A new I/O subsystem for modern data-intensive workloads

Aldo Tali
ge59faz@mytum.de
Technical University of Munich
Munich, Germany

ABSTRACT

In the traditional CPU-centric architecture the processing unit (e.g. CPU) is separated from the storage devices (DRAM, HDD, SSD). As a result, any data processing requires constant moves of data between the processing unit and the storage device. Although CPU-centric architecture was the dominant architecture for many years, the continuously increasing volume of generated data creates new challenges. Modern (ML/AI etc.) and traditional (databases etc.) data-intensive workloads experience significant performance bottlenecks, due to the constant move of big volumes of data between the storage device and the processing unit. In an effort to provide a solution for the above problem, researchers explore the idea of Near Data Processing (NDP). Small and flexible computation units, such as FPGAs or embedded processors are placed inside the storage devices. Therefore instead of moving the data, the computation is taking place closer to the data source. In that way, users can offload parts or entire applications to the processing units inside the storage device. Subsequently, the data movement is drastically reduced, decreasing the energy consumption and increasing the performance for both the applications and the overall system. The operations running in the storage devices can have direct access to the storage devices and the main processing units are decoupled from I/O requests, being able to execute other processes.

Even though NDP is not a new idea, and despite its benefits, NDP has not been integrated in computing systems yet. At first, the small computation capabilities of early storage devices could not meet the requirements of data processing. In the last decade, with the advances in low-power processing units, such as SoCs, FPGAs and ASICs, researchers created prototypes of storage devices with stronger computational power [1]. However, only very recently storage devices with strong computation power have been available on the market [2,3]. Consequently, there is limited available hardware and big diversity in both the computation units of each suggested architecture (ISA etc.) and the host integration interface. Therefore, researchers and engineers face limited accessibility and high complexity, complicating their effort to build a suitable solution for such devices. Most of the prior work, was more focused on proving the advantages of NDP and less on creating practical solutions. The majority of these projects, used a specific application (i.e. databases [4],) and explored, how NDP could help such applications. On the other hand, only a few projects [6,7] provide an interface that can be used for more generic workloads.

Our goal is to design, implement a new I/O subsystem, which can efficiently manage and provide a generic-enough interface for user applications. Furthermore, since such devices will equip the cloud servers, an important aspect of our proposal should be the support of multi-tenant environments. To the best of our knowledge, current solutions do not support such environments, creating further issues in the adoption of this new technology. Moreover, due to the diversity of the proposed architectures in NDP, we aim to provide a generic-enough solution, that will be easily adaptable in various architectures (FPGAs, ASICs, SoCs etc.).

In conclusion, we will analyze the prior work and discuss their applicability in virtualization technologies. Next, we will design and implement a new I/O subsystem which will enable task or function offloading of the data-intensive workloads to the processing unit located closer to the data. Finally, we will evaluate the performance and the compatibility of the new subsystem with existing libraries/frameworks/workloads that are widely used for data-intensive tasks.

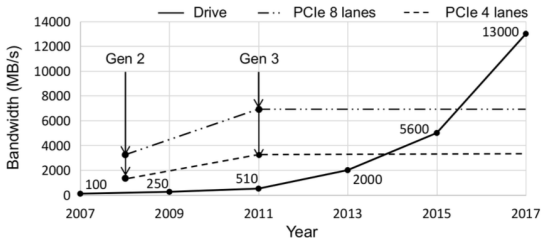

Figure 1: The bandwidth evolution of storage drives. Analysis taken from Ruan et.al

1 PROBLEM AND MOTIVATION

The advancement of big data technologies and analytics has created an unprecedented amount of workloads making compute the read bottleneck of learning [6]. The tech giant Meta, previously Facebook was warehousing at a rate of 600TB daily and storing about 300 PB of data in a data storage in 2014 [5][6]. The usual compute required that data to be moved fast in the inter process communication mechanisms. In 1965 Moore's Law stated that the number of transistors in a chip will double roughly every two years. The developments on silicon transistor technology has indeed led a revolutionary change in terms of compute power making data accesses the main bottleneck in the classical Von-Neumman architecture. Close to reaching the physical boundaries on how small transistors can get the researchers and big techs have to rely on making data moment faster both within the device and via a network or change the compute medium from generic compute (CPU) to specific compute (i.e FPGA).

TUM, Munchen, Aldo Tali
2022. ACM ISBN GR project.

Warehouse-scale profiling [11] give insights that the typical workloads in a data center have shifted towards data analytics making it one of the main challenges for system engineers both for load handling but also for processing. A massive and efficient analytical platform or system is therefore a hot topic for cloud providers and not only [6][7]. While CPU's have started to reach their peak improvement many new designs have tried to increase the I/O throughput of drive storage [19]. This is no surprise since taking drive speed forward is not only important for the data processing side but also for the overall in-memory computing.

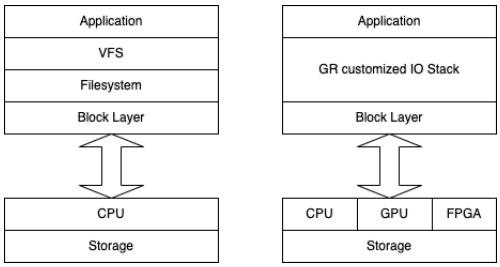

Figure 2: A classic architecture on the left, and our proposed architecture on the right.

Storage advancements have not been missing either however with the increase of storage speed i.e (user programmable SSDs as well as the two-level hierarchy of modern storage) the bottleneck was only then shifted to the I/o movements between host and storage interconnections. The data movement wall is the main reason why the performance is not being exploited at its finest. It is only natural therefore to think that instead of making the CPUs (i.e the compute) take all the storage to give a result, it would make sense to try and deliver the compute to the storage. This idea holds that it removes the data movement wall issue and all that is let to be dealt with is efficient building of the computational storage units, to offset going further away from the traditional architecture.

There are many different approaches that come up as a solution in this case such as task offloading and full leverage of in storage embedded ARM cores [5][6]. These systems however face the challenge of standardisation and adaptation since migrating old running infrastructure requires effort. On top of it Ruan et.al suggests that these systems typically suffer from three main challenges [12]:

- Limited performance or flexibility.
- High programming efforts.
- Lack of crucial system support.

The limited performance concern is attributed to the fact that many of the computational storage solutions are initially designed in a hardware customization that achieved stellar performance for specific tasks but the scalability to all type of loads is still questionable. By definition, targeting a specific workload and optimize for its performance will not enable better speeds on the general workloads [8]. The second concern is that since many of these solutions are not POSIX compliant they either offer or have to implement a standardized API in order for it to be used with ease. However transitioning to something completely new takes time especially for the industry. In the case that such API's are not available it is even worse since the developer then is left with the burden of managing all file pointers in system level. Lastly, the current work done on these type of computational storage devices is focused on the scenarios that need optimization which are highly singular whereas a typical analytic system includes a variety of processes and parallel work on the data [13]. A support for parallel data access, processing and protection is needed.

These challenges have led us to present a POSIX compliant novel I/O subsystem that will deal with task off loading in a different manner. The system is designed with the following principles in mind.

Efficient task computation. We design a system that aims to make compute time gains by skipping parts of the Linux stack when dealing with file storage and its processing. Offloading the operations from the general control of the Virtual File System to direct calls to the NVMe should increase the processing speed and the availability of compute, the latter indirectly.
System Abstraction. The subsystem should be POSIX compliant so most jobs can be shifted with relative ease to the NDP unit [3]. It goes without saying that we try to focus on keeping similar interfaces for the file read and writes with the twist of changing this kernel level logic. The changed logic takes care that there is no data movement but the system simply uses what it already has.

System support The interaction between the added modules and the POSIX environment or the kernel built ins should be clearly seen on the codebase. We aim on creating a file system that like ecryptFs encrypts and decrypts data as needed making the files obsolete to the users unless they have the hash key to the files.

We built some small prototypes to document the usage of the functions from the user space, the VFS kernel space and by bypassing the vfs on kernel level. In addition we tried to give an example implementation of read-write function calls in order to check if the encrypt-decrypt is a viable solution in the novel I/O subsystem. It supports multiple offloads and is compatible with any of the usual operations on linux that are POSIX compliant.

2 RELATED WORK

In this section we will discuss the availability on the market of several different approaches and the focus of each approach so far. The following subsections do not consider a full scale analysis on NDP, however they try tackle some of the other novel approaches that may be potential solutions to the problem mentioned above. These go in line with the motivation of this paper discussed on the previous section.

2.1 Current State of Data Intensive workloads

The Big Data trend is putting a strain on modern storage system as they have to provide for high-performance I/O accesses in order

to deal with the large quantity of data. The Von Neumann architecture moves data back and forth between computing (CPU) and storage (DRAM, NVM Storage). A big data volume inherently translates into a large number of I/O movements. With the definition of Moore's law and the recent developments in CPU's and GPU's the processing power has been doubling ever 1.5 years whereas disk movements have had very slow improvements, subsequently making data transfer the bottleneck in data intensive flows [8][9]. Some of the main drivers of these operations are not only data filtering and processing but also the emerging need for ML and DL workflows. Typically these loads focus on some type of spatial filtering usually on 2D scale to operate on images. These have lead to the need to suggest detachment from usual CPU compute and shift into accelerator based systems. The most common of these are GPU's and FPGA's along with SoC becoming increasingly more popular [10]. All these approaches however have one thing in common, offloading of some type of work to be able to compute and operate on a decentralized manner.

Typically data intensive workloads are evaluated on four different features referred to as the big four Vs:

- **Volume** Refers to the size of the data workload.
- **Velocity** Refers to how fast the data is coming is and going out.
- **Variety** Refers to how diverse is the number of incoming data sources.
- **Veracity** Refers to how trust worthy, consistent and complete the data is.

Most of the optimizations thus far on the data side have focused on tuning these criteria from he software frameworks like Hadoop, Spark or MPI. Typically the focus has been on building a distributed compute upon available devices to increase the processed volume, increase speed due to addition of a large number of processing devices, keeping the variety the same and compromising in a way for veracity give n that in distributed systems some of the consistency is lost by definition. While there is a large work in optimizing these parallel, distributed frameworks there is little consideration on using hardware specific solutions that go outside optimizing for CPU and NUMA effects.

2.2 Current State of System on Chip

Computational Storage is a type of near data processing (NDP) architecture that enables data to be processed within a storage device instead of being transported to the host CPU [1] [2]. The main advantages offered are that the host cpu can be unloaded to leave room for a cheaper CPU, decrease in data transfers increases performance, less energy consumption is obtained overall and lastly the data Center performance does not have to be dependent to having a fast network. However, adoption suffers from lack of a clear unified interface to deal with different types of ephemeral units and the results of each unit along with the security, usability and data consistency concerns. The current SoC's are limited to stateless processing where the access to persistent storage is mediated by the host OS that runs on CPU's [1][2]. The main need for data intensive workloads then becomes the read and write operations which for decades have been done in terms of fixed reads or writes managed by the system [2].

SoC is inherently a hard and difficult task therefore most of the focus in designing accelerators that are domain specific but allow for specifications of flows that make the hardware and software integration appear seamless. The addition of new capabilities is strictly limited by the engineering effort and the team sizes. As such the focus for SoC is on Open Source in the hope to make contributions that eventually standardize the workflows from the data processing perspective. One such approach is ESP that focuses on making a programmable grid that allows the architect decide the structure of SoC by mixing the tiles on a graphical user interface. The paradigm bases the flows in 4 main tiles: processor, memory and accelerator tile. These are strictly higher level solutions and as such seamless integration for older systems is not as straightforward.

2.3 Usability of Accelerators in ML/DL workloads

Read/Write operations are the foundations of any ML/DL workloads and therefore they make a good test case to test solidity of the application on it. Typical applications in here include User-Programmable SSDs, FPGA arrays for Computer vision and System-On-Chip coins the definition of embedded machine learning. The most straightforward implementation that considers both the hardware layer as well as the software layer is ESP4ML which is a system on chip design for ML workloads based on ESP. This required a runtime system on top of Linux hat takes the data-flow and translates it into pipeline of accelerators that are dynamically configured. The code generated by the compiler and the runtime is fully transparent making the load highly accessible both on a high level and on a low level. The main operations are loads and stores which translate on the typical read-write operations.

The development goes through similar lines of change and progress when it comes to databases. The most typical operations here are reads, filters and aggregation queries which can be offloaded and improve in execution time. Typical queries that can be offloaded to NDP compute are aggregation queries like:

```
SELECT  avg(metric1),
        max(metric2),
        sum(metric3)
FROM    my_table;
```

Listing 1: Aggregation query for task offloading

The operations defined here represent a lesser challenge in the synchronization of the loads since no other data than the ones at the near processing device is needed to come up to the result. An operation like sum or average is strictly suitable for task offloading on a parallel environment. This is the case for any task that does not require heavy synchronization since the compute can be easily be localized without the need to make any hardware changes.

3 DESIGN OF THE NEW I/O SUBSYSTEM

Linux organizes its file access layer via a POSIX compliant driver file API and it allows for maintaining and developing new systems to allow support for custom made subsystems within the kernel. All the operations are handled by a Virtual File Interface/Switch which acts a software layer that allows for creating the filesystem interface to user programs [20]. Its abstraction allows the implementation and coexistence of different file systems. This project focused on four main stages that we will go over.

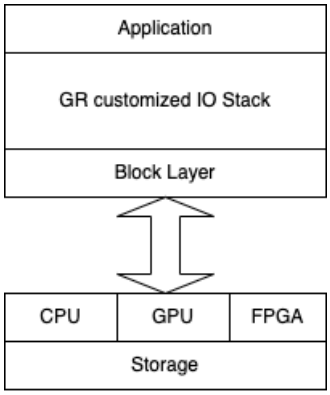

Figure 3: Structural unit of the new io subsystem

3.1 Defining a safe workspace

The linux ecosystem is built on a simple division between user space and kernel space. All the files and programs running on top of the system make up the user space whereas the system calls in its entirety make up for the kernel space. The classical architecture of a computer has defined that it is critical for both these spaces to communicate in order for the system to work. User space programs make function calls that compilers translate to system calls that the kernel executes. In a sense the user space provides with the full ability to make use of the device run and the capabilities of the hardware by communicating with the kernel's interface.

The kernel on the other hand provides the abstraction needed to the internal structures, ports, hardware and enhances the elements of security. Building a new I/O subsystem strictly required the use of the latter space [14][18]. It goes without saying that one would not want to mess around with the drivers of the system and make bit level file changes, to avoid the risk of failures on device level. The POSIX library provided by the kernel comes to use to alleviate these problems. Linux Kernel is extensible via the POSIX library but still runs the risk of having bad addressing or general systematic failures on new kernel builds. This paper made use of QEMU as the Hypervisor of choice in order to work on the new kernel level builds and modules loaded. The typical operation in this case included the creation of a block that contains the rootfs, a mount point directory for the system and the linux image that contains the newly developed or the changed kernel.Then on each new development and kernel built, the root image wold be resupplied to QEMU making it easier to interact in a fail proof way.

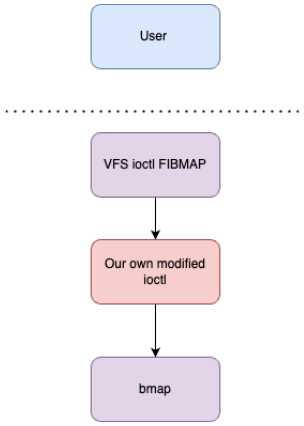

Figure 4: Passing ioctl operations over to the stack.

3.2 Defining a generic system call

A system call is the entry point by which a user space program requests the kernel to do some low level operation on its behalf. A manipulation of strings or integers is considered to be strictly a user-space program whereas operations on devices, files or processes are part of the kernel space. The added level of security on the kernel side is that unlike the user level program it does not jump to some part of the code or the library. It first needs to ask the CPU to shift its processing to Kernel mode and then go to a statically hard set location to make the call.
Implementation of a syscall requires the update of the syscall table which is used by the system to refer to the available calls [14][16]. Besides the provision of the new entry to the syscall table, the definition in the kernel is made by the predefined macros. We added one such call of our own to test the setup. We used the predefined macro "SYSCALL_DEFINEN" which takes one argument being the name of the system call and N-1 arguments to be used by the call itself. This setup allows the kernel to be built and provide support for the new syscall functionality. The call to the functionality is done via the respective systable number gotten from the initial step. In a simpler sense the kernel side demands that a call is first declared on its availability table list, then added dynamically to the running kernel then allow for the general run to make sure that the new call is correctly integrated.

```
int main(){
    int block_cnt;
    int i, block, res;
    struct stat statBuf;

    int fd = open(fileName, O_RDONLY);

    struct stat buf;
    fstat(fd,&statBuf);

    int total_size = statBuf.st_size +
         statBuf.st_blksize;
    block_cnt = (total_size - 1)/statBuf.st_blksize;


    printf("The block count is %d %ld %ld\n",
        block_cnt, statBuf.st_size,
        statBuf.st_blksize);
```

```
    for (i = 0; i< block_cnt; i++){
        block=i;
        ioctl(fd,FIBMAP, &block);
        printf("%3d %10d \n" , i , block);
    }

    return 0;
}
```

Listing 2: User Space Implementation of stats.

3.3 Connecting file stats via Kernel

A system call needs to provide some functionality that is otherwise not reachable by system itself. We Continued with given provisioning of the system call for the implementation of the stat terminal command in order to identify the underlying structure of the file read/write operations. The operations on the kernel side are handled by the ioctl utility which is also visible in the filefrag terminal command or the filefrag module. The user space makes use of the FIBMAP and FIEMAP ioctls. The functionality of the calls are handled largely by the virtual file system. We replicated the address mappings of the file by creating a loadable module into kernel that transitions from user space to kernel space and makes use of an address space struct to call the bmap operation which is the underlying unit for filegrag and stats. Bmpa allows for the mapping of the file into blocks. This is largely unsafe considering that the virtual file system already has an equivalent call, the vfs_read [15]. This part allowed us to be able to access from kernel perspective the size of the file, the number of blocks, the size of the block etc. This became the basis for the initial setup of our loadable read/write modules.

```
int res;
struct kstat stater;
struct file *f;

int myblk,block_cnt,i;
sector_t block;
mm_segment_t security_old_fs;

security_old_fs = get_fs();
set_fs(KERNEL_DS);

res = vfs_stat((const char __user *)fileName,
    &stater);
f = filp_open(fileName, O_RDONLY, 0);

int total_size = statBuf.st_size + statBuf.st_blksize;
block_cnt = (total_size - 1)/stater.blksize;

pr_info("The block count is %d %ld %ld\n",
    block_cnt,stater.size, stater.blksize);

for (i = 0; i< block_cnt; i++){
    myblk=i;
    vfs_ioctl(f,FIBMAP,&myblk)
    pr_info("%3d %10d \n", i, myblk);
}

set_fs(security_old_fs);
```

Listing 3: The use of virtual file system to simulate the stats call.

3.4 Redefining Read/Write operations for the new filesystem

A file stat gives the information of the high level nature of the file (i.e the block count, the block size etc). We went further to define modules that when integrated with the running OS would redirect the read/write operations from the virtual file system, directly to the filesystem. We defined that when inserted this would be defined as a new filesystem which gets handled by the vfs. This requires replacing the respective vfs_read and vfs_write to GrFsRead and GrFsWrite functions. This way once the GrFs is mounted any 'cat' like operation would be directly forwarded to the GrFsRead instead of the VFS read operation.

```
ssize_t grFileWrite(struct file *file, const char
    __user *user_buf,
        size_t count, loff_t *ppos)
{
    printk("write grfs files");
    mm_segment_t security_old_fs;
    ssize_t ret;

    ret = 0;
    security_old_fs = get_fs();
    set_fs(KERNEL_DS);

    ret = vfs_write(file, user_buf, count, ppos);

    set_fs(security_old_fs);


    return ret;
}
```

Listing 4: Passing from user level to kernel level in the write setup.

In order to keep the scope of the project inline we decided to simulate an ecrypt decrypt operation on the GrFs read/write calls. That is a GrFs write takes an input string and encodes the message in a format that is unrecognizable to the user before writing it to the output file. A GrFs read on the other hand takes as input the content to be read and decodes it to a format that is meaningful to the caller. Similar operations can be handled in the case of data filtering or built-in data intensive work.

```
static struct file_operations GrFS_file_operations = {
    .open = grFileOpen,
    .read = grFileRead,
    .write = grFileWrite,
};
```

Listing 5: File operations reset.

3.5 Stacked file systems

Linux allows for extension of its core kernel connection to the file system and redefinition of the interaction with the files, along with the addition of new calls and processes. The way it does this is by structuring its interface in two main layers, the user layer and the kernel layer. The abstraction level between the user level (request service layer) and kernel level (system call layer) is called the virtual file system or VFS. It is sandwiched between the two layers. It is a helpful design since the interface allows users to program their programs in a file system-agnostic way. In turn this

allows developers to also develop their own filesytem and attach it to the VFS without the need to make changes in the user space. The VFS is able to offer the abstraction to the user for any attached system since it defines its operations as an interface that can be attached to the main. In fact the operations of the current running system can also be overridden by custom functions although this is not advisable due to the severity of failures that one may encounter. Regardless the linux kernel allows the file system API to define a new file system, give it a name and define the most commonly used operations. A mounting of the new filesystem to a specific location allows the VFS to call the function's respective codebase for the available operations committed on the new system. Figure 2 gives a broader understanding how this happens in practise.

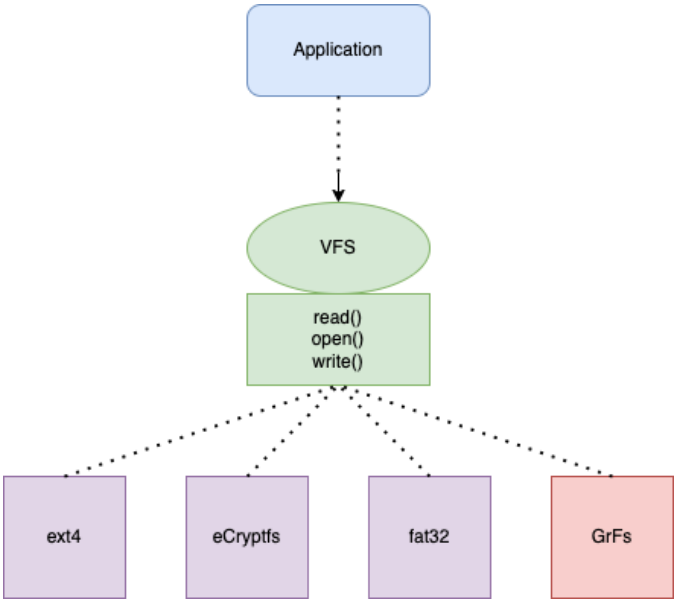

Figure 5: Setup of stacked file systems. ext4 and fat32 are different storage systems, eCryptfs is a stacked system we base our GrFs.

4 IMPLEMENTATION

A kernel module defines the underlying brick of this whole implementation process. The positive sides for modules is that they are pieces of code that are dynamically inserted/loaded onto the kernel after it has been booted. We made use of two main functions the 'init' and 'exit', defining the module linkage and module detach. In order to make the read/write work the linux foundation makes use of an already build interface Wrapfs which simply replaces the linux kernel calls to custom ones and acts as a layer of simply propagating work downstream to a lower level. This inherently means that two implement these functionalities one needs to provide both file operations and directory operations so as not to incur access in wrong/bad addresses. The Linux Filesystem provides several abstractions which we have found useful in this project including dentry, inode, file object and super operations. The listing is outlined here as follows.

Dentry is the internal linux structure used for directory operations and it provides the link between inodes and filenames. Its most common operations are the make root and adding a file to dentry cache. The latter is a cache that is used by most of the VFS calls like chmod, open or stat. The dentry cache (dcache) acts a fast lookup. In their life cycle dentries are most likely to be kept in RAM and never saved into disk. Regardless, computers are not always able to fit all dentries in RAM making some cache bits go missing or over spill [17]. The cache itself in short terms is meant to keep the view of the entire file space and to lookup at inodes.

Inode is an object that can be best described as a single dentry or filesystem objects in the form of FIFO, files, directories etc. They can either be on disk, the case for GrFs as it is a block filesystem, or they can be in memory also known as the pseudo filesystems. On the case of block device systems the changes committed on the inode are copied to memory then written back to disk. An inode it looked up by the lookup method on the VFS side. In linux this is represented and made use of by the internal provided structure struct inode. The main operations provided are i_sb,i_rdev, i_no, i_blkbits, i_mode, i_size, i_mtime, i_nlink, struct file_operations and icount which deal with the suberblock, the mounted device, the inode number, number of digits used for block size, access rights, the size in bytes, access to time, number of entries that use the node, the number of blocks and the file mappings for read/write.

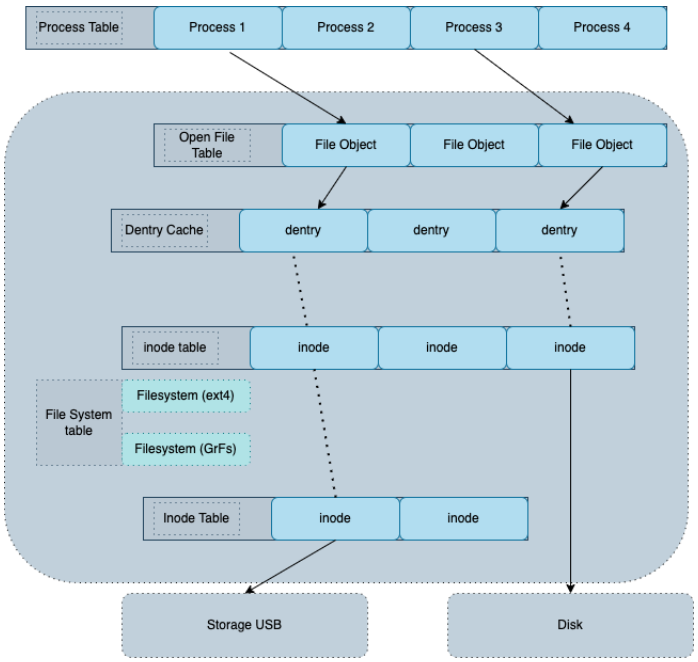

Figure 6: A virtual file system

File object is the file structure allocation on the kernel side. This is needed since opening a file a file descriptor is needed to refer to it. Typically it follows an initialization with a pointer to the dentry and the set of file operation member functions. The operations described here are VFS managed or VFS operations that are initiated by using user space file descriptors called afterwards. An open file will keep the dentry in use as well as the VFS in use.

Super operations make the same operations as the usual inode operations, however they are focused on making the initial organizations as well. A typical example would be the registering and un-registering of the filesystem during a mount. Other operations used when working with inode are alloc-inode, write-inode, evict_inode etc.

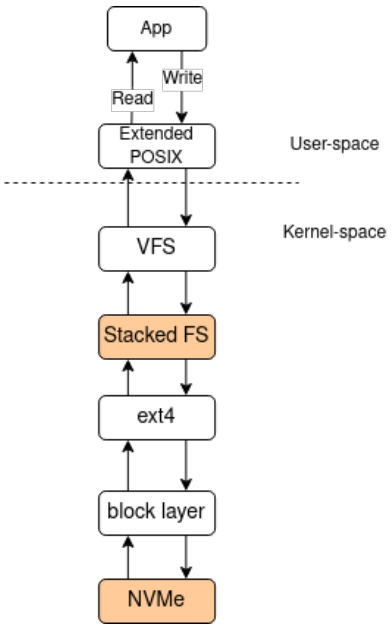

Figure 7: Desired Architecture.

5 FUTURE WORK

The implementation provided above is faulty when dealing with the recursive part of the VFS read and writes. Nevertheless the connectivity and integration of the File Systems API works accordingly. This implementation did not cover the encryption and decryption part so the future work to be done involves integrating a safe way to write encrypted files and also decrypt them. An ideal system should not only handle the read and write but also the integration of other calls and extension of the existing read writes to include extra parameters so that the customized I/O subsystem supports the default linux calls but also ads to functionality.

6 REFERENCES

[1] Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. 2020. Cosmos+ OpenSSD: Rapid Prototype for Flash Storage Systems.:https://doi.org/10.1145/3385073

[2] NGD Systems heralds in-situ processing for NVMe SSDs https://ngdsystems.com/ngd-systems- heralds-in-situ-processing-for-nvme-ssds

[3] Xilinx/Samsung SmartSSD https://www.xilinx.com/applications/data-center/computational- storage/smartssd.html

[4] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. 2016. YourSQL: a high-performance database system leveraging in-storage computing. https://doi.org/10.14778/2994509.2994512

[5] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. 2019. Cognitive SSD: a deep learning engine for in-storage data retrieval. In <i>Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference<USENIX ATC '19. USENIX Association, USA, 395–410.

[6] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong- Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. 2016. Biscuit: a framework for near-data processing of big data workloads. https://doi.org/10.1145/3007787.3001154

[7] Antonio Barbalace, Martin Decky, Javier Picorel, and Pramod Bhatotia. 2020. BlockNDP: Block-storage Near Data Processing. In <i>Proceedings of the 21st International Middleware Conference Industrial Track https://doi.org/10.1145/3429357.3430519

[8] Giri, D., Chiu, K.-L., Eichler, G., Mantovani, P.,&amp Carloni, L. (2021). Accelerator Integration for Open-Source SoC Design. Accessed 11 Oct. 2022.

[9] Heinz, C., & Koch, A. (2021). Supporting on-chip dynamic parallelism for task-based hardware accelerators. Applied Reconfigurable Computing. Architectures, Tools, and Applications, 81–92. https://doi.org/10.1007/978-3-030-79025-76

[10] Lukken, C., Frascaria, G., & Trivedi, A. (2021). ZCSD: a Computational Storage Device over Zoned Namespaces (ZNS) SSDs.

[11] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a Warehouse-scale Computer. In Proceedings of the 42Nd Annual Interna- tional Symposium on Computer Architecture, ISCA '15, pages 158–169, New York, NY, USA, 2015. ACM.

[12] Ruan, Z., He, T., & Cong, J. (2049). INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive.

[13] Barbalace, A., & Do, J. (2021). "Computational Storage: Where Are We Today?"

[14] Brennan, Stephan. Write a System Call, 14 Nov. 2016, https://brennan.io/2016/11/14/kernel-dev-ep3/. Accessed 11 Oct. 2022.

[15] Get Physical Disk Block Number of a given File ITmyshare, https://sites.google.com/site/itmyshare/shell-scripting-programming/c-useful-examples-for-admin/get-physical-disk-block-number-of-a-given-file.

[16] Adding a New System Call Adding a New System Call - The Linux Kernel Documentation, https://www.kernel.org/doc/html/v4.10/process/adding-syscalls.html?highlight=system+calls.

[17] Awamoto, S., Focht, E., & Honda, M. (2021). Designing a Storage Software Stack for Accelerators.

[18] Seshadri, S., Gahagan, M., Bhaskaran, S., Bunker, T., De, A., Jin, Y., Liu, Y., & Swanson, S. (2014). Willow: A User-Programmable SSD.

[19] A Comprehensive Memory Analysis of Data Intensive Workloads on Server ... http://mason.gmu.edu/ spudukot/Files/Conferences /Mem_Subsys18.pdf

[20] Zadok, E. (2003, May 1). Writing Stackable Filesystems. Retrieved October 12, 2022, from https://www.linuxjournal.com/article/6485