Linux memory management subsystem. These virtual memory operations (using the kernel's VMA operations) are implemented in the nDiskFS as part of the project to expose file-mapped virtual address to the user space.

Modular resource accounting support: This projects proposes and implements a new resource accounting manager that supports memory allocation requests from different nDiskFS subsystem to be tracked. Resource accounting manager accounts the memory used by the nDiskFS at both page and variable sized slab granularity and supports debugging memory leaks in the filesystem. Further, for better policy design, we propose to maintain the access statistics at three different granularity—block, file and filesystem. This enables seamless use of statistics by any general policy framework.

Pluggable policy framework:

The pluggable policy framework segregates mechanisms and policies and, removes the rigidness of the policy design in nDiskFS. Using this framework, access pattern policies and eviction policies can be decoupled from the filesystem. This design makes custom policy design seamless which can be used for further research on the policy directions. Moreover, this can be used as a tunable parameters by the system administrators.

The summary of project contributions are as follows :-

- Design and implement functionalities like memory mapped file I/O, synchronous file I/O, vectored I/O calls, direct I/O to support database workloads.
- Propose a single-point resource accounting manager to provide statistics related to resources like the memory and file data access.
- Create a access pattern framework and eviction framework to implement various pluggable polices.
- Demonstrate database support of nDiskFS by executing MySQL server workload and comparing the performance with Ext4 filesystem.
- A comprehensive document describing the design and implementation of nDiskFS with detailed reference to the code-base.

The rest of report is organized as follows. Chapter 2 explains the background details, some description of system calls working in traditional filesystems . Chapter 3 provides the implementation details of new features added to the nDiskFS . Chapter 4 explains the experimental setup and presents evaluation of nDiskFS performance vis-a-vis Ext4 filesystem. Finally Chapter 5 concludes the project and presents the future scope of the nDiskFS filesystem.

Chapter 2

Background and Related Work

This Chapter explains necessary functionalities required to support database engines and description about how these functionalities are implemented in the Linux kernel. Moreover, we explain the nDiskFS modularity design concerns for resource accounting and policy framework along with their limitations in the current design.

2.1 Filesystems in Linux

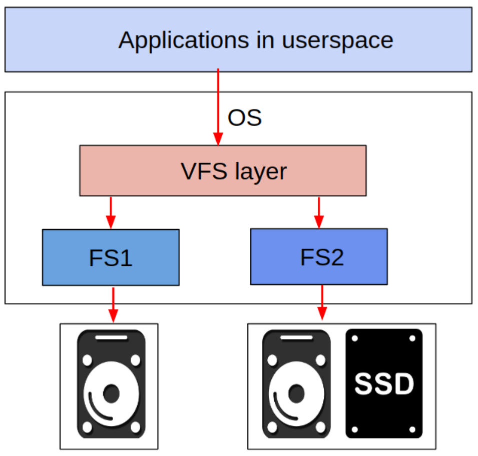

FIGURE 2.1: Storage software layers in the Linux kernel

HDDs and SSDs stores information in a persistent manner. Storage devices expose low level APIs to store and access data at a sector/block granularity. Filesystem software, an important subsystem of the OS, abstracts the low-level storage devices to expose a user-friendly view of the storage in the form of a file system tree.

Modern OSes support multiple filesystems and these filesystems support single block device or multiple types of block devices (hybrid) as shown in Figure 2.1. nDiskFS is a hybrid filesystem and manages both HDD and SDD. Linux kernel exposes system call API's to the user space applications and these applications use the system calls to retrieve and store data from the underlying file system. The Linux kernel internally has VFS layer which acts as a generic redirection layer between the system calls from the user space applications and the underlying filesystem implementations. The generic VFS layer allows flexibility in designing new filesystems by enforcing a standard interface for the filesystem developers.

2.2 Page-cache in the Linux kernel

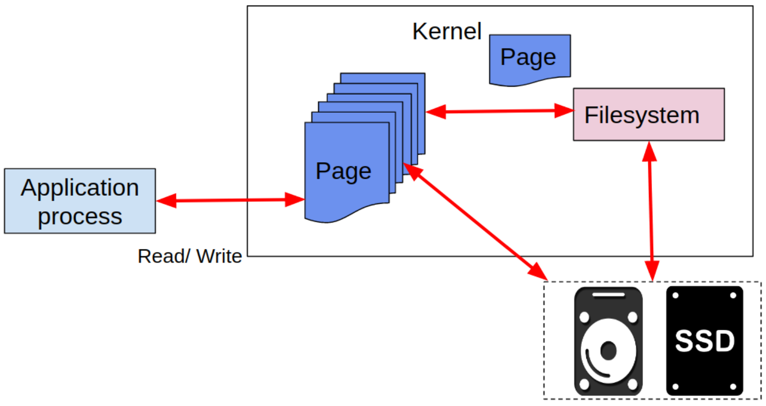

Figure 2.2: Cached I/O using the page-cache in the Linux kernel

Traditionally filesystems manage disk devices and they implement system calls from the user space. In the Linux kernel, the VFS layer acts as an intermediate layer between the system call handlers and the file systems to perform read/write the content to/from the user-space memory. User space applications are shown to be actively using some particular set of blocks [7]. Access to frequently accessed disk blocks can be optimized by caching the content in main memory and avoiding disk operations. Without caching, disk can become a bottleneck in application performance.

Kernel mostly refers to the page-cache while performing file I/O operations on behalf of the applications. If a process wants to read data from the filesystem to its memory, kernel first checks the file block in the page-cache. If the kernel finds the requested file content in page cache, then it copies the contents to the process memory. If the file block is not found in the page cache, kernel requests the filesystem to read the file data from the disk into the page cache. Next, the filesystem performs block I/O into the page-cache memory as shown in Figure 2.2. Finally, the page cache content is copied to the process memory. In the same way, while writing to a file, kernel writes the data to page-cache pages and later pages are written back to the underlying disk. This enables multiple processes to access the same file without performing additional disk I/O or mapping the file into process address space. All the page-cache pages are reclaimable by the kernel; during memory pressure, kernel evicts the page-cache pages. To free up the pages from the page-cache, kernel employs cache eviction policies to prioritize the pages for eviction.

2.3 Page-cache in nDiskFS

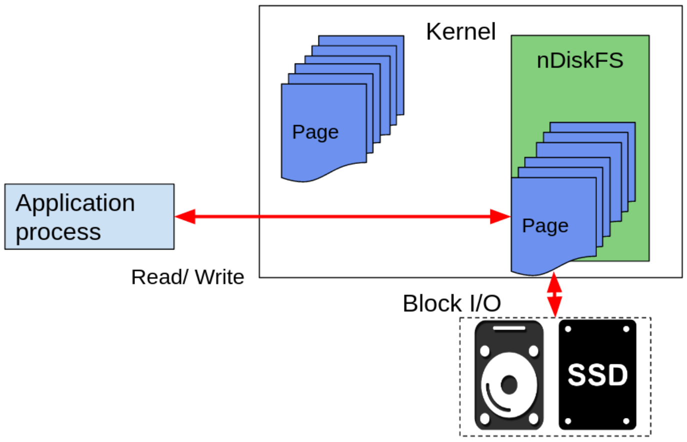

FIGURE 2.3: Caching file pages in the nDiskFS

nDiskFS maintains its own page-cache with an objective of managing with policies suitable for hybrid storage. Files and directories in nDiskFS bypass the Linux page-cache and directly request the nDiskFS as shown in Figure 2.3 . When a process wants to read/write to a file, kernel first checks the weather the Linux page-cache operations are enabled or disabled by the filesystem. nDiskFS disables the default Linux page-cache operations such that the kernel directly sends the request to the filesystem. When a read/write request made to nDiskFS, nDiskFS iterates through the page-cache pages and performs copy to/from to the user space memory. While iterating through the pages, if nDiskFS doesn't find the page then it requests to the block layer to read the data to the page as shown in Figure 2.3. All the nDiskFS page-cache pages are accounted as page-cache pages in the Linux kernel. nDiskFS page-cache pages are reclaimable by the kernel. During memory pressure, the kernel requests nDiskFS to free up the pages. nDiskFS has its own policies to evict the pages from the nDiskFS page-cache. While performing write operations in nDiskFS, dirty pages are accumulated over a period of time in the filesystem. nDiskFS employs a kernel thread that runs throughout the lifetime of nDiskFS and performs disk syncing by writing the dirty pages to the storage device at regular intervals or when a certain threshold of outstanding dirty pages is reached.

2.4 Support for Databases

nDiskFS unimplemented features	**Functionality**
Vectored Read and Write	Single vectored read/write to perform multiple regular reads/writes operations
File MMAP	Mapping the file into the process virtual address space
File synchronizations	Synchronize the file data present in memory and the persistent storage
Direct I/O	Facilitate applications to perform reads/writes directly to the disk bypassing the page-cache

Table 2.1: List of nDiskFS unimplemented features required to support database engines. The list is based on system call and VFS profiling data collected during execution of MySQL database server on a Linux system.

Database engines require extensive support from the filesystem. Databases map the file to the process address space through `mmap` and performs file operations on the address space. This provides more flexibility for file data handling compared to normal POSIX read/write APIs. Database engines support various modes of file data operations—*(i)* Databases maintain their own caches, *(ii)* Databases directly use disks without any caching. If the database engine maintains its own caching mechanisms, it requires for frequent file synchronizations using the filesystem APIs to maintain data integrity. However, some database engines directly use the filesystem without any caching but they require direct I/O support from the filesystem. Along with the above mentioned filesystem support, databases invoke vectored I/O operations using system calls like `readv`, `writev` etc. In vectored I/O, a single read/write system call contains multiple read/write requests and a file system should provide implementation for the corresponding VFS interfaces to serve the system calls.

Table 2.1 shows the list of file operations required for database support that are currently not implemented in nDiskFS. nDiskFS supports most basic file operations like creating files, opening files, performing read/write to/from file, closing of files, changing the file position using `lseek`, page-cached I/O and many other POSIX features. However, nDiskFS doesn't support vectored I/O operations, file synchronizations, mapping files to process address space and direct I/O bypassing the caching layers.

The rest of the section is organized as follows. Section 2.4.1 explains the working of `mmap` in the Linux kernel. Section 2.4.2 describes the file synchronization in the Linux kernel. Section 2.4.3 provides details of direct I/O mechanisms in the Linux kernel.

2.4.1 File mmap in Linux

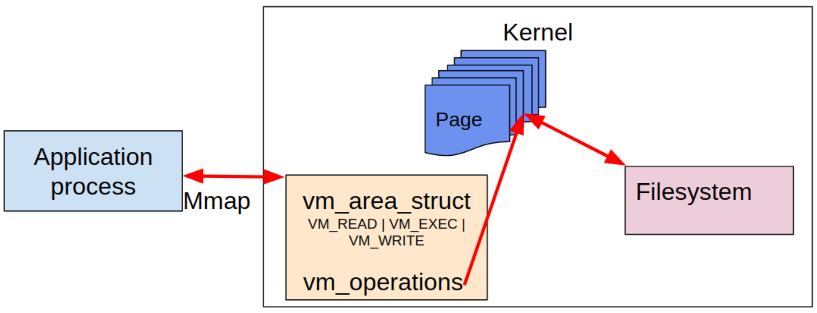

FIGURE 2.4: Mapping file to process in the Linux kernel

The `mmap` system call with a file as an argument maps the file to the process address space, after which the process can operate on the address space. MMAP-based file access offers more performance potential to the user-space applications because—*(i)* MMAP avoids extra copy of data from kernel pages to user buffer while performing read and write *(ii)* MMAP gives advantage of inter process communication across processes sharing the same mapping. Due to above reasons, many user-space applications take advantage of file MMAP.

Every Mmap is associated with a virtual memory area (VMA) which represents a range of virtual address with a specific permission. The Linux kernel internally manages the VMAs and implements different VMA operations. For example, when `mmap` is called from the user space, kernel allocates virtual memory area in the virtual memory address space and registers VMA operations. If a file is mapped to the process address space, kernel maps the user pages to file pages by registering VMA operations to the page-cache operations as shown in Figure 2.4. While accessing the VMA from the user space, if a page fault occurs, kernel invokes the kernel page-cache to handle the page fault. Kernel fixes the fault by filling the page from the page-cache and if a page is written, the kernel invokes marks the page-cache page as dirty. This allows the user space applications to operate on file seamlessly without being worried about multiple read/write system calls.

2.4.2 File synchronization

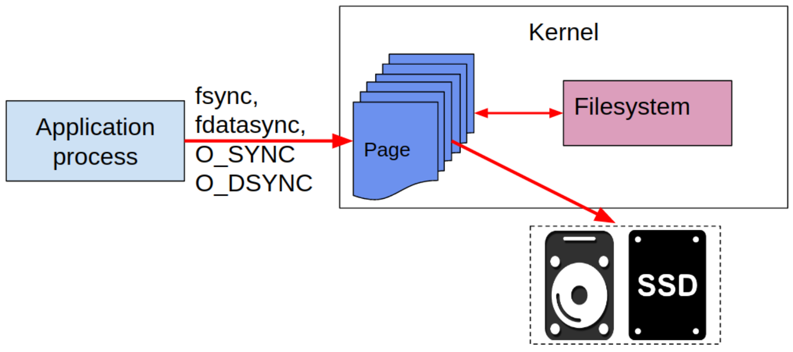

FIGURE 2.5: Linux file synchronizations

Generally, Linux page-cache is employed to boost the application I/O performance, but they require careful management to avoid data inconsistency issues. For example, when writes are performed on cached pages, page-cache state is different from the on-disk state which may lead to inconsistency in the filesystem. In an inconsistent state, if the system crashes, an application on resumption may not see the earlier modifications or the filesystem may be corrupted. Therefore, user space applications synchronize the in-memory file content with the disk by regularly invoking system calls like `fsync`, `fdatasync`.

When applications read/write to the files, the page-cache brings the corresponding file blocks into the page-cache with the help of the filesystem. Applications write content to page-cache pages and these pages not immediately written to the disk (in the default cached-I/O mode). Applications use system calls like `fsync, fdatasync` or use file flags like `O_SYNC, O_DSYNC` to synchronize the file content and file meta-data to the disk as shown in Figure 2.5. At the time of synchronization, the filesystem writes the file pages to respective blocks by performing block I/O write operations. Moreover, the current Linux kernel block layer implementation employs disk caching (different from page cache) to speed up the block accesses for blocks currently in the write queue. The kernel flushes the disk cache after completion of the block I/O.

2.4.3 Direct I/O in Linux

Direct I/O is a feature of filesystem to access the data bypassing the page-cache layers. Direct I/O avoids the extra copying of the file contents to/from the kernel page-cache layer. Unlike cached I/O, direct I/O performs file reads/file writes directly to/from the storage devices and bypasses the Linux page-cache. Most database engines maintain their own caches and consistency mechanisms. Therefore, these applications require Direct I/O.

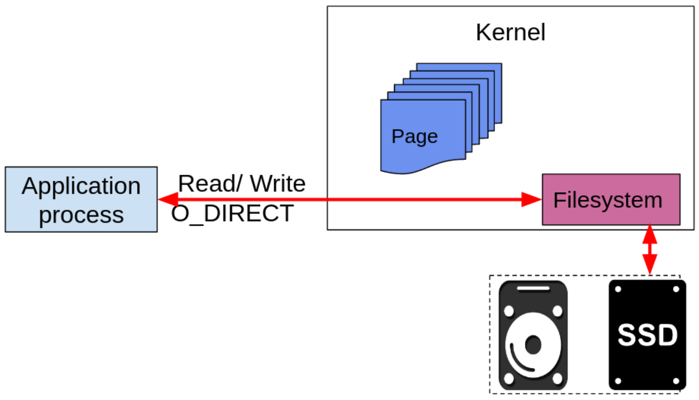

FIGURE 2.6: Direct I/O in Linux

Application perform I/O by setting `O_DIRECT` file flag while opening a file or through `fcntl` system call. Kernel checks file flags and if `O_DIRECT` flag is set, kernel bypasses the Linux page-cache as shown in Figure 2.6. Filesystem performs block I/O directly to the user space memory which avoids the copying the contents to/from the kernel page to the user buffer. Usually, direct I/O is sector aligned means smallest data transfer is done at the sector granularity.

2.5 Modular resource accounting support

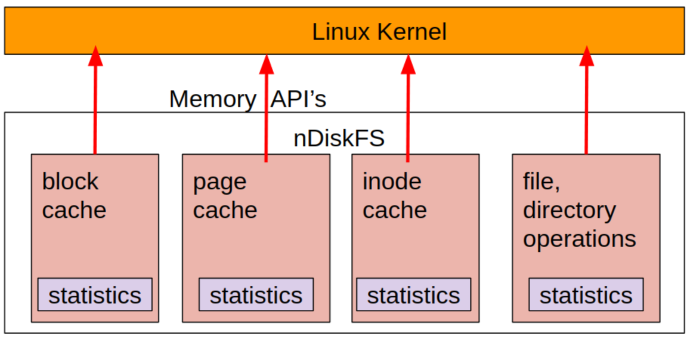

FIGURE 2.7: Scattered design in nDiskFS for memory API's and filesystem statistics

Current implementation of nDiskFS doesn't support modularity in memory accounting and maintaining statistics in the system. Most of the subsystems in nDiskFS use kernel memory API's to manage internal data structures. However, there is no central authority through which the subsystem memory usage can be tracked. Due to lack of single-point accounting, it becomes difficult to debug memory leaks in the current implementation of nDiskFS. nDiskFS maintains statistics to employ better policies like placement of blocks and page-cache management. Current implementation doesn't support modularity in accounting statistics and these accounting variables scattered across the filesystem code-base. These accounting variables are tightly coupled with the subsystems and while changing the subsystem design, management of these accounting variables also require change. We highlight two scenarios to demonstrate lack of modular resource accounting support in current nDiskFS.

Scenario 1: nDiskFS subsystems like block-cache, page-cache, inode-cache, file operations and directory operations use kernel memory API's to manage internal structures as shown in Figure 2.7. These subsystems allocate memory from the kernel slab caches and page allocators. These APIs are independently invoked from every subsystem. Therefore, it is very difficult to account for these memory usage statistics in the current nDiskFS.

Scenario 2: nDiskFS page-cache subsystem maintains statistics in internal structures of blocks and files. These statistics are not visible to the other subsystems. These statistics, when exposed, may provide better insights to design new policies.

2.6 Pluggable policy framework

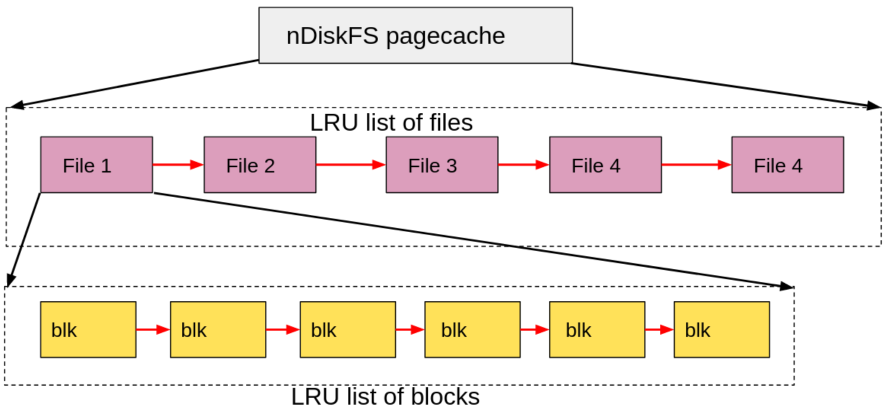

FIGURE 2.8: Shared design of nDiskFS for page-cache, eviction policy

nDiskFS has its own page-cache, maintains pages by employing eviction policies. However, eviction policy and design of the page-cache are tightly coupled with each other. In nDiskFS, data placement policy that migrates data from HDD to SDD is also part of the page-cache design. These policies use same page-cache design structures. We highlight two scenarios to explain the lack of pluggable policy framework in the nDiskFS.

Eviction policy: The file system needs to evict some of the cached blocks to handle memory pressure situations. It maintains two weighted evict lists—one for HDD blocks and the other for SSD blocks, and each evict list contains a list of files in the increasing order of their last access time as shown in Figure 2.8. A file can be present in the both the lists if it contains blocks from both the disks. If a file is being iterated or migrated, it won't be present in either of the lists. For each file in the evict lists, there is a list of clean blocks maintained in the increasing order of their last access time as shown in Figure 2.8. If memory pressure rises, the first file from each of the list is chosen if present and they are evaluated on the basis of their recency and the eviction factor of disk type to get the most preferred file to evict. The weights are given such that the blocks from SSD are preferably evicted than the blocks of HDD with same recency.

Here in the current page-cache design, eviction policy shares both the LRU list of files and the LRU list of blocks in the file. Therefore, any new policy that works at a block layer is difficult to incorporate in the current design.

Scenario 2: access pattern policy Every file and directory access are analyzed to deduce their pattern of access after every read or write operations. The pattern analysis is done as a requirement to data migrations. Access pattern policy uses statistics of page-cache and these statistics are part of the page-cache internal structures. Therefore, it is not trivial to design new data migration policies without understanding the details of the nDiskFS page-cache management internals.

Current nDiskFS has some unimplemented features which are required to support the database applications. Therefore, we need to provide vectored reads/writes, file synchronizations, mapping filesystem files to the process address space and direct I/O to access data directly from the persistent disks. Moreover, nDiskFS should have a single-point resource accounting manager and a pluggable policy framework to provide a platform for extensive policy exploration. In the next Chapter, we discuss detailed design and implementation of the above features in nDiskFS.

Chapter 3

Design and Implementation

In this chapter, we discuss about design and implementation details of the project. In Section 3.1, we provide the implementation details of functionalities required for database support. Section 3.2 describes the design to support modular resource accounting. Finally, Section 3.3 explains interfaces for the nDiskFS which supports pluggable policy framework.

3.1 support for databases

As discussed in the Section 2.4, nDiskFS has its limitations to host and execute database engines. The requirements of database engines are mentioned in the Section 2.4. Database engines require file mappings to process address space, file synchronizations, vectored read/write, cached I/O, direct I/O. nDiskFS supports page-cached I/O as discussed in Section 2.3. In this section we discuss the design and implementation details of other requirements for database support. Specifically, we provide implementation details to support MySQL database on the Linux system.

3.1.1 nDiskFS mmap

Generally, `read/write` systems calls take the data from the user space and copy to/from to the block devices. These calls perform read/writes to the pages in the kernel, this is inefficient because for every small read/write, process has to call to `read, write` system calls. Instead of that, if the process maps the file into the process address space then it can directly perform operations on the memory addresses. This allows processes to avoid extra

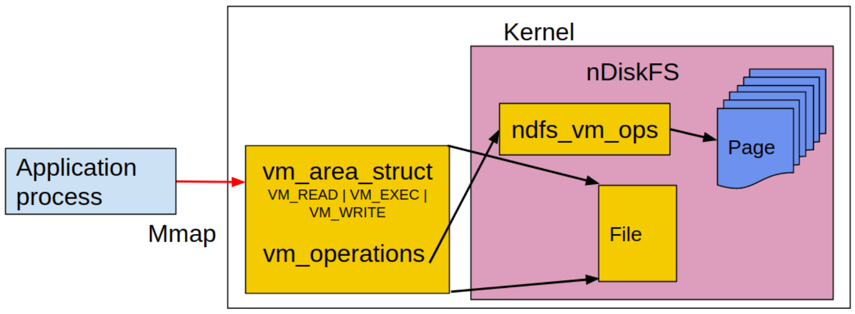

FIGURE 3.1: File mmap in nDiskFS

copy from user buffer to file pages and multiple processes operate on single file seamlessly by doing `mmap`. In most of the database engines, multiple threads work on the same files by mapping the file into the process address space. nDiskFS disables the Linux page-cache and to provide `mmap` functionality, nDiskFS exposes necessary functionalities to the Linux kernel.

To handle the `mmap` system call, Linux kernel allocates a VMA as we discussed in Section 2.4.1. nDiskFS exposes the `mmap` call handler such that, when a process invokes `mmap`, the VFS layer diverts the call to nDiskFS with VMA information. nDiskFS registers the VMA operations to the filesystem's VMA operations as shown in Figure 3.1. These nDiskFS VMA operations primarily deal with two functionalities—*(i)* handling the page fault in the mapped VMA region. *(ii)* keep track of writes to a page and forward page dirty information to the filesystem. The following code snippet provides the VMA handlers for nDiskFS.

```
static struct vm_operations_struct ndfs_vm_ops = {
.fault = ndfs_filemap_fault,
.page_mkwrite = ndfs_filemap_page_mkwrite,
.map_pages = ndfs_filemap_map_pages,
};
```

Handling page-faults in the mmap VMA

In nDiskFS, `fault` VMA operation registered as `ndfs_filemap_fault`. This nDiskFS handler is called when a page fault happens in virtual address range represented by the VMA corresponding to the file. `vm_fault` is structure contains the information about the page-fault. `vm_fault` contains file pointer and page pointer to be filled along with the page offset. nDiskFS searches the page in the page-cache and fills the page pointer with the nDiskFS page-cache page and gets the reference to the page. VFS layer expects the filesystem to lock the page before returning to the VFS layer.

Forward page dirty information

In nDiskFS, `page_mkwrite` VM operation registered as `ndfs_filemap_page_mkwrite`. This nDiskFS handler called when a page is dirtied in the VMA which informs the filesystem through this VM operation. nDiskFS gets dirty page offset from the `vm_fault` structure which is passed as a parameter. nDiskFS searches page offset in the page-cache pages and marks the page as dirty; later this page is synced to the disk by performing block I/O.

3.1.2 nDiskFS vectored read/write

Regular read and write operations access a contiguous file chunk i.e., a single system call performs a single read/write operation on the file. In vectored I/O, one single operation may contain multiple reads/writes to a single file at different offsets. Applications perform vectored reads/writes by calling the system calls like `readv, writev`. Vectored I/O operations can also be called from within the kernel through exported functions like `vfs_iter_read, vfs_iter_write`. For all of the above operations, VFS layer ultimately invokes the `read_iter, write_iter` implementations of the underlying filesystem. The `read_iter` reads the data from different offsets of a given file to multiple user buffers. Similarly, `write_iter` writes the data to the file from multiple user buffers. Due to this flexibility, database engines like MySQL use vectored I/O operations because these are atomic and can improve performance.

3.1.2.1 read_iter

In nDiskFS, `read_iter` file operation is implemented using the `ndfs_file_read_iter` callback and its prototype is, `ssize_t ndfs_file_read_iter(struct kiocb *iocb, struct iov_iter *to)`

Algorithm 1: ndfs_file_read_iter Algorithm

```
Input: iocb: I/O control block, iovIter: I/O Vector
Result: Transfer data to user space buffer from the file pages
FilePTR = iocb→ki_filp;;
i = 0 ;
while  i < iovIter→segments do
    ndfs_read(FilePTR, iovIter[i].base, iovIter[i].count, FilePTR.pos );
    i++;
end
```

VFS layer invokes this operation with `iov_iter` as an argument which contains segments of user buffers and lengths along with the `kiocb` object which contains the file pointer. The implementation of `ndfs_file_read_iter` is shown in Algorithm 1 where the handler iterates on the `iov_iter` vector and invokes the regular `read` implementation of nDiskFS.

3.1.2.2 write_iter

In nDiskFS `write_iter` regular file operation registered as `ndfs_file_write_iter`. callback and its prototype is,

`ssize_t ndfs_file_write_iter(struct kiocb *iocb, struct iov_iter *to)`

Algorithm 2: ndfs_file_write_iter Algorithm

```
Input: iocb: I/O control block, iovIter: I/O Vector
Result: transfer data from file pages to user buffer in iov_iter
filePTR = iocb→ki_filp;;
i = 0 ;
while  i < iovIter→segments do
    ndfs_write(FilePTR, iovIter[i].base, iovIter[i].count, FilePTR.pos );
    i++;
end
```

VFS layer invokes this operation with `iov_iter` as an argument which contains segments of user buffers and lengths along with the `kiocb` object which contains the file pointer. The implementation of `ndfs_file_write_iter` is shown in Algorithm 2 where the handler iterates on the `iov_iter` vector and invokes the regular `write` implementation of nDiskFS.

3.1.3 Extended nDiskFS page-cache API

nDiskFS bypasses the Linux page-cache and maintains its own page-cache. nDiskFS page-cache doesn't have API to support file synchronization operations and direct I/O implementation. These following newly proposed page-cache APIs assist in the implementation of file synchronizations, direct I/O feature in nDiskFS.

3.1.3.1 File data synchronization operations

In the proposed design, nDiskFS page-cache uses a new API i.e., `do_file_sync` method to syncs data pages of a file. This function write a range of dirty data pages of an inode to the disk. This method can be used to implement both `fsync` and direct I/O features.

Algorithm 3: File data syncing Algorithm

```
Input: filePTR: file pointer, startPage, lastPage
Result: sync range of data pages for a inode
pageCache = get_pagecache(filePTR) ;
pageCache→status = iteration ;
pageCache→itr_count ++ ;
pageOffset = startPage;
while pageOffset is in range(startPage, lastPage) do
    locks( fileDirtyLists );
    page = get_page_from_pageCache(pageOffset);
    if page is valid then
        if page is dirty then
            sync_page(page);
            remove_page( fileDirtyList, page ) ;
            if ref_count(page) = 0 then
                add_page( fileLruList, page ) ;
            end
        end
    end
    unlocks( fileDirtyLists );
    pageOffset ++;
end
pageCache→itr_count - - ;
```

`do_file_sync` pagecache API syncs the range of file pages as shown in Algorithm 3. This is an another iterator in the nDiskFS and it sets the page-cache status (maintained for eviction of inode etc.) to “on-iteration” and increments active thread count iterating on this file. This function locks the file LRU dirty lists so that no other iterator can add or remove blocks from the file LRU dirty list till the page is submitted to block layer for performing device write. Next, the function iterates over every dirty page in the range from the AVL tree of pagecache and creates block I/O requests (BIOs) and submits to the block layer to perform syncing. After syncing, the unreferenced clean blocks are added to the file LRU lists and file LRU dirty list is unlocked. Finally it decrements the active iterators count on inode; if the count becomes zero, the method sets the file status to idle.

3.1.3.2 File meta-data synchronization

Algorithm 4: File metadata syncing Algorithm

```
Input: sb: superblock, inodePTR: inode pointer
Result: sync metadata pages of a inode
pageCache = get_pagecache(inodePTR) ;
pageCache→status = iteration ;
pageCache→itr_count ++ ;
sync_inode();
sync_block_bitmaps(sb);
while page from the AVL indirect tree do
    lock ( fileDirtyLists ) ;
    if page is valid then
        if page is dirty then
            sync_page(page);
            remove_page( fileDirtyList, page ) ;
            if ref_count(page) = 0 then
                add_page( fileLruList, page ) ;
            end
        end
    end
    unlock( fileDirtyLists );
end
pageCache→itr_count - - ;
```

We created a new nDiskFS page-cache API i.e., `do_file_metadata_sync` to sync file system meta-data like inodes, bitmaps etc. This method can be used to implement both `fsync` and direct I/O features.

`do_file_metadata_sync` is also another iterator which iterates on the file page-cache. Firstly this function sets the pagecache status to "on-iteration" and increments the active iterators count on the file's inode. It syncs the inode, block bitmaps to the block device as shown in Algorithm 4. This function locks the file LRU dirty lists so that no other iterator can add or remove blocks from the file LRU dirty lists, till the page is submitted to block layer to sync. The method collects dirty indirect blocks from the AVL tree, creates BIO requests and submits the BIOs to the block layer to write the indirect blocks to the disk. After syncing, the function adds the unreferenced clean blocks to file LRU lists and finally it unlocks the file LRU dirty lists. Before returning to the upper layer, the function decrements the active iterators count on inode; if the count becomes zero, it sets the file status to idle.

3.1.3.3 File data eviction

nDiskFS page-cache implementation is extended with `do_evict_from_page_cache` API to evict range of clean pages of a file's inode. Usually direct I/O operations invoke this API.

Algorithm 5: File data eviction Algorithm

```
Input: filePTR: file pointer, startPage, lastPage
Result: evict range of data pages from a file
pageCache = get_pagecache(filePTR) ;
pageCache→status = iteration ;
pageCache→itr_count ++ ;
pageOffset = startPage;
while pageOffset is in range(startPage, lastPage) do
    lock( fileLruLists );
    page = get_page_from_pagecache(pageOffset);
    if page is valid then
        if page is !dirty then
            remove_page( fileLruList, page ) ;
            evict_page(page);
        end
    end
    unlocks( fileLruLists ) ;
    pageOffset ++ ;
end
pageCache→itr_count - - ;
```

`do_evict_from_page_cache` pagecache API evicts the range of file pages as shown in Algorithm 5. `do_evict_from_page_cache` function is an iterator which sets the file pagecache status to "on iteration" and increments the active iterators count of the the file inode. The function first acquires a lock on the file LRU lists to prevent other iterator from adding or removing blocks from the file LRU lists while eviction is active. This function gets the range of the pages from the AVL tree and evicts those pages and frees up the internal structures. Finally it unlocks the file LRU lists and decrements the active iterators count on the file inode.

3.1.4 nDiskFS file synchronizations

As we discussed in Section 2.4.2, normal write caches the blocks in the nDiskFS page-cache and syncs the blocks to the block device at a latter point of time. This improves the application performance but comes with a cost of data integrity. For example, if system crashes or a power failure occurs, then the application doesn't see the latest update on system reboot because the data cached in the volatile memory is lost. Databases provide ACID properties and always want to their data to be synced with on disk data, therefore use file synchronization APIs.

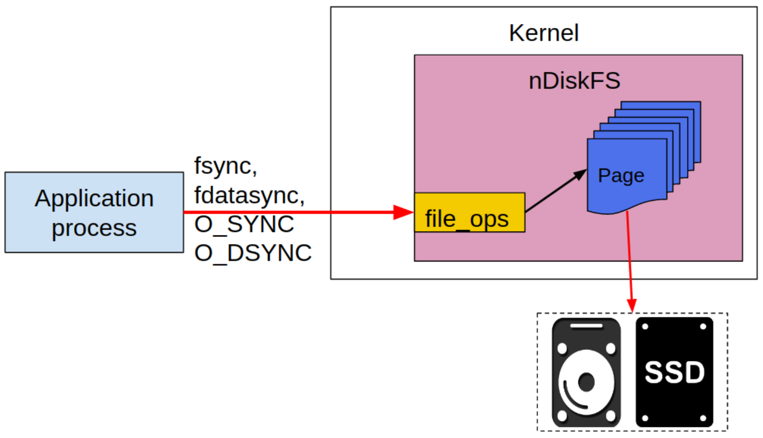

FIGURE 3.2: nDiskFS file synchronizations

nDiskFS maintains its own pagecache all the writes are cached in nDiskFS pagecache. If an application wants to forcefully sync the file then it calls `fsync, fdatasync` system calls or application sets the `O_SYNC, O_DSYNC` flags. nDiskFS calls to nDiskFS page-cache APIs `do_file_sync` to sync the file pages in the range of starting, ending pages. `do_file_sync` is one of the proposed nDiskFS page-cache extensions as discussed in the Section 3.1.3.1. If `datasync` or `O_DSYNC` flags not set, then nDiskFS calls to another pagecache API `do_file_metadata_sync` to sync metadata pages.

3.1.5 nDiskFS direct I/O

Cached I/O using page-cache has a disadvantage of copying the data from the block device to the kernel page followed by a copy from the kernel memory to user buffer. Instead of that, direct I/O directly copies the data from the block device to the process memory. Database engines like MySQL maintain their own caching structures and use direct I/O to make every write request to be persistent in the block device. In nDiskFS direct I/O implementation, both the Linux page cache and nDiskFS page-cache layers are required to be bypassed.

Linux kernel modification

Every filesystem in the Linux registers address space operations to use Linux pagecache. nDiskFS disables the Linux pagecache, due to this, Linux kernel doesn't allow to call direct I/O on filesystem. When an application opens a file with O_DIRECT flag, kernel doesn't invoke nDiskFS APIs. To bypass the Linux page-cache and allow the kernel to pass the direct I/O call to nDiskFS, we introduce a new O_FSDIRECT flag into the Linux kernel. We modified Linux kernel such that if O_FSDIRECT flag is set, kernel calls the nDiskFS direct I/O callbacks to perform read or write operations.

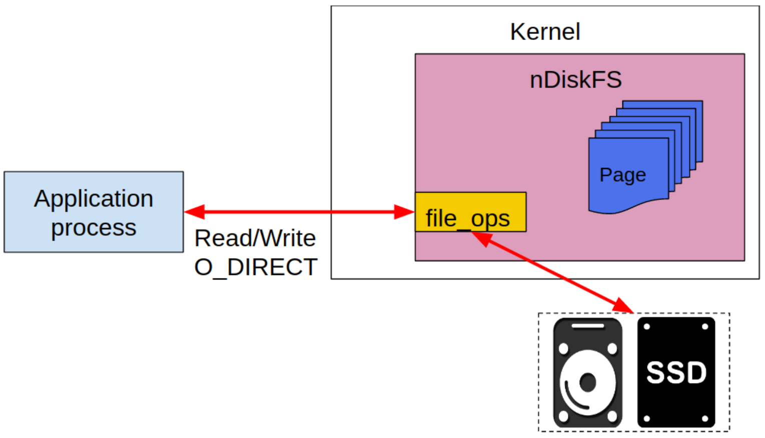

FIGURE 3.3: Direct I/O in nDiskFS

When application sets O_DIRECT either by opening file or through `fcntl` call, nDiskFS adds O_FSDIRECT flag to the file flags to bypass the Linux kernel.

nDiskFS direct I/O handler

In nDiskFS direct I/O, smallest granularity to do direct I/O is 4KB and it should be page aligned. Otherwise direct I/O falls back to cached I/O. Before performing direct I/O, nDiskFS makes sure that inode metadata and data to be in sync with the data on the disk. nDiskFS calls pagecache APIs (Section 3.1.3) `do_file_sync`, `do_file_metadata_sync` to sync the file data, metadata and calls `do_evict_from_page_cache` to evict the pages from the pagecache. nDiskFS directly copies to/from user pages from/to the disk device but kernel does not allow to do so because kernel does not have the mappings to the user pages. nDiskFS calls to `get_user_pages` kernel API to create kernel mapping to the user pages. For every block in the range, nDiskFS creates BIOs, submits to block layer and directly transfers data between disk device and the kernel mapped user page as shown in Figure 3.3. Finally nDiskFS unmaps the user pages from the kernel.

3.2 Modular resource accounting support

As we discussed earlier in Section 2.5, nDiskFS has lack of modularity in *(i)* Memory usage accounting *(ii)* Accounting statistics.

nDiskFS has many internal subsystems and these subsystems use kernel memory APIs to allocate and deallocate the internal structures. Due to lack of memory usage accounting, it is non-trivial to debug memory leaks in the filesystem. Therefore, there is need of a central authority, that takes care of memory usage of the entire filesystem.

nDiskFS doesn't have a modularity in accounting statistics and these are tightly coupled with internal data structures. These accounting statistics assist pagecache management policies, block placement policies. However, lack of modularity, introducing policies into the filesystem becomes a complex task.

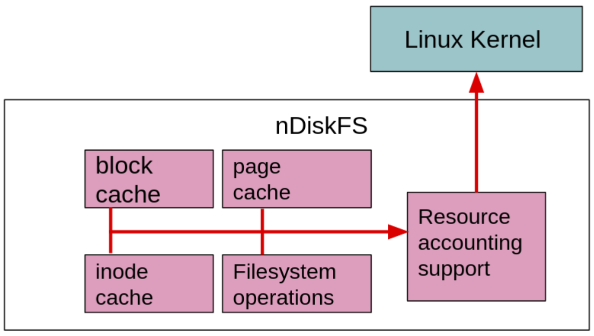

FIGURE 3.4: nDiskFS modular resource accounting

Memory usage accounting: Resource accounting module accounts memory usage and provides accounting statistics related to access in the nDiskFS. All the subsystems in the nDiskFS interact with resource accounting module fulfill their memory needs. These subsystems request the resource accounting module for allocation of memory which internally diverts the requests to the kernel. Some subsystems like pagecache, block-cache, inode-cache initialize its internal data structures through this module. Resource accounting module internally creates slab caches for the nDiskFS subsystems memory needs. This module accounts all the memory usage by the subsystems and exposes them as `sysfs` variables in the user space. This design provides robust support to debug memory leaks in nDiskFS. Some example variables are: `kmalloc_32`, `kmalloc_64`, `kmalloc_greater_64`, `view_page_cache_used`, `view_kmalloc_size_used`.

Other accounting statistics: nDiskFS has many accounting statistics that are scattered across the subsystems. Resource accounting module brings all the statistics to single place through which entire accounting is done. For subsystems like page-cache, this module maintains statistics at block, file and filesystem granularity. Some block level statistics are `num_reads, num_writes`, `block_type`, `sequence_number`. Example file level statistics are `content_type`, `num_blocks`, `num_dirty`, `file_sequence_number`, `max_write`, `max_read`, `total_read, total_write` .

3.3 Pluggable policy framework

nDiskFS lacks of pluggable policy framework as we discussed earlier in Section 2.6. nDiskFS policies are tightly coupled with the pagecache management and these policies are designed as integrated part of the pagecache. Pagecache management policy and block placement policy are part of the pagecache design. Due to this, nDiskFS lacks the flexibility to support multiple policies. This project separates the policies with mechanism and creates the interfaces to support multiple policies. This pluggable policy framework gives nDiskFS to study multiple policies without bothering about internal design of the pagecache.

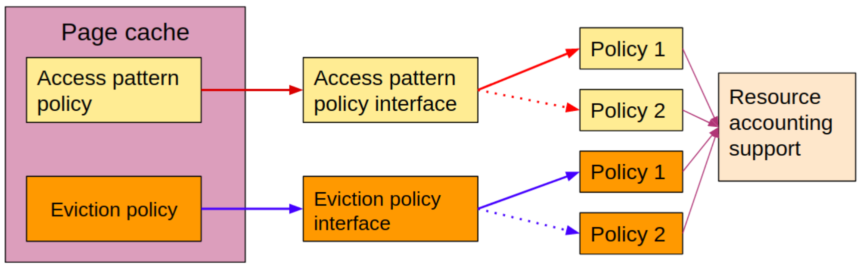
Page cache
Access pattern policy
Eviction policy
Access pattern policy interface
Eviction policy interface
Policy 1
Policy 2
Policy 1
Policy 2
Resource accounting support

FIGURE 3.5: nDiskFS pluggable policy framework

Pluggable access pattern policy: Access pattern policy analyzes the pattern of the blocks and migrate the suitable blocks to SSD. This policy is part of the every file pagecache. While file being accessed by the applications, this policy selects the blocks to migrate to SSD. The proposed module creates interfaces for this policy. Any policy uses this interface by registering the interface operations. While file pagecache being created, nDiskFS registers access pattern policy by registering the interface operations. For every file access, pagecache calls to the interface operations and eventually it calls the underlying policy as shown in the Figure 3.5. Underlying policy selects the blocks to migrate using the statistics in the resource accounting module as shown in the Figure 3.5 This enables to plug any policy without bothering about changing internal design of the nDiskFS.

```
static struct access_pattern_operations const ndfs_pattern_ops = {
    .block_accessed = pattern_block_access,
    .file_accessed = pattern_file_access,
    .analyze_pattern = check_pattern,
};
```

`block_accessed` is a interface to record the block access pattern by any policy implementation. Similar callbacks can be provided to capture the file access pattern. The page-cache layer invokes `analyze_pattern` operation to check the status of the file/block and its suitability for migration.

Lock debugging framework nDiskFS has many concurrent execution contexts sharing a lot of data structures. To debug the locks, we created a framework that tells at which function and line number lock has been acquired. We extend this lock debugging framework to any lock in the filesystem based on compile-time configurations.

Pluggable eviction policy: nDiskFS accumulates pages in the pagecache and if it crosses a certain threshold or on an explicit kernel request (through the shrinker API), pages are evicted from the pagecache. Eviction policy and pagecache share the same structure as shown in Figure 3.5. This project creates framework to support multiple policies by exposing the generic eviction operations to the policies. Currently nDiskFS has pluggable eviction framework, but this framework not tested with any eviction policies.

Chapter 4

Evaluation

We evaluated the correctness of nDiskFS for database applications and compared the performance against Ext4 filesystem by running `TPC-C` benchmark. Further, we gathered application initiated file data synchronization (using `fsync`) statistics to evaluate correctness of the file synchronizations in the nDiskFS. Finally, we demonstrate the resource module accounting statistics by showing some sample metrics collected by the module.

4.1 nDiskFS Support for MySQL Database

We added necessary functionalities to nDiskFS to support database engines and performed the correctness testing using MySQL [8], a popular open source database. To test the nDiskFS correctness for database applications, we required a concrete multi-threaded database benchmark suite. `OLTP-Bench`[9, 10] is an open source database benchmark framework which provides built-in workloads that can execute on a database schema loaded onto a MySQL server. OLTP-Bench allows a user to control over the workload by providing configurations for database size, time of the workload execution, transaction mix etc. `TPC-C` [11] is built-in multi-threaded benchmark in the `OLTP-Bench` that creates a database schema and runs concurrent transactions using MySQL server.

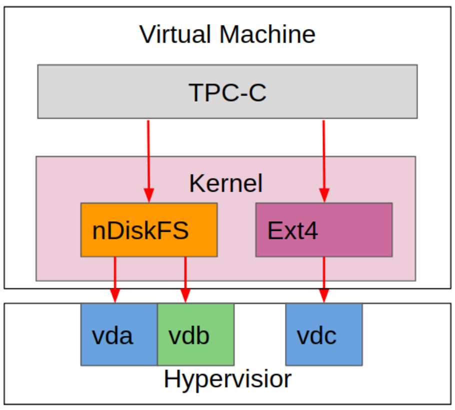

FIGURE 4.1: Virtual machine setup to evaluate nDiskFS, Ext4

Experimental setup: We evaluated the nDiskFS performance using a virtual machine. Three virtual disks, each of size 20GB, are exposed to the virtual machine. We used two disks to create nDiskFS, assuming one of them to be HDD and another to be SSD as shown in Figure 4.1. The third disk was formatted with Ext4 filesystem. For this experiment, we created nDiskFS filesystem using a logical partition created from the HDD and SSD devices before mounting the nDiskFS filesystem.

`TPC-C` is an online transaction processing(OLTP) benchmark that runs on MySQL server. `TPC-C` is multi-thread workload that creates database and does concurrent transactions on the database. User is allowed to configure `TPC-C` scale factor (database size) and running time of the workload. We changed the MySQL data storage to nDiskFS filesystem mount path to evaluate the performance and correctness of nDiskFS. Further, we compared nDiskFS performance against Ext4 performance by hosting the MySQL data store in the Ext4 filesystem. More about `TPC-C` benchmark execution is given in Appendix A

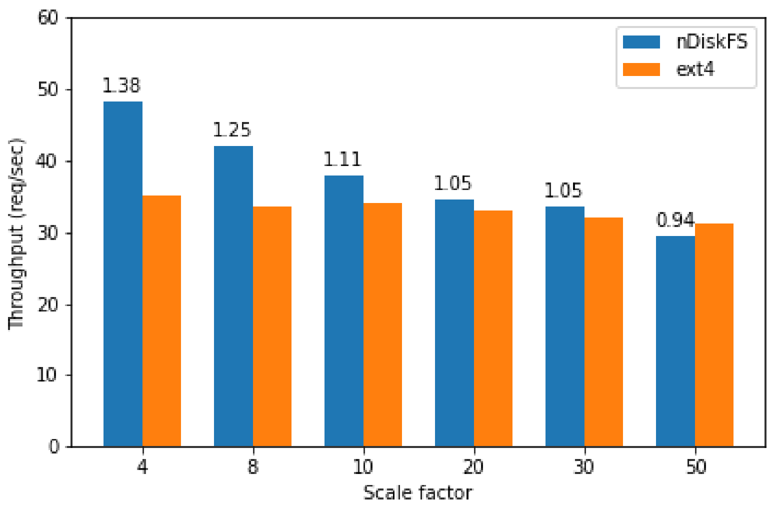

FIGURE 4.2: Comparison of nDiskFS and Ext4 performance for TPC-C benchmark executed with various database sizes on a VM with 4GB memory

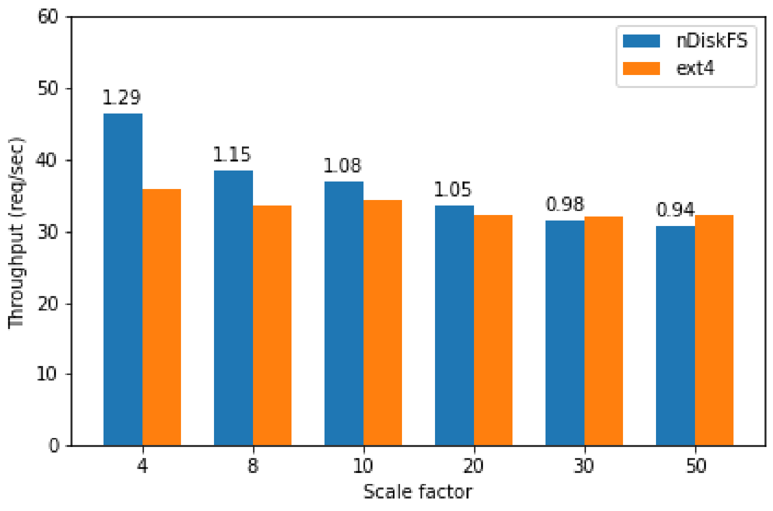

FIGURE 4.3: Comparison of nDiskFS and Ext4 performance for TPC-C benchmark executed with various database sizes on a VM with 2GB memory

We performed the experiments for two different VM memory configurations i.e., 2GB and 4GB. Above experiment was executed in different VM memory configurations to evaluate the correctness of nDiskFS for different memory availability, specially to check for memory leaks. We executed `TPC-C` workload with various scale factors for ∼25 minutes to take the readings for both VM memory configurations. The results presented here is an average over three executions. Figure 4.2 and Figure 4.3 show the throughput (requests/sec) of the `TPC-C` benchmark in the filesystems for different scale factor. We compared the nDiskFS performance with Ext4. nDiskFS resulted in 1.38x throughput compared to

Ext4 for smaller scale factors. However, for larger scale factor Ext4 performed better than nDiskFS. For example, nDiskFS resulted in 0.94x throughput compared to Ext4. This experiment concludes that for bigger scale factors, nDiskFS doesn't perform well because of poor page-cache management. This observation can be used to design scalable page cache schemes in future.

4.1.1 File Synchronization Operations

nDiskFS resulted in similar performance for both VM configurations as shown in Figures 4.3 and 4.2. To further evaluate the correctness of the file synchronization operations, we present the collected file system level statistics as explained below. nDiskFS syncs the file data in two ways—*(i)* forced synchronizations initiated by the application *(ii)* synchronization using the dirty syncer thread in nDiskFS. Dirty sync thread always executes in the kernel mode and it syncs the dirty pages whenever it crosses certain thresholds in nDiskFS. On the other hand, applications can force a sync operation in nDiskFS to enforce data consistency, especially required by database workloads to correctly implement transaction semantics. We collected the application initiated forced synchronization statistics to evaluate the correctness of the file synchronizations in the nDiskFS.

we executed the `TPC-C` benchmark for around 25 minutes with a scale factor 50 and collected statistics related to sync operations.

Operation count	**VM memory size = 2GB**	**VM memory size = 4GB**
#`fsync` calls	151467	145964
#data blocks synced	12050667	12244537
#meta-data blocks synced	6849	6209
#inodes synced	1456964	1449898
#bitmap blocks synced	1955	1959
#file blocks synced	12057516	12250746

Table 4.1: `TPC-C` nDiskFS File synchronizations statistics with VMs configured with memory 2GB and 4GB

Table 4.1 shows the file synchronizations statistics of nDiskFS while running the `TPC-C` benchmark. All the forced synchronization statistics shown similar numbers without respecting the memory configuration of the VM. Above statistics demonstrates the correctness of the file synchronization operation in nDiskFS as the number of different sync operations originated from the user space is similar. By implication, as the application layer invocations are same irrespective of VM memory size, we conclude that these experiments show the correctness of the nDiskFS for MySQL database.

4.2 Resource accounting and pluggable policy framework

We have designed resource accounting manager to provide memory usage and other filesystem accounting statistics. Here we provide demonstration of memory usage statistics in resource accounting module. These reading are taken in nDiskFS while running `TPC-C` benchmark for around 3 minutes with 4GB VM configuration and a scale factor of two.

Number of allocations in nDiskFS

```
 # pages   125895 ~ 491MB
 # <32B allocations   126197
 # <64B allocations     126196
 # >64B allocations  126527
```

Memory usage(in Bytes) for slab caches in the nDiskFS.

```
view_cache_page_used 13093080
view_cache_block_used 12085920
view_inode_cache_used 266976
view_kmalloc_size_used 15319656
```

We have designed a pluggable policy framework for block placement policy and eviction policy. Currently, we have an implementation ready but not tested extensively. For example, eviction policy framework is not tested with any policy. While block placement policy was tested with generic policy in the nDiskFS but this can not be used to conclude the correctness of this framework. We believe that, the generic policy framework is required to be tested with block-level eviction policies like LRU in future.

Summary: We evaluated the correctness of nDiskFS for MySQL database in virtual machines with 2GB and 4GB memory configurations. nDiskFS performance was compared against Ext4 filesystem by running `TPCC` benchmark. Further, we evaluated the correctness of the nDiskFS file synchronizations by collecting the application initiated forced synchronization statistics. We demonstrated the resource accounting manager statistics by collecting some memory usage metrics collected in nDiskFS. Multiple policies need to be designed for both block placement policy and eviction policy to properly evaluate the correctness of the proposed pluggable policy framework and left as future work.

Chapter 5

Conclusion and Future Work

As part of this project, we enabled nDiskFS support to host database engines (e.g., MySQL) and proposed a modular resource accounting framework for efficient resource management in nDiskFS. This project extended the nDiskFS to collect accounting statistics and implemented a pluggable policy framework to design efficient placement and page-cache management policies. Finally, we evaluated correctness of nDiskFS for database applications by comparing the TPC-C workload performance against Ext4.

nDiskFS still lacks proper policies for block placement to migrate data from HDD to SDD and page-cache management policies to evict the pages from the page-cache. nDiskFS is not fully POSIX standard compliant and changes are required in the operations layer of the filesystem to make nDiskFS POSIX compliant. nDiskFS requires additional policy research to deal with SSD endurance problem. Current version of nDiskFS does not support migration of blocks from the SSD to the HDD. Finally nDiskFS does not have any sophisticated file system consistency support like journaling to handle crashes. While the POSIX sync APIs are implemented, in future strict consistency designs may be considered.

Appendix A

TPC-C Benchmark Testing

Mounting nDiskFS filesystem: First build (`Makefile`) the nDiskFS source, creates `ndiskfs.ko` and inserts the `ndiskfs.ko` module into kernel. Setting up the disks `/dev/vdb /dev/vdc` and create `vg0` volume group to use both HDD and SDD. Create file system by using `ndfs_mkfs` program in the `vg0` volume group. After creating the filesystem, ndiskfs can be mount to any directory and run benchmarks.

All this above process can be done through the nDiskFS scripts `/nDiskfs/tests/test.sh`, `/nDiskfs/tests/start.sh`. We can add/comment/uncomment the default benchmarks running in the `/nDiskfs/tests/start.sh` file.

```
go to the nDiskFS source directory
$ cd ~/nDiskFS

building nDiskFS source
~/nDiskFS $ make

to build the nDiskFS, create the filesystem and mount it to /mnt/ndfs
~/nDiskFS $ ./tests/test.sh -nc -nt
```

After completing the above steps, nDiskFS filesystem is created and mounted in /mnt/ndfs directory.

Running `TPC-C` benchmark in nDiskFS: To run `TPC-C` , first we need to install `oltpbench, MySQL` in the system. `MySQL` server creates database and runs transactions in the nDiskFS filesystem to test the nDiskFS filesystem.

```
If MySQL server is running in the system, to stop MySQL server.
$ sudo /etc/init.d/mysql stop

replicate existing database directory to the nDiskFS
$ sudo rsync -av /var/lib/mysql /mnt/ndfs/

Change the datadir to nDiskFS filesystem.
edit MySQL configuration file (/etc/mysql/mysql.conf.d/mysqld.cnf)
...
datadir = /mnt/ndfs
...

Restart MySQL server
$ sudo /etc/init.d/mysql start

go to oltpbench source directory
$ cd oltpbench

Before going to run TPC-C benchmark, create "tpcc" database in MySQL server.
oltpbench $ mysql
>>  create database tpcc;

To load tpcc database, execute transactions in the nDiskFS.
oltpbench $ ./oltpbenchmark -b tpcc -c config/sample_tpcc_config.xml --create=true
           --load=true --execute=true -s 5 -o outputfile

To change the scale factor of database, workload running time we have to edit
config/sample_tpcc_config.xml. Results are stored in oltpbench/results folder.
```

Bibliography

[1] Kadve, Anagha, (2016). Trade Of Between SSD and HDD. International Journal for Research in Applied Science and Engineering Technology (IJRASET), pages 473-475.

[2] agrawal N, prabhakaran V, wobber T, davis J D, , manasse M, and panigrahy R.(2008) Design Tradeoffs for SSD Performance. In Proc. of USENIX ATC (2008), pages 57-70.

[3] S V shunmuga Sundar, (2019). nDiskFS: A Filesystem for hybrid storage. In *IIT Kanpur, M.Tech Thesis*.

[4] Jian Xu, Steven Swanson, (2016). NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323-338.

[5] Shengan Zheng, Morteza Hoseinzadeh, Steven Swanson. (2019). Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks. *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 207-219.

[6] Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau, (2018). Operating Systems: Three Easy Pieces. In *ArpaciDusseau18-Book*.

[7] Megiddo, Nimrod and Modha, Dharmendra S. . (2003). ARC: A Self-Tuning, Low Overhead Replacement Cache. Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST 03), pages 115-130.

[8] MySQL database management system. https://www.mysql.com/.

[9] Difallah, Djellel Eddine Pavlo, Andrew Curino, Carlo Cudre-Mauroux, Philippe, (2013). OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases Proc. VLDB Endow., pages 277–288.

[10] OLTP-Bench https://github.com/oltpbenchmark/oltpbench/wiki.

[11] TPC-C On-Line Transaction Processing Benchmark http://www.tpc.org/tpcc/.