Tags: A Unifying Primitive for the Storage Data Path

Weisu Wang

FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

TAGS: A UNIFYING PRIMITIVE FOR THE STORAGE DATA PATH

By

WEISU WANG

A Dissertation submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

2019

Weisu Wang defended this dissertation on November 12, 2019.
The members of the supervisory committee were:

An-I Andy Wang
Professor Directing Dissertation

Adrian Barbu
University Representative

Zhenhai Duan
Committee Member

Peixiang Zhao
Committee Member

Weikuan Yu
Committee Member

Zhi Wang
Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the dissertation has been approved in accordance with university requirements.

ACKNOWLEDGMENTS

I would like to express my sincere appreciation for the great help I received from the committee and faculty members: my advisor Dr. Andy Wang, admin specialist Yu Wang, department chair Dr. Xin Yuan, Dr. XiuWen Liu, Dr. WeiKuan Yu, Dr. Zhenhai, Duan, Dr. PeiXiang Zhao, Dr. Zhi Wang, and others. They provided academic advice, warm emotional support, and understanding during difficult times in my life. My research group members, Sarah Diesburg, Shuanglong Zhang, Leah Rumancik, Bobby Roy, and others also have helped me significantly. Without them, I would not have survived to complete my study and research here.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The legacy storage data path is largely structured in black-box layers and has four major limitations: (1) functional redundancies across layers, (2) poor cross-layer coordination and data tracking, (3) assumption of high-latency storage devices, and (4) poor support for new storage data models. Many works have been proposed to resolve one or more aspects of these limitations, but none of them have addressed all four limitations.

We introduce Tags, a unifying primitive that can be used throughout the storage data path. This white-box approach enables all storage layers to coordinate and track data using shared data structures that are constructed through the Tags API. Our case studies show that by eliminating redundant services, our Tags-based key-value store can outperform LevelDB by 10-220% when inserting and deleting 100-byte key-value pairs. We also built a Tags-based file system (TagFS) and differentiated storage services to demonstrate the usability, extensibility, and robustness of Tags. In addition, we built per-file secure deletion via TagFS to show data-path-wide coordination and data tracking.

CHAPTER 1

INTRODUCTION

1.1 Overall Status Quo and Motivation

The storage substrate of an operating system has been a system-wide bottleneck for decades, and this is likely to remain the case for the foreseeable future. Improving the storage data path, therefore, is likely to result in improved performance for the overall system.

The legacy storage data path is largely disk-centric and structured in layers. However, hard disk drives (HDDs) are being routinely replaced by low-latency solid-state disks (SSDs), which have very different characteristics. Applications also demand more coordination and control across storage layers (e.g., data tracking). Many approaches have been proposed to address these limitations, but they have their own limitations. These driving forces make us rethink how to preserve the advantages of layering while granting more cross-layer control and how to design a data model to provide more support for different emerging storage media.

1.2 Background

A computer storage system consists of storage devices and a corresponding software stack or the storage data path. The legacy storage data path is designed for mechanical storage media (e.g., hard disks or HDDs). However, these mechanical storage devices are rapidly transitioning to non-volatile memory-based media (i.e., NVM), including NAND flash memory (the most prevalent), Phase-Change RAM (PCRAM), and Magnetic RAM (MRAM), etc [Bez and Pirovano 2004; Shin 2005]. NAND flash memory-based drives are also referred to as SSDs, which have different physical interfaces (e.g., SATA or PCIe) and form factors (e.g., NVM-express or 2.5"). As shown in Figure 1.1, the number of HDDs that were shipped steadily decreased, from about 460M units

in 2012 down to 400M units in 2017; in the same period, the number of SSDs rose from about 30M units to 240M units, almost eight times more. Projection shows that the trend may continue. Therefore, we are embarking on an era in which SSDs are likely to overtake HDDs in the foreseeable future.

With the rise of SSDs, the legacy design of disk-based storage data paths (Section 1.5.1) becomes increasingly limiting. Besides low-latency and high-bandwidth, the NAND flash device has other features regarding reads and writes that affect access patterns. The recent Linux storage software stack components (Figure 1.2) reflect various improvements to accommodate SSDs.

In addition to storage hardware, more and more diverse requirements for storage have emerged, such as object-database storage and distributed storage. It is difficult for legacy data path elements to fit into such architecture. There are new popular data models beyond files, such as key-value stores.

The following sections highlight the important characteristics, limitations, and existing SSD-based improvements of the legacy storage software stack. Section 1.3 provides an overview of the new storage hardware and discusses evolution and background of the storage data path. Section 1.4 discusses the philosophy and design logics behind various data path features. Section 1.5 details limitations as we move toward SSDs and various proposed solutions. Section 1.6 presents our motivation regarding a novel design of a storage data path.

1.3 Legacy Storage Data Path Evolution

1.3.1 The layered structure of the legacy storage data path

The legacy storage data path, or storage stack, is composed of layers. Every request issued from applications traverses through the storage stack to access storage devices. The layered legacy Linux storage stack is shown in Figure 1.2.2. Similar functions are abstracted into multiple highly customizable layers.

Starting from the top, the application layer fetches data from and stores data through the underlying Virtual File System (VFS) layer through system calls. Applications reside in the user space, while the other layers are in the kernel space.

The VFS and file system layers are immediately below the application layer. The VFS consolidates common file-system functionalities into a single layer, provides a unified interface to applications, and leaves file-system-specific implementation details to the file-system layer. The files-system layer translates file-system-level requests into requests to the block I/O layer.

The block I/O layer merges, splits, reorders, and schedules the block requests to make use of the devices more efficiently. The requests are then submitted to the device-driver layer.

The device-driver layer, composed of vendor-specific device drivers, interfaces with physical devices and processes block requests. This layer implements all storage operations supported by the device.

1.3.2 Character of NAND flash storage device

NAND flash storage devices present in variable forms (e.g., SD card, 2.5" SSD, and M.2) and interfaces (e.g., SATA and NVMe). Their speed classifications and capacity vary by usage. All of

these devices store data the same way: by electronic charge. The voltage levels of electronic charge in the NAND cell can be used to represent bit(s) of data. The read and write operations to NAND flash are based on a unit of a "page," which is usually composed of 4Kbytes. The erase operation is based on a unit of a "block," which consists of 4-64 pages. The erase operation is slowest, but it is necessary before overwriting. Write and read are about 10x and 100x faster than erase respectively. A cell can only be recharged (overwritten) for limited times. Operations can be executed via multiple channels simultaneously.

1.3.3 Linux storage software stack components' adaptation to SSD and NVMe devices

Transitioning to flash-based media forces Linux storage software stack components to evolve. One example is the NVM-express (i.e. NVMe) support. The NVMe is a class of SSD storage device that has a much better performance than an ordinary SATA-interfaced SSD and can stress the legacy storage stack even more. Figure 1.3 shows a recent Linux storage stack, highlighting two new things: One is that the Intel SPDK [Intel 2016] provides NVMe support for a customized stack: the SPDK components include hardware drivers, storage protocols, etc. The other is the NVMe support at the block translation layer: the host software maps SCSI commands into NVMe commands.

1.4 Legacy Data Path Design Philosophy

The legacy data path is designed for traditional slow mechanical disks, with performance as the primary focus. Disk have improved over time, but they are very slow and lag far behind the performance evolution of other computing components (e.g., CPU and memory). Even though methods such as RAID can help, latency is still several micro seconds. In this case, the overhead of the storage stack is less relevant to the whole system: most of the time is consumed in

completing the disk's I/O operation (e.g., a small IO request may use 99% of the time waiting for the disk to complete writing) [Swanson and Caulfield 2013]. Therefore, for disk-based legacy storage hardware, the software overhead is negligible compared to the hardware overhead. The legacy storage stack also must address the limited amount of RAM and CPU cycles for certain environments (e.g., mobile devices), and consider minimizing its own resource consumption.

The layered structure is a good abstraction of legacy storage hardware functions and tasks. By extracting common functions and organizing them in layers, the storage stack can consolidate and save resources as much as possible, especially RAM consumption. Each layer can focus on its own data types and mechanisms to achieve information hiding, and it is easy to extend each layer by adding optional modules for additional functions to fit future emerging requirements. Because the software overhead is very limited compared to the disk overhead, the overhead introduced by communication through layers is acceptable.

However, as described in Sections 1.2 and 1.3, limitations in legacy storage stack design are becoming more and more severe. In the next section, the four main limitations are first discussed with relevant solution strategies, and various solutions are detailed thereafter. Some of them aim at one typical limitation, while others are mixed solutions that address more than one limitations.

1.5 Legacy Storage Stack Design Limitations and Solutions

1.5.1 The legacy storage stack layers are black boxes and introduce redundant functions across layers, which is difficult to optimize

As mentioned in Section 1.3.1, each layer has its own representation of data structure. Both logical and physical layers try to manage data layouts. For example, a database at the application level can represent data as a B-tree, and this B-tree is translated and represented as a skewed tree at the

file-system layer [Rodeh et al. 2013] and remapped again as a linked list at the flash translation layer. When one layer passes data to another layer, it needs to translate the original data representation into another form, leading to unnecessary overhead in terms of both data storage and processing time.

Also, the legacy layered structure of storage hides information across layers. Layers do not share functionalities, so essentially identical functions tend to be implemented at multiple layers. (e.g., for reliability, databases at the application level perform logging, the file system layer performs journaling, and the flash translation layer performs some forms of logging.) The isolation between layers also creates extra difficulty for optimization of these redundant functions.

A possible solution to the redundancy problem is to have tailored data paths for different applications, with multiple data paths specialized for different functions. For example, databases can have the option to bypass file system-journaling [Shen et al. 2014]. The challenge here is to maintain consistency across tailored data paths. Other examples include journaling of journals [Lu et al. 2016], TableFS [Ren et al. 2013] and Conquest [Wang et al. 2006] to demonstrate this idea.

1.5.2 The legacy storage stack has difficulty in coordinating and tracking across layers

Information hiding renders it easier for each layer to develop and extend its own functions, but also makes it difficult for each layer to make informed decisions. For example, device firmware may not be able to tell data blocks from metadata blocks or determine the file ownership of a block; similarly, if a block is in use, a file system may not know which device actually stores the data.

Lack of information sharing leads to lack of ability to track data across layers. For instance, it is hard to delete data securely: if a delete request is initialized from the application layer, the legacy storage data path does not know if there are hidden copies encoded in RAIDs, or where to delete data after wear-leveling mechanisms litter data on the SSD, or whether transformations of data can leave multiple versions of data (e.g., encrypted, and compressed versions).

One solution is to add customized interfaces to different layers along the data path to supplement this information-passing capability (data path augmentation). Since the solution involves structural retrofitting at multiple layers, its design and implementation are often complex. Willow extends the programmability of SSDs [Seshadri et al. 2014], and TrueErase incorporates the ability of cooperation between layers to achieve secure deletion [Diesburg et al. 2012]. These solutions often would also help mitigate limitations cited in Section 1.5.1 and/or 1.5.3.

1.5.3 The legacy storage stack is not designed for low-latency storage

While the default storage medium transitions from disk to flash, latency becomes less than 100 µs, compared to several milliseconds for disks, meaning that the relative portion of time consumed by the device I/O raises significantly. For a small IO request, only 30% of the time consists of waiting for flash storage to complete, as opposed to 99% for disks (Section 1.4) [Swanson and Caulfield 2013]. This means that the software overhead is much more important.

Continuing from section 1.2, the basic operations of flash-based media are reading, writing, and erasing, in contrast to those of disk-based media (just reading and writing). Flash-based media require erasing prior to writing to the corresponding zone. This complicates the read and write optimization strategies significantly. For flash-based media, the basic unit is a page for reading and writing operations but a multi-page block for the erasing operation. A typical block consists

of 32-256 pages. If an overwrite operation is issued to a previously written location (even one no longer in use), a write to a new location is performed. Otherwise, an erase operation must be performed to the whole block containing the previously written location before to writing the previously written location again. The writing of a single byte triggering writes to new locations, writes involved to migrate in-use pages from a to-be-erased block to an empty block, etc. are described as "write amplification." A wear-leveling mechanism is also needed to distribute writes to all writable locations as uniformly as possible, since each location can only be erased a limited number of times (e.g., 800).

The combination of these factors renders obsolete certain assumptions made by the legacy storage path. As the relative overhead of passing read/write requests through all layers becomes high, modifications become necessary to reduce the overhead of the legacy storage stack. For example, if the latency of a specific storage medium becomes comparable to RAM speed, the various caching-related mechanisms in the legacy storage stack actually become harmful. The typical solution is to tailor and bypass the data path or short-circuit part of the path. It often is discussed and resolved together with the limitations discussed in Sections 1.5.1 and/or 1.5.2. The Journaling Flash File System (JFFS) [Woodhouse 2001] is a solution example targeting this limitation.

1.5.4 The legacy storage stack has limited support for new storage models

The legacy storage stack focuses on retrieving and storing local data as fast as possible. However, modern applications have more diverse requirements and workloads for data access. For example, there are large-scale parallel file systems that are based on local storage systems and require huge overall bandwidth. Another example is that it is hard to cooperate with customized cache storage persistence mechanisms, as it is not easy to work with a distributed DRAM cache pool to support

a parallel file system. Though there were few such architectures in the old era, they are now commonplace. All of these issues are difficult for the current layered storage data path to accommodate.

Workloads representing new data models, such as key-value stores, are quickly replacing traditional files. Correspondingly, since the legacy storage data path is mostly dedicated to the POSIX API, it is difficult to extend the storage interface to non-filesystem APIs (e.g., key-value store APIs for non-relational DB access).

To solve this problem, we may (1) retrofit the layer structure, or (2) add a new layer or layers for new data types. A typical example is Cassandra [Lakshman and Malik 2010]. These solutions may also address the limitations in Section 1.5.2.

1.5.5 Existing approaches and their limitations

1.5.5.1 Journaling of journals

Journaling of journals [Shen et al. 2014] combines some aspects of application-level logging and file-system-level journaling to eliminate redundancy. For an ordinary sync of an application log, file systems with built-in journals have to introduce more write than necessary, which may impact performance. Combining journaling of log file data and metadata (which is considered journaling of journals) into a single sequential I/O operation saves the overhead of a separate write. Further optimization could be achieved by omitting synchronous writing to the main file-system structure, because it would be overwritten or deleted after subsequent transactions. Overall, consolidating application layer and file-system layer facility reduces functional redundancy to some extent.

1.5.5.2 TableFS

TableFS implements one data path for accessing small files/metadata and another for accessing large files [Ren et al. 2013]. The methods that effectively access small files and metadata are very different from those for large files. For large files, the goal is effective scaling for high-bandwidth data transfers, which is easy to achieve directly by the local file system through the legacy data path. But, for small files and metadata, a layer featuring key-value stores with large in-memory caches is helpful. Therefore, TableFS implements another data path specifically for small-file and metadata access.

We can see in Figure 1.4 that, after the FUSE module redirects application file-system system calls to TableFS, TableFS follows two different paths: The content of large files goes to the large-file store, which simply stores whole content under another name, and the content of small files and all file metadata is consolidated through LevelDB as a key-value store, and then forwarded to the local file system.

1.5.5.3 Conquest

Similar to TableFS, Conquest also divides data paths partly according to file size/metadata but also by attributes such as executables/shared libraries. It stores all small files, metadata, executables, and shared libraries in battery-backed RAM for persistence, while the disk only holds other large files. Because data types stored in persistent RAM are most frequently accessed, such an approach can achieve more of a performance boost than pure disk storage can.

1.5.5.4 Willow

Willow [Seshadri et al. 2014] aims at allowing developers at different storage layers to customize and install code pieces, called "SSD Apps," to extend SSD functionality and behavior. Being able to access data and functions across layers is essential for Willow to work. To implement this cross-layer coordination, Willow not only defines customized interfaces and provides software support but also integrates the necessary hardware to support the interfaces efficiently.

Figures 1.5 shows that in hardware, Willow uses storage processor units (SPUs) to replace CPUs in a general PCIe SSD. The SPU features four components: an SPU processor, local non-volatile memory, network interface, and programmable DMA controller. The network interface provides high-bandwidth access for the SPU interlink, a channel to local NVMe, and a connection to interfaces at other layers on the host. The programmable DMA controller supports high-bandwidth data processing for the RPC mechanism between interfaces across layers. The SPU runs SPU-OS to support different SSD Apps running in separated parallel contexts.

Based on the underlying system in Figure 1.5, the structure in Figure 1.6 shows how different interfaces and related components support Willow at different layers and call each other. The SSD App, which is on the device layer, communicates with the Willow driver and applications through the DirectIO mechanism to create a cross-layer context for customized code to run. This approach breaks the isolated layering structure for augmentation.

1.5.5.5 DAX

Direct Access X (DAX) is a mechanism in the Linux kernel for accessing high-speed non-volatile-memory-based storage hardware [Wilcox 2014]. DAX circumvents the traditional memory caching designed for high-latency storage. Since the speed of the non-volatile memory is very

close to RAM, this improvement can be meaningful. The DAX data path is demonstrated in the Figure 1.7.

By bypassing the caches on multiple levels and short-circuiting cache for direct media access, DAX can save both resources and persistence overheads. For instance, when the page caches of the VFS layer are eliminated, the pages can be mapped directly to a physical non-volatile memory region to save the overhead of serializing and deserializing the memory content.

1.5.5.6 JFFS

The JFFS [Woodhouse 2001] was originally developed to overcome problems in the Flash Translation Layer (FTL). The JFFS replaces the FTL and the standard log-based file system with a consolidated JFFS layer to not only avoid the problem involved with using the FTL but also eliminate the redundant journaling that occurred at both the file-system and FTL layers. By placing a log-based file system directly on flash chips, the JFFS no longer needs block translations provided by the FTL. The data path of the JFFS is shown in Figure 1.8.

1.5.5.7 Arrakis

Arrakis [SImon et al. 2014] is a system in which the kernel is removed from the data I/O path, routing the I/O requests to and from the application's address space directly. The applications rely on a user-level I/O stack library to achieve this. Figure 1.9 demonstrates that the kernel is no longer in the data path but only on the control plane. Libos, which represents a user-level I/O stack, enables each application to access a virtualized device instance (VNIC/VSIC) exclusively, without the concern of sharing/concurrency or security check. For network traffic, libos communicates through Virtual Network Interface Cards (VNIC) with the NIC; for data storage, libos talks with Virtual Storage Interface Cards (VSIC) to Virtual Storage Area (VSA), which is a logical storage

location with virtual offsets, and the VSA is converted into physical storage locations by storage controllers. The VNIC, VSIC, and VSA replace the kernel functions on the data plane.

1.5.5.8 KVFS/KVDB

The KVFS/KVDB system is designed to handle mixed file system and database workloads efficiently [Shetty etal. 2013]. The system incorporates KVFS and KVDB layers to handle key-value requests. Traditional applications can still be processed by the KVFS layer component via the file-system API. This design is shown in Figure 1.10. For a POSIX-compliant file-system call, the call request first goes through VFS to FUSE, and then it is passed to the KVFS layer to be translated into key-value operations for the KVDB layer. Databases and other applications can issue requests directly through key-value APIs through the Socket layer. Thus, the KVDB can efficiently process two different kinds of API requests/workloads simultaneously.

1.5.5.9 Cassandra

Cassandra is a distributed decentralized storage system for large-scale structured data [Lakshman and Malik 2010]. It was initially developed by Facebook to meet the requirement of very high throughput for writes and scaling capability because of the ever-increasing number of users. Cassandra can route a read or write request to any node to determine either the closest replica/first-responding replica for a read request or related replicas for a write request.

For local data persistence, a node has components to maintain in-memory data storage and a corresponding commit log above the local file-system layer. The new component layer supports a customized key-object API, which allows for storing and retrieving highly structured objects based on keys instead of the traditional file-system API. The layer still relies on local file systems for persistence and represents data as underlying files.

1.5.5.10 RamCloud

RamCloud is a class of storage aiming to provide scalable throughput and very low latency [Ousterhout et al. 2010]. RamCloud achieves this by putting all data in the DRAM of distributed servers. The magnetic disks are only used for backups and archives. Figure 1.11 presents the design of RamCloud server storage.

RamCloud treats objects as blobs of raw bytes, with aggregation and indexing for lookups. A single copy of each object is held in DRAM, while updates to the copy are logged to two or more other servers. The log entries are written to the disk asynchronously for persistence.

1.5.5.11 DevFS

DevFS [Kannan et al. 2018] moves the file system component into the storage device, so that applications can directly access a storage device without trapping into the OS for most of the operations while maintaining integrity, consistency, and security guarantees. Figure 1.12 demonstrates that DevFS, which resides on SSD, incorporates multiple functions that are original within the file-system kernel. DevFS the leverages the device-level power-loss-protection capacitors to eliminate redundant writes caused by logging and associated garbage collection mechanisms.

1.5.5.12 TrueErase

TureErase [Diesburg et al. 2012] is a framework designed to perform secure deletion for sensitive data. In this framework, the file-system layer can inform the device layer file ownership of a block, for the block to be securely deleted. The user can use attribute-setting tools to set a secure-deletion-related attribute for file and directory. The additional TAP module propagates secure-deletion attribute information to enhanced storage-management layer, and the management layer issues

storage-specific secure overwrite or deletion commands accordingly (Figure 1.13). This framework coordinates across layers by augmenting a data path to share secure-deletion information

1.5.5.13 Differential storage services

The differentiated storage services [Mesnier et al. 2011] coordinate the IO class information by expanding the block I/O data structure to propagate the information. To accomplish this, the OS, file system, storage system and application all need to be retrofitted to be priority and classification information aware and able to handle them respectively. Putting these changes together, users can exploit this feature as normal development with a few flags added. By only exposing information regarding the level of differentiation, the differential storage services manage to achieve a balance between cross-layer coordination and layered abstraction, without sharing so much information as to break the interoperability across layers.

1.5.5.14 Spiffy

Spiffy [Sun et al. 2016] allows file-system developers to annotate file-system-specific data structures and then compiles and generates a library so that application developers can write file-system-aware utilities using only generic library calls without knowing the specific file system's format. An application developer can use the Spiffy API to achieve implicit agreement with file-system developers on file-system specific logic. This is akin to two layers communicating and passing information through a third-party agent to make two otherwise isolated layers coordinate.

1.5.5.15 Strata

Strata [Kwon et al. 2017] leverages the strengths of NVM, SSD, and HDD and provides an integrated cross-layer design (NVM at the user level and SSD and HDD at the kernel level) to

achieve both high throughput and low latency. It implements a hierarchical structure to store different types of data into customized layers corresponding to zones in different storage media, processing the data conversion and migration between layers. It also separates specific operations from kernel functions into user level to improve efficiency.

1.5.5.16 PMFS

Persistent Memory File System (PMFS) [Dulloor et al. 2014] uses memory mapping and bypasses the block layer to achieve substantial performance gains. It is specialized for an environment with persistent memory storage media, implementing memory-like operations and totally eliminating the block layer. By avoiding data copy between DRAM and storage, and applying memory-style byte-based atomic instructions to implement storage operations, it can significantly improve performance compared to traditional storage operations.

1.5.5.17 FSP

File Systems as Processes (FSP) [Liu et al. 2019] moves the entire kernel-level storage stack into user space. As shown in Figure 1.14, the file-system process takes over the storage-related management task from the kernel. The application process sets up a private communication channel with the file-system process with help from the kernel, and the file-system process is in charge of interaction with the storage device to serve the application. Here, the user-space file-system process replaces the block-IO and device-driver layers. Combined with lightweight IPC, FSP can provide sub-microsecond latency while accessing NVM.

1.5.5.18 Aerie

Aerie [Volos et al. 2014] provides a framework to support custom storage abstraction (e.g., key-value store) and interfaces beyond conventional files; it is specialized for persistent storage-class

memory. For these super-speed storage media, delay caused by kernel interaction can be relatively significant or even critical to performance. Taking user-level libraries to manage this kind of device can break away from traditional layers, such as in-memory naming to reduce overhead. Aerie builds its own object stores on a customized naming scheme that represents new abstraction other than files.

1.5.5.19 Summary of above solutions and their limitations

Table 1.1 shows the solutions and the limitations they target. One approach to these limitations is to bypass the legacy storage data path by accessing the storage device directly (e.g., direct IOs, DAX [Wilcox 2014]). The downside to this is that application programmers may need to duplicate existing services in the legacy storage stack. Some solutions insert layers to separate the management of metadata and data (e.g., [Lu et al. 2016]) or deduce information across layers (e.g., [Sivathanu et al. 2003]), but they do not address the issues of redundant services and medium-specific mechanisms. Imperfectly deduced information may lead to optimizations based on conservative decisions [Arpaci-Dusseau et al. 2006]. To streamline storage requests and avoid redundant services, integrated design across multiple layers is possible (e.g., [Sun Microsystems 2004]). However, either these solutions are tailored for specific workloads [Shen et al. 2014] or the black-box treatment of layers remains and hinders information flow.

1.6 Summary

Although evolving, the legacy storage data path is still highly optimized for traditional magnetic disk and storage demands. The layered design ensures swift extensions of functionalities at all layers, while meeting the goals of performance and resource efficiency under the assumption of high-latency storage media.

Diverse storage demands and the introduction of low-latency storage devices put strain on the legacy storage data path design. The overhead of the storage data path itself can increase significantly, and layered design prevents communication and coordination across layers to meet diverse demands.

Various solutions have been proposed to resolve subsets of problems arising from the legacy storage data path design, but none have eliminated all the constraints. A unifying solution was yet to be found, and we were in search of a new storage data path design.

Table 1.1: Existing approaches for addressing storage data path limitations

Solution/ Targeted Limitation	Optimize for low-latency storage media	Support for new data model	Optimize through Redundant functions across the layers	Support for coordinating across layers
DAX	√		√	
JFFS	√		√	
KVFS/KVDB	√	√		√
Cassandra		√		
Ramcloud	√	√		√
Journals of journal			√	√
TableFS			√	
Conquest	√		√	
Willow		√		√
TrueErase	√			√
Arrakis	√		√	
DevFS	√		√	√
Differential Storage Services		√		√
Spiffy				√
Strata	√			√
PMFS	√			
FSP	√		√	
Aerie	√	√		

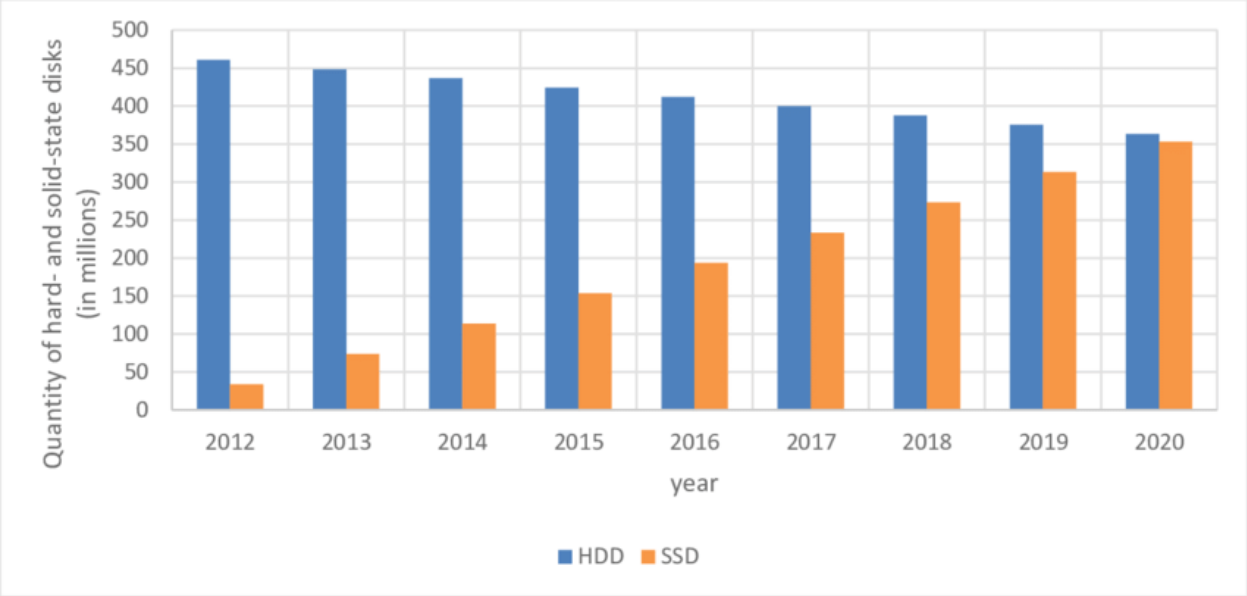


Figure 1.1: Shipments of HDDs and SSDs (in millions) worldwide in computers from 2012 to 2017 and projections for 2018-2020 [Statista, 2017]

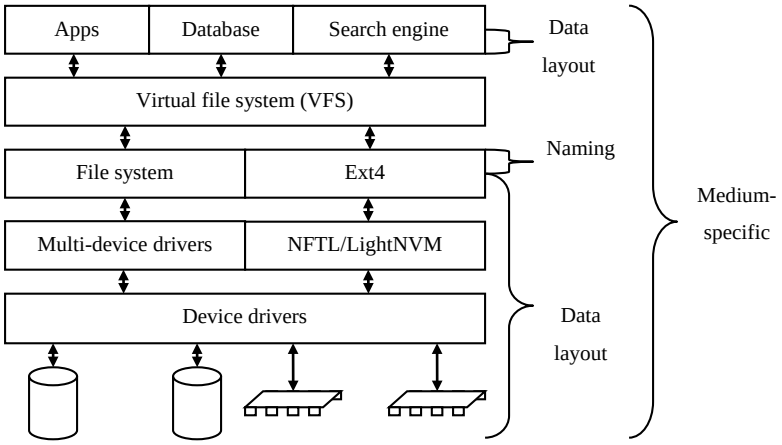


Figure 1.2: Legacy storage data path

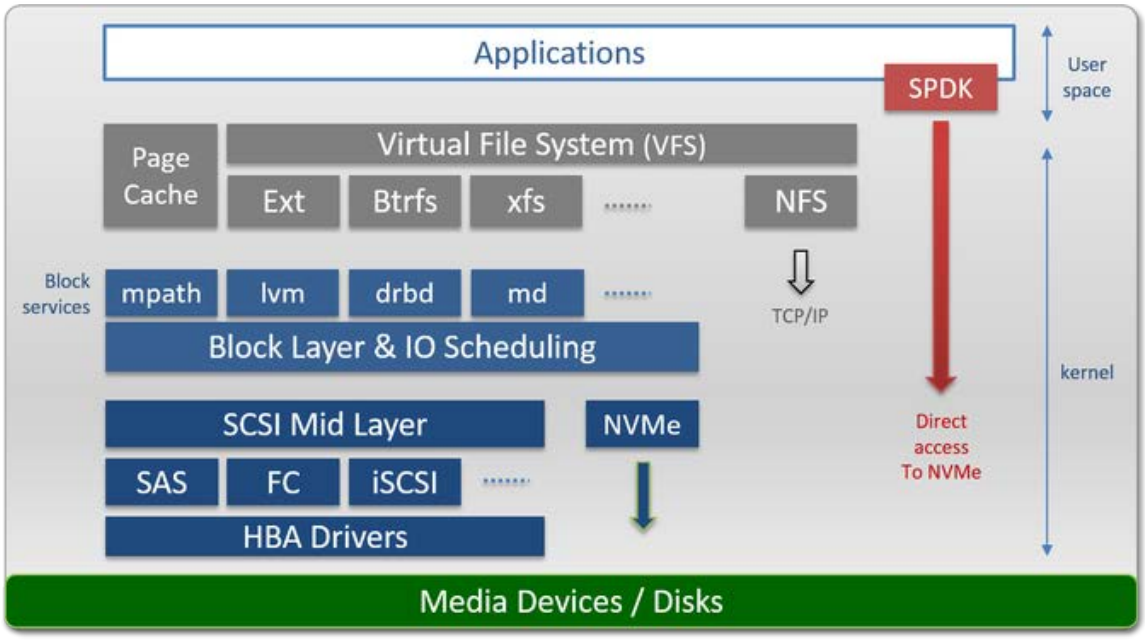


Figure 1.3: Recent Linux storage stack components

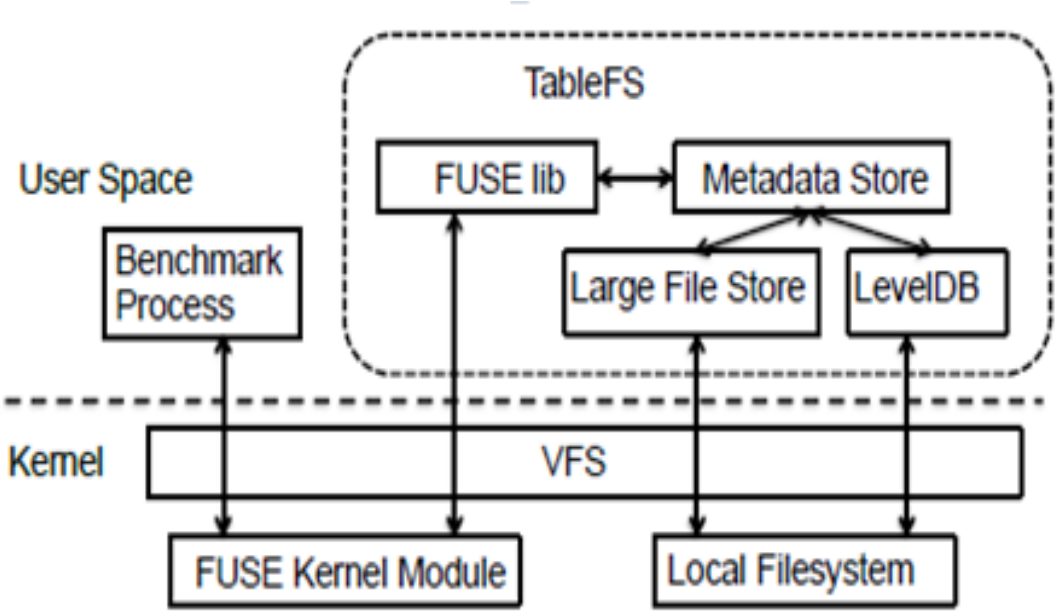


Figure 1.4: Dual data paths in the architecture of TableFS

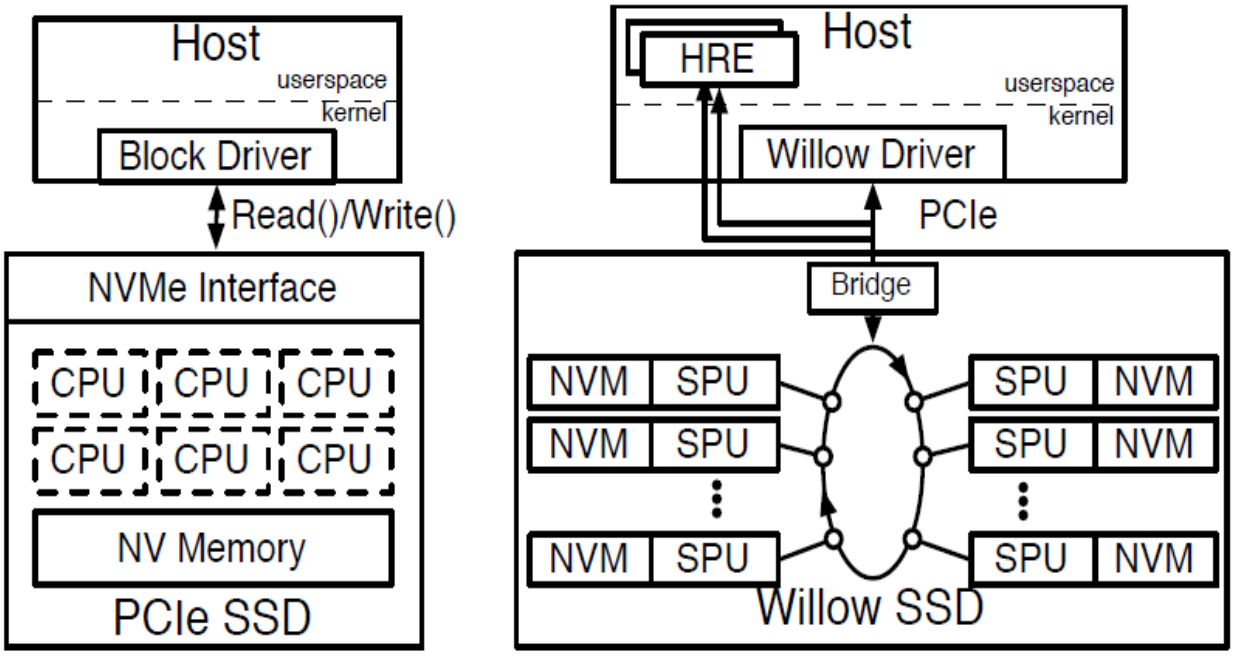
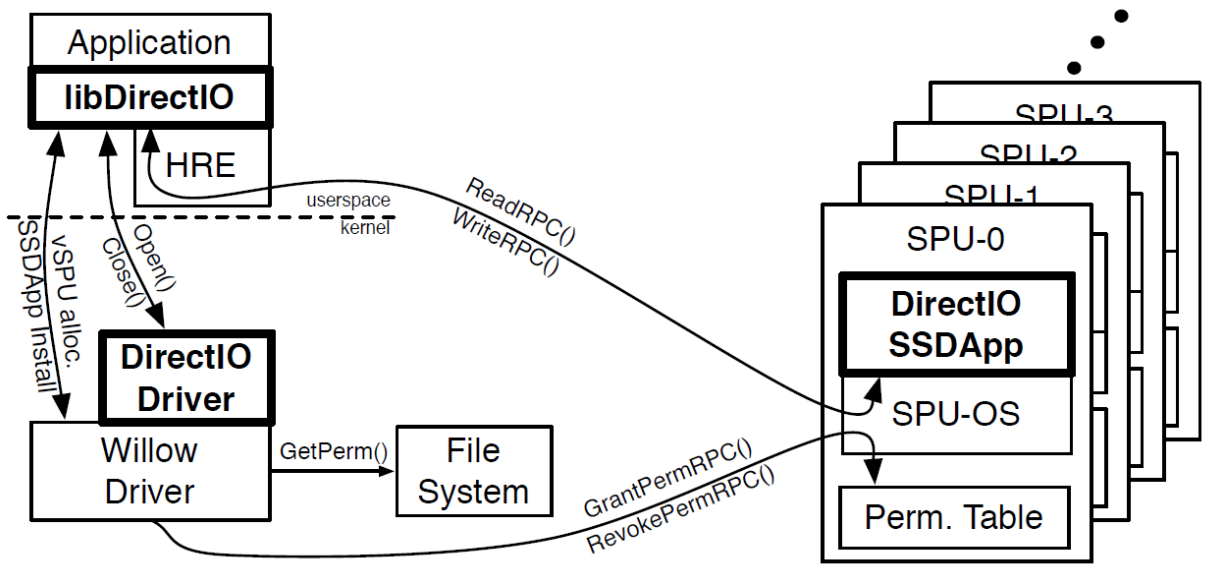


Figure 1.5: Hardware architecture comparison general PCIe SSD vs. Willow SSD



PCIe SSD vs. Willow SSD

Figure 1.6: The communication structure of the Willow SSD app

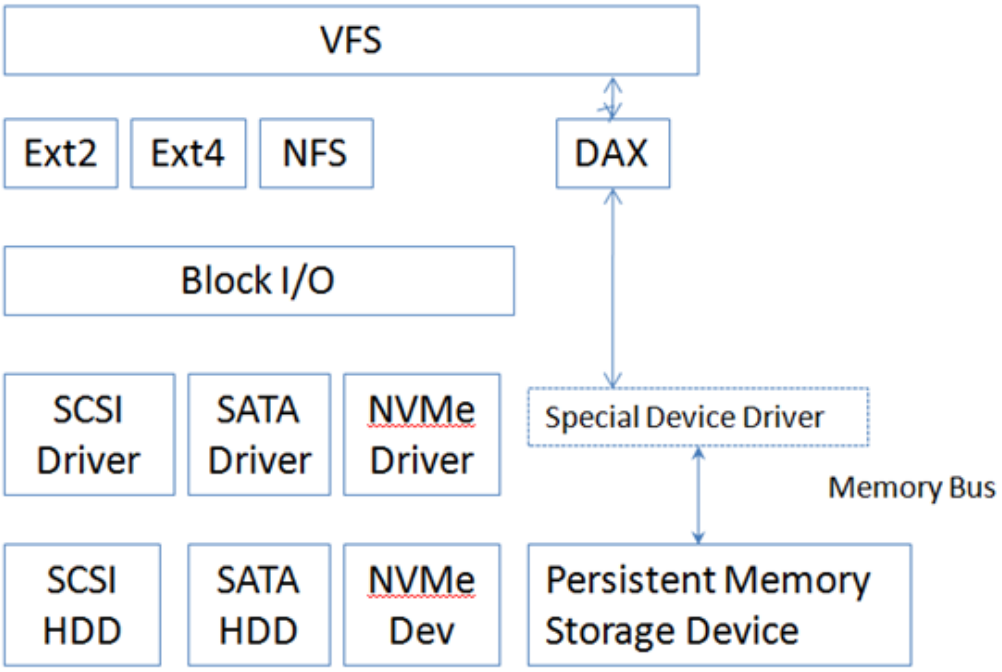


Figure 1.7: The DAX modified structure

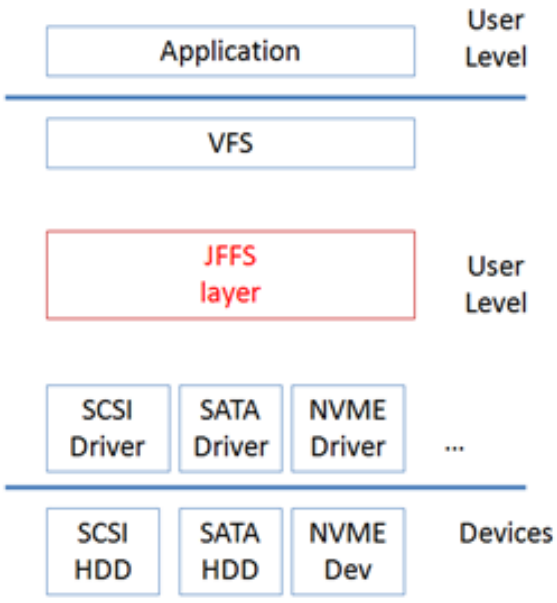


Figure 1.8: JFFS structure

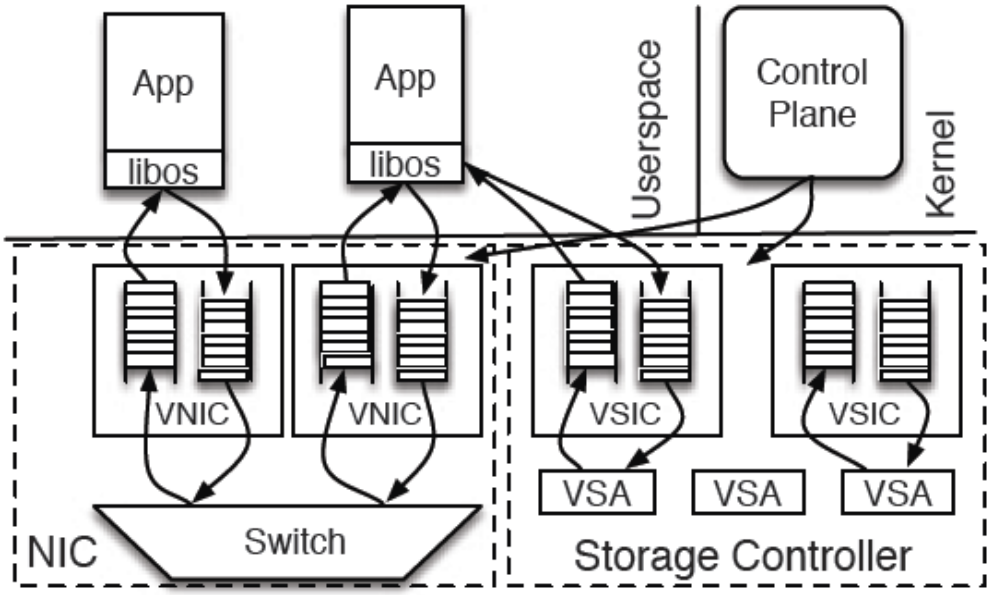


Figure 1.9: Arrakis architecture

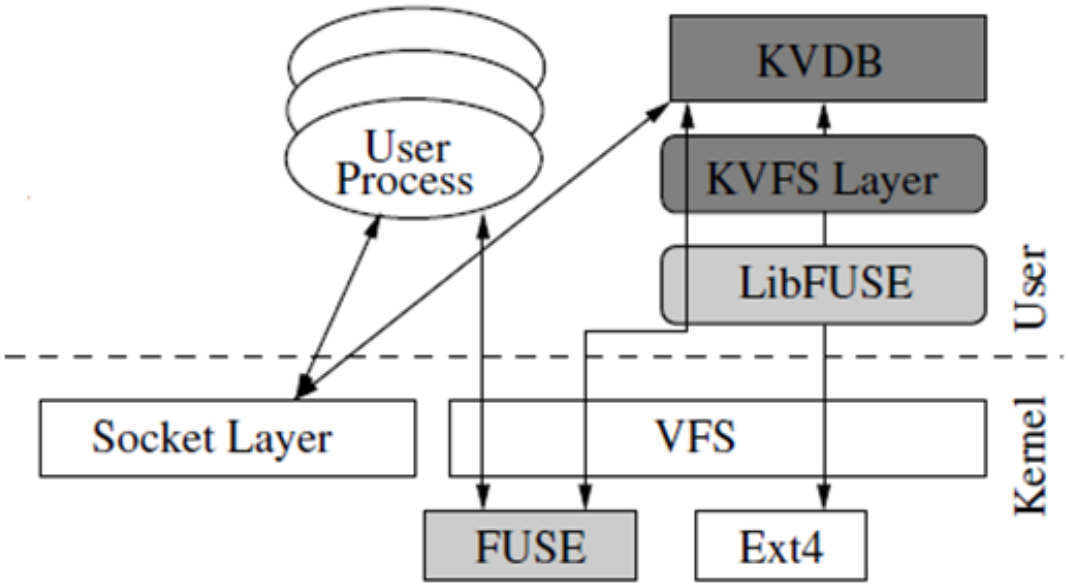


Figure 1.10: KVFS/KVDB structure

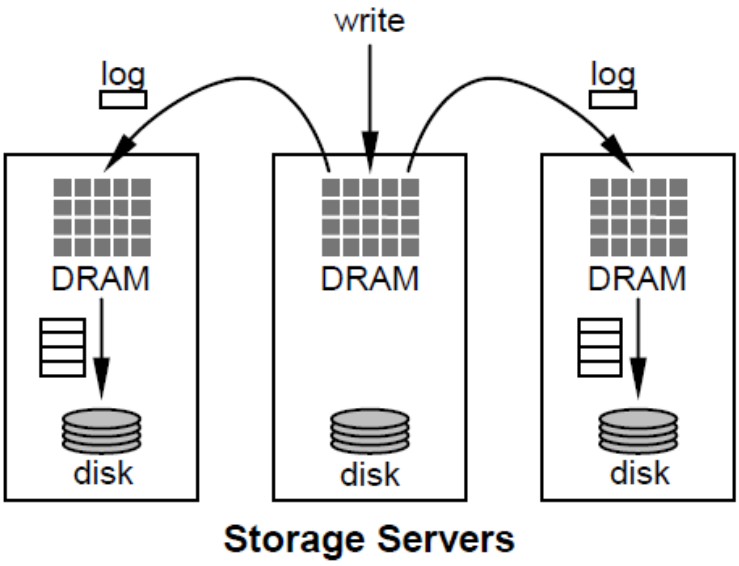


Figure 1.11: RamCloud server storage structure

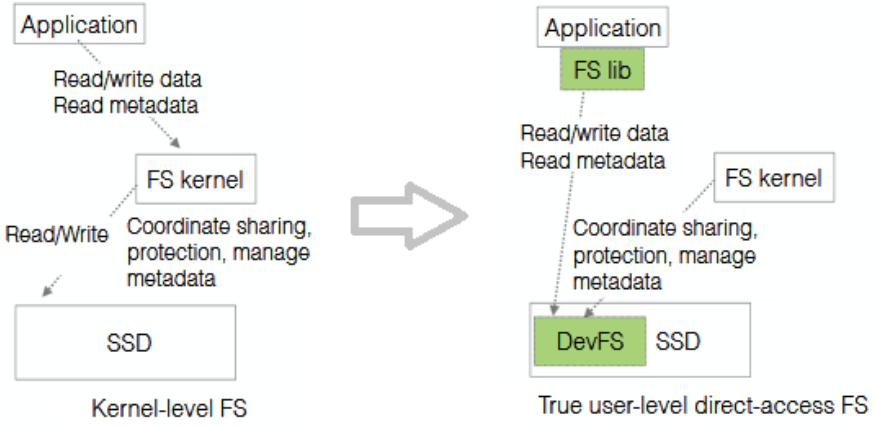


Figure 1.12: DevFS structure comparison with traditional FS

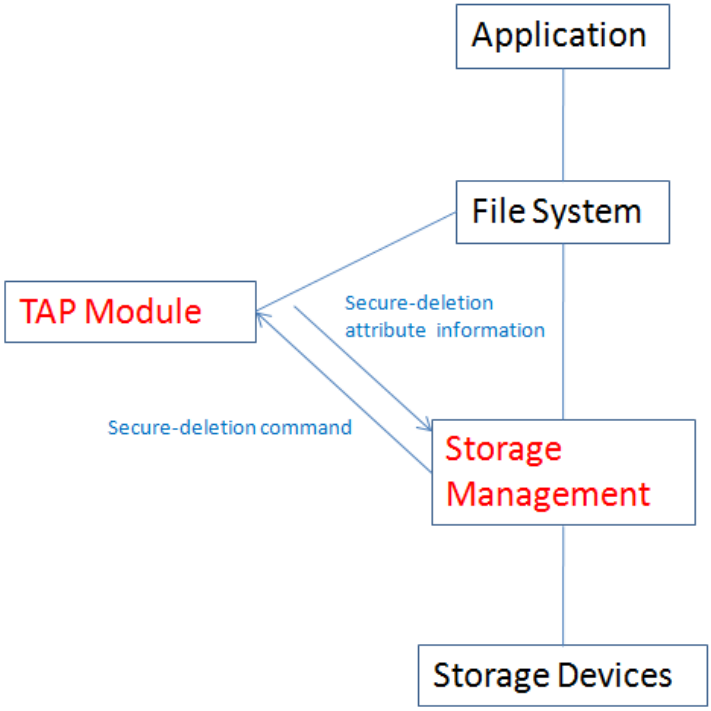


Figure 1.13: TrueErase framework in data path

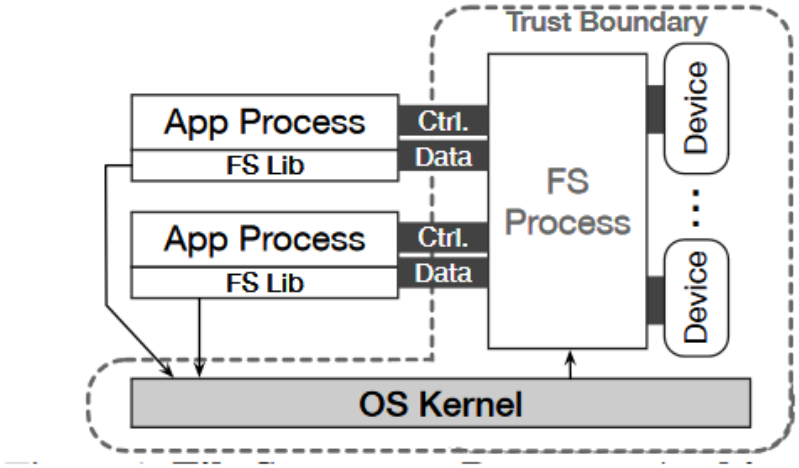


Figure 1.14: FSP system components diagram

CHAPTER 2

TAGS' SYSTEM DESIGN

2.1 Tags Concept

Legacy limitations prompt the question of how to design and build a new storage data path that is modular, and supports data-path-wide coordination, tracking, and emerging storage media. A more fundamental question is what makes a storage system a storage system? In essence, a storage system provides storage and retrieval of data. At a minimum, a storage data path needs the ability to store and retrieve data from a storage medium, and to tag data to provide persistence and control. From these basic requirements, we can rethink the concept and then design a unifying framework that addresses various limitations of the legacy data path.

We introduce the Tags framework, which uses tags, a unifying primitive (Figure 2.1), to construct shared data structures throughout the storage data path. Conceptually, each piece of data is associated with one or more tags, indicating how data pieces are related and should be handled within the data path. The collection of data pieces and tags forms a single-level data-tagging layer. To ease coordination, these tags provide global and logical communication throughout the data path. Tags also offer a common denominator for high-level storage layers and applications, allowing enough flexibility to accommodate the direct construction of name spaces by file systems and of indices by databases and to bypass redundant services (e.g., data structure remapping). Below the tagging layer, a consolidated layer containing the physical tags and data management makes informed decisions on how tags and data pieces are accessed and stored. As data traverse through the data path, they can be tracked using tagging.

Unlike in the traditional data path, Tags separates logical access from physical storage management, which enables medium-specific optimizations, easing the accommodation of emerging storage technologies.

2.2 Tags Design Space

Conceptually, each piece of data is associated with a globally unique ID (i.e., <data ID, “data”>). Each data ID can be associated with one or more types of globally registered and extensible tags, each in the form of <tag type ID, data ID>. Figure 2.2 is a Tags primitive example. It shows that the ID for “data” is 0. The access permission tag (with edge type ID = “E_PERMISSION) for “data” refers to the data ID of 1, which is “READ_ONLY”. The edge type ID "size" tag for “data” refers to the data ID of 2, which is “5”.

Although the data model is simple, storage modules can use tags as a common denominator when building data structures intended for cross-layer coordination and tracking. For example, through data ID indirections, we can build hierarchical graphs commonly used in file systems.

2.2.1 Graph-based API

Because a tag expresses the relationship between two pieces of data, we can logically transform Tags in terms of nodes and edges, with the nodes holding data and the tag types representing directional edges (Figure 2.3).

Figure 2.4 shows the core API for Tags. A node can be created to hold a dynamically allocated copy of data. A node can be destroyed given a node ID. An edge-type ID can be created with a given name. An example edge type is string, which can be used to specify a data node that contains, for example, a file name. To create or delete an edge, we must specify the IDs of the source,

destination nodes and edge type. Because dangling edges (those without end nodes) may lead to corrupted graphs, this API requires that the end nodes be created first, before the edge between the two. Before an edge can be deleted, the nodes must exist on both ends, and the user must delete the edge before deleting the end nodes. When an edge needs to point to NULL, an empty node can be used to assure that each edge is formed between two nodes. Certain edge types involve enumeration (e.g., block ID edge type); thus, when operating on edges, an additional optional info parameter is used to pass in the enumerated number.

A node can be accessed through its ID or the incoming edge of another node. To disambiguate, although a node can potentially be reached from different nodes through the same in-bound edge type, a node can be associated only with unique out-bound edge types.

Although it seems to be simple, applying tags along with data can build up complex graphs to represent arbitrary data structures. Providing the primitives to other layers can allow communication and coordination throughout the data path and also potentially replace the need to introduce extra data-scheme-related information in other layers. For example, a user can directly store and load B-trees built with Tags primitives without serialization and deserialization, avoiding the overhead of potentially remapping these B-trees in different layers.

In the physical management layer, data layout can be decoupled from the logical representation. For example, we can optimize across data-structure boundaries, so that frequently used data-structure fields can be grouped across data structures to exploit locality.

2.2.2 Group operations

One problem with using the graph-based API on fine-grained tags is achieving atomicity across many tag operations. Any failure along a sequence of graph operations would require lengthy cleanup code. When making multiple Tags calls, a caller needs to check the return value to each operation to ensure each node/edge operation is successful, and it invokes fallback logic if the return value indicates an error. During the fallback process, in the fallback handler, the node and edge structures created before the failure are removed. A typical fallback code structure is shown in Table 2.2.2.1. Because there could be multiple potential failure points, multiple goto statements for the corresponding failure handler can obscure the program flow and make the program logic difficult to follow and track.

To mitigate this issue, we added group operations (Figure 2.5). If an error occurs between the beginning and commit calls, the abort call automatically performs the graph cleanup and rollback to the graph states. The code in Figure 2.6 is then converted into the code in Figure 2.7, which has a single commit point and simple error return statements instead of various failure handler codes. The logic flow is much more streamlined compared to that in Figure 2.6. On implementation, when Tags APIs are called, they are first journaled in memory (put into a temporary memory buffer region to wait to be executed) until commit or abort is invoked. After committing, they are journaled in memory and executed in sequence, with mechanisms to ensure correctness (see Section 2.2.4). For an abort, the buffered region would be cleared and all pending operations cancelled. If Tags execution breaks or crashes, all uncommitted operations would lost.

2.2.3 Physical representation of tags components

Tags is a single-level store with operations revolving around nodes and edges. Nodes contain data, and there are directional edges between nodes to connect them. The physical representation needs to provide the foundation to support this logical design.

2.2.3.1 Nodes

Tags nodes are variable-sized, memory-mapped storage chunks governed by a memory allocator (e.g., slab [Bonwick 1994] and buddy allocators [Peterson and Norman 1997]). A node's memory address (offset by the starting memory-mapped address) is used as a unique ID for that node, freeing us from implementing node-allocation management.

2.2.3.2 Edges

Tags edges are implicitly stored in an extensible hash table [Fagin et al. 1979]. Basically, hash(source node ID, edge type ID, edge info) returns the destination node ID. The destination node can be tagged with a magic number to perform a dynamic type check prior to accessing the node's content. For example, the magic number can confirm whether the node content "123" will be accessed as a string type or an integer type.

The key of the hash table is generated from the originate node ID and the edge type ID:

key = hash(from_node_ID, edge_type_ID)

The value in the entry of hash table is the destination node ID, or a pointer to the destination node:

table[key] = to_node_ID

When an edge is created, a new entry is inserted into the hash table:

table[created_key] = to_node_ID1

When an edge is updated, just the entry value is switched:

table[created_key] = to_node_ID2

When an edge is deleted, the entry value in the hash table is deleted.

The from_node_ID and edge_type_ID needs to be created/deleted beforehand/afterward to maintain valid node edge graph. Otherwise there would be orphan nodes or dangling edges.

2.2.3.3 Persistence

To survive reboots, the memory allocator's states must be persistent. The governed memory is divided into regions for metadata, data, ephemeral states (to optimize the Tags internal data structures), and storing critical start-up information (e.g., offset of active persistent state storage). The metadata region includes the states of the memory allocator and edge table. Metadata and data regions (except for the ephemeral states) are flushed from memory and disk according to the snapshot protocols in Section 2.2.4. Regions, which are identified by memory address ranges, are memory mapped to memory pages (non-persistent information and data) or persistent storage (when active, these would also have memory pages assigned to them). The memory locking control can be applied to these pages to support concurrency. The critical information that needs to be loaded at system start-up time is stored at a fixed offset, so that the system can address them directly. This information includes address ranges of other regions, and these regions can only be loaded afterward.

2.2.3.4 Data layout

With the storage organization for Tags, data layouts are largely governed by the memory allocator for nodes and the representation of the hash table for edges. We first allocate a memory pool, then perform customized allocation within it. For memory allocation requests greater than a page, we

use a buddy allocator. It uses and manages a contiguous memory region by dividing it into memory with power-of-two sizes. The blocks of same block size can be merged or split to make space available for requests of different sizes. To encourage locality and reduce fragmentation, we used a customized slab allocator for sizes below one page, which saves memory space consumption by small Tags objects and also encourages objects of similar sizes to be collocated within a memory page.

Since pure hashing has poor locality, we modified hashing to encourage the destination node IDs (pointers to destination nodes) from the same source node ID to be collocated. As a simplified 32-bit example, suppose the from_node_ID has a hash key of 0x0011011. We use the upper 20 bits of from nodes to determine to store the key in a 4KB memory bucket 0x00110. Suppose the to_node_ID has a hash key of 0x00001100, and it will use the upper bits of its source node 0x00110 to determine in which memory bucket to store the to_node_ID, and the lower 12-bits to locate the to_node_ID within the bucket. As the hash table increases in size, fewer upper bits will be used to locate bigger memory buckets.

Note that it is possible to reach the same to_node_ID from different from_node_IDs, and each from_node_ID can have its own cluster of to_node_ID (pointers to destination node).

2.2.4 Snapshotting

The Tags system holds its data and metadata including edge and memory allocator internal data, in persistent storage. If a power outage or failure occurs, we need to ensure that the metadata itself and metadata-data mapping do not become inconsistent and result in total failure and loss of data.

To resolve this problem, Tags' failure-recovery mechanism is based on the state-based snapshotting technique. Tags maintains three copies of metadata, one in memory for metadata updates (MM), one on storage for rolling back (previously checkpointed) metadata (MS), and one on storage for committing in-transit updates (MS'). Tags maintains two copies of data, one in memory (DM) and one on storage (DS). A group of operations that has been committed are first journaled in memory. To process transactions of updating nodes and edges, Tags performs the various steps in three phases:

- ***Phase 1 (Commit data)*:**

Update data items from DM to DS. Deleted data items are marked as to be deleted in DM until the metadata commit is complete. If a failure occurs before Phase 1 is committed, no updated metadata will be written. In the case of a crash, it may be possible for old metadata to point to newly updated data items; however, the consistency semantics is akin to that of the ordered mode for ext3. Data nodes without edges can be detected and deleted via delayed garbage collection.

- ***Phase 2: (Commit metadata):***

Commit metadata creations, updates, and deletions from MM to MS'. If a failure occurs before Phase 2 is completed, MM and MS' will be restored to the previously checkpointed metadata (MS).

- ***Phase 3: (Deletion and metadata checkpoint):***

Delete data items that are marked as to be deleted. Propagate metadata creations, updates, and deletions from MM to MS. If a failure occurs before this phase is complete, the transaction can be replayed (roll forward) from the commit.

To reason and verify the correctness of group operations, we applied the file system consistency properties defined [Sivathanu et al. 2005] for analysis:

The reuse-ordering property ensures that once a file's data block (a Tags node) is freed, the block will not be reused by another file before its free status becomes persistent. Otherwise, a crash may lead to a file's metadata (a Tags edge) pointing to the wrong file's content. In Tags, all edges are deleted and committed prior to deleting the connected node, assuring that the node is no longer reachable from the remaining graph before the node is deleted and reused.

The pointer-ordering property ensures that a reference data block (a Tags node) in memory will become persistent before the metadata (a Tags edge) in memory that references the data block. With this ordering reversed, a system crash could cause the persistent metadata to point to a persistent data block location not yet written. In Tags, all nodes are flushed prior to the edges.

The non-rollback property ensures that older data or metadata versions will not overwrite newer versions persistently. In our case, since journaled entries are ordered and applied to the global snapshot chronologically, Tags has fulfilled this property.

2.2.5 Access control

Since Tags aims to create primitives smaller than the granularity of common data structures, we anticipate many small tags, rendering high overhead for per-node permission checks. Allowing edges to be created between any two nodes is also an unwieldy way to enforce the permission to access restricted nodes. However, since many tags share the same permission, it is logical to check and enforce permissions at fewer locations (see super nodes below). In addition, a certain

degree of restrictions on how edges can be formed can manage the access control properties of the resulting graph topology.

2.2.5.1 Super nodes

The idea of super nodes (s-nodes) is to reduce the number of places where permissions are set and checked: only s-nodes have edges to permission nodes (e.g., ownership, group, and permissions). All nodes belonging to the same s-node implicitly share the same permissions. In terms of restrictions, edges can be created from an s-node to its nodes (the rule is shown in Table 2.1). Edges can also be created from any node to an s-node, since that destination s-node can enforce the access permissions. Section 3.3 demonstrates the use of s-nodes through a file system example. An edge also can be formed from an s-node to itself (e.g., the support of "." for directories). However, forming edges between nodes that are under different s-nodes is prohibited, and a source s-node cannot create an out-bound edge to a node under another s-node.

One challenge to realizing s-nodes is finding a node's s-node without additional edges or lookup tables. Since our unique node IDs are based on 64-bit memory-mapped addresses, we borrowed unused *S* high-order bits. An s-node ID is a unique *S*-bit number, zero-extended to form a 64-bit ID. To access its nodes, the s-node must be connected to at least one of its nodes. To locate the permission from a node under an s-node, we use hash (zero-extended upper S bit of the node ID, permission edge-type ID).

In terms of the API, a programmer can use a special call to produce s-node IDs and use them to create node IDs (Figure 2.8). The s-node tracks the number of nodes created beneath it. To delete an s-node, all its nodes must first be deleted. Otherwise, the permission of the undeleted node will be either undefined, or defined by a newly allocated s-node with a reused s-node ID.

In this model, the s-nodes can form loops themselves, without a conflict access rights problem. Developers and users can determine whether to use this feature; if so, they need to consider the data model to divide normal nodes into different access right groups. An application that does not use it can have very little performance overhead.

Suppose a user decides to change the design of the graph and wants node N1 under s-node S1 to have a different permission. This change will involve creating a new s-node S2, so that the prefix of the s-node ID can be used to create a new node N2, which is a copy of N1. All existing relationships between N and nodes under S1 need to be created or reestablished through S2 as an intermediary node for permission checks, and N1 can be removed from the graph. Given the complexity and potential associated overhead of this operation, a user should not change the access control points lightly.

2.2.5.2 Sessions

Since node IDs are capabilities, we need the ability to revoke privileges. Thus, other than for the root node ID, the user should interact with Tags through translated IDs. This mechanism is enabled using session open and close calls and a primitive translation table (Figure 2.9). An open session call is needed before accessing the translated root node of a Tags graph. All subsequent node IDs obtained from the root node's edges are translated via a translation table. At the end of a program, a close session call is needed to delete the translation table. Optionally, a timeout can bespecified to close a session when the system registers no activity within a timeout period.

2.3 Summary

This chapter discussed the high-level logic concept and design logics of the Tags system. The next chapter focuses on detailed implementation of these design logics and case studies of user applications built upon Tags.

Table 2.1: Rules for creating edges between nodes/s-nodes

from \ to	s-node A	s-node B	s-node A's nodes	s-node B's nodes
s-node A	**Yes**	**Yes**	**Yes**	**No**
s-node B	**Yes**	**Yes**	**No**	**Yes**
s-node A's nodes	**Yes**	**Yes**	**Yes**	**No**
s-node B's nodes	**Yes**	**Yes**	**No**	**Yes**

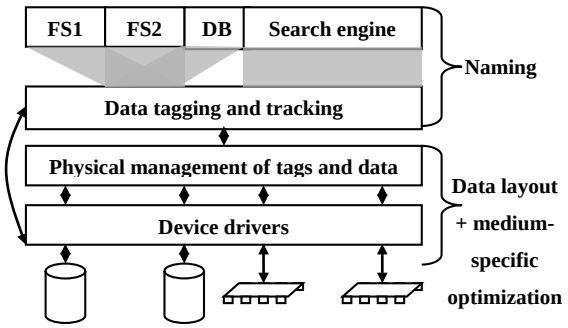


Figure 2.1: Tagging-based storage data path

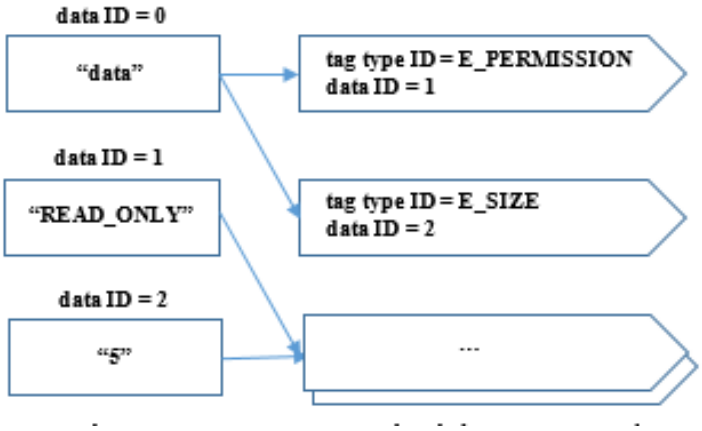


Figure 2.2: Tags primitive example

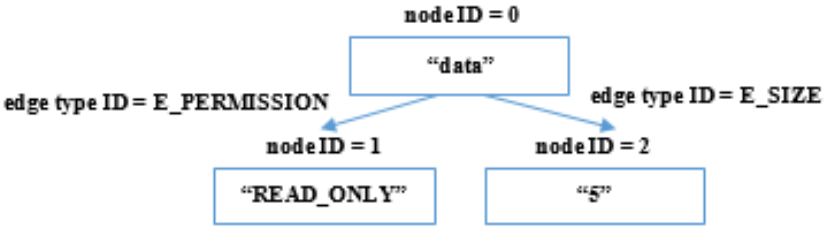


Figure 2.3: Graph-based representation of Tags

```
node_ID = tags_create_node(data, len, …);
tags_delete_node(node_ID);

edge_type_ID = tags_create_edge_type(name);
tags_delete_edge_type(edge_type_ID);

tags_create_edge(src_node_ID, dest_node_ID,
edge_type_ID, <edge_info>);
tags_delete_edge(src_node_ID, dest_node_ID,
edge_type_ID, <edge_info>);

tags_get_dest_node(src_node_ID, edge_type_ID,
<edge_info>);
tags_ID_to_node(node_ID);
```

Figure 2.4: Core API for Tags

```
group_op_ID = tags_group_begin_ops();
tags_group_abort_ops(group_op_ID);
tags_group_commit_ops(group_op_ID);
```

Figure 2.5: Group operations for Tags

```
inttags_add_edge(…) {
…
err = create_edge(edge1);
if (err)
goto failed_1;
err = create_edge(edge2);
if (err)
goto failed_2;
…
failed_2:
delete_edge(edge1);
  failed_1:
    return;
…
}
```

Figure 2.6: The error handling of Tags without group operations

```
inttags_add_edge (…) {
  …
group_op_id = tags_group_begin_ops()
err = create_edge(edge1);
if (err) {
tags_group_abort_ops(group_op_ID);
    return err_no_1;
  }
  err = create_edge(edge2);
  if (err) {
tags_group_abort_ops(group_op_ID);
    return err_no_2;
  }
  …
tags_group_commit_ops(group_op_ID);
…
return success;
}
```

Figure 2.7: The error handling of Tags with the use of the group operations

```
s_node_ID = tags_create_s_node(mode);
node_ID = tags_create_node(data, len, s_node_ID);
```

Figure 2.8: Super node operations

```
translated_root_node_ID
  = tags_open_session(root_node_ID, <time_out_minutes>);

tags_close_session(translated_root_node);
```

Figure 2.9: Super node operations for sessions

CHAPTER 3

TAGS IMPLEMENTATION, EVALUATION, AND CASE STUDIES

3.1 Tags System Implementation

Tags is prototyped in C as a user-level library. Tags applications link and load the library to use the API to perform storage tasks. Figure 3.2 shows how a Tags-based key-value store (Section 3.3.2) uses the Tags library to interact with the kernel and communicates with the kernel via memory mapping and shared memory.

Figure 2.1 shows the two major components of Tags. The data-tagging and data-tracking component implements the graph API, nodes and edges, group operations, and access control. The physical management component implements the persistent memory allocator, which also controls the physical data layout. Currently, the Tags library does not support sessions or multi-threaded and nested group operations.

While Tags system initializes, it first constructs its layout and initializes the memory allocator according to loaded critical start-up information. Table 3.1 shows part of the variables being loaded, and then metadata items (including memory allocator information and hash table) are mapped, loaded, or created. If the system is dirty (there was a failure or crash), the recovery steps mentioned in Section 2.2.4 would be executed according the dirty value indicating in which phase the failure or crash occurred.

The memory allocator uses a combination of buddy allocator and slab allocator (see Section 2.2.3). The minimum page size of buddy allocator is customizable, currently defined as 4KB. Below the page size, a slab allocator is used. For in-memory metadata and data, the corresponding memory

allocator is constructed on the malloced memory pool. For on-storage persistence, metadata and data are memory-mapped.

While Tags is running, it controls flushing of metadata and data in the form of memory allocator pages conforming to the procedures and requirements defined in Section 2.2.4. Figure 3.1 shows the main components of Tags.

3.2 Tags Evaluation via Case Studies

While evaluating Tags, we wanted to show its ability to:

1. Avoid redundant layered features,
2. Achieve usability and robustness when building complex software,
3. Extend and support new data models and features beyond the ones provided by the legacy data path,
4. Coordinate and track data across layers, and
5. Perform well with both HDD and SSD storage media.

To show that Tags can perform well with HDDs and SSDs, in each experimental setting, we conducted benchmarks on both media. Each experiment was repeated five times and is presented at the 95% confidence interval. Table 3.2 shows the system configuration.

3.2.1 Tags-based key-value store

To show the benefit of the direct support for new data models, we prototyped a key-value store using the Tags library. The Tags data path had no file system and associated redundant efforts to manage data layout (Figure 3.2).

Given that Tags is built on a hash table that stores edges to nodes, its operations can be directly mapped to support key-value store operations. We began by creating a root node. For the key-value `Put(key, data)` operation, we created a node to store the data and used the key as an edge type ID. For `Get(key)`, we called `tags_get_dest_node(root node ID, key)` to retrieve the data node. For `Delete(key)`, we called `tags_delete_edge(root node ID, node ID)`, followed by `tags_delete_node(node ID)`.

We compared the Tags-based key-value store with LevelDB 1.9.0 [Ghemawat and Dean 2019]. Figure 3.2 shows the differences between the two data paths. For the workload, we inserted 10 million 100-byte key-value pairs, each with 16-byte keys. Figures 3.3 and 3.4 show the results.

For both storage media, Tags and LevelDB have similar read performance, since both systems use memory-mapped IOs to avoid copying. Both systems also use bulk updates (group operation for Tags) to speed up small updates. For HDDs, Tags can outperform LevelDB by a factor of 1.6 for inserts and 2.2 for deletes. For SSDs, Tags can outperform LevelDB by a factor of 1.1 for inserts and 1.3 for deletes. We also ran LevelDB with the raw mode enabled to confirm that Tags' insertion and deletion performance improvements are attributed to eliminating the file-system layer.

3.2.2 Tags-based file system

To demonstrate usability, we prototyped TagFS to show that the interface and primitives provided by Tags are expressive enough to build meaningful and complex applications. While users of Tags need to learn a new interface, Tags saves them from writing serialization and deserialization code. TagFS was implemented at the user space via the FUSE framework [Szeredi 2005]. Figure 3.5 illustrates the flow of data requests.

TagFS implements several FUSE interfaces, including read, write, open, unlink etc. By working with Tags primitives inside these interfaces, TagFS was able to provide functional file system services. More precisely, TagFS translates POSIX file system calls into Tags-based nodes and edges; this task involves many node and edge operations, simplified by group operations. Basically, all i-nodes (permission holding nodes) are replaced with s-nodes, and all attributes are accessed through edges (Figure 3.6). Directory entries can be accessed via ID hashes. For traversals, a directory entry can locate the next and previous entries through `hash(current ID, next edge type)` or `hash(current ID, previous edge type)`. Data blocks are accessed through enumerated edges to support indexing on top of the hashing data structure.

Although we could use a single node to contain all attributes of an i-node, we explored this scenario to show that even if tags are naively applied, we can still configure the system to achieve reasonable performance. We compared our TagFS with ext4 stacked on FUSE. The elapsed times for TagFS and ext4 + FUSE to compile the OpenSSL (v1.1.0f) [2019] were statistically the same (87 ± 0.01 seconds).

We also prepared a directory containing Linux 4.1 (after a complete build) and examined the elapsed time for recursively listing through the directory (Figure 3.7). The hash-based doubly linked list of directory entries in Tags is slightly slower than ext4 + FUSE for both HDD and SSD.

For running LFS large- and small-file benchmarks [Rosenblum and Ousterhout 1992], TagFS performed reasonably well when its block size was configured to 32KB, to amortize the cost of fine-grained access to attribute nodes and dynamic type checks (Figures 3.8-3.11). We admit that using a larger data block is not a fair comparison (4KB for ext4 + FUSE); however, our objective is to show that Tags' API is sufficiently rich and its implementation robust enough to build

software with comparable complexity to a file system with reasonable performance. As a side effect of the large block size of TagFS, it significantly increases the performance of random reads (due to unintended cache warmup per random read access). The random write bandwidth numbers for both ext4 + FUSE and Tags are higher than expected due to buffering. On the other hand, the overhead of fine-grained per-file-attribute Tags does show up on the creation and deletion benchmarks, where each operation involves creating and deleting many tags.

3.2.3 Tags-based B+ tree implementation

Another example to show Tags' capability to support new data model is to implement B+tree. Similar to the key-value store, it is built on the Tags library and has no file system involved.

The nodes in B+ tree are represented by Tags nodes, while the connections between nodes are represented by Tags edges.

We compare the Tags-based B+ tree implementation with normal B tree implementation, which utilizes the serialization/deserialization method to load from/save to files on storage media. The serialization/deserialization method takes a breadth-first way to process the nodes in a B+ tree into data in a file.

The dataset for test contains five B+ trees; each has a depth of three and fan-out factor of 342. The key size is 4 bytes, and pointer size is 8 bytes; therefore, each node occupies about 4KB. The whole size of each tree is about 480MB, and the total size of all trees is about 2.4GB.

We assessed the performance regarding query, deletion, and insertion to the Tags- and file-based implementation. For query, 500K queries for random-key-mapped pointers to 80% full B+tree were performed. For insertion, 500K insertion operations to 40% full B+tree with random key-

pointer pairs were performed. For deletion, 500K deletion operations to 80% full B+tree were performed. For file-based B+ tree, serialization/deserialization was included. Figure 3.12 and 3.13 show the test results for operation time consumed (a shorter time is better).

3.2.4 Differentiated storage services

To demonstrate how Tags can be extended to support features beyond the legacy data path, we implemented a feature similar to differentiated storage services [Mesnier et al. 2011], which allow different classes of data to be treated differently. In our example, we defined large files, small files, and metadata as different classes, and we controlled the cache policy at page cache level by modifying the LRU cache to give preference to caching small files and metadata to achieve performance gains.

In terms of Tags, we leverage the per-file permission lookup mechanism to directly access a file's s-node, and reach its subsidiary node tagged with the data class. The caching mechanism then can prioritize caching for blocks tagged as small files and metadata. We modified the mmap call mechanism and page cache structure to pass down and store the block classification to be used as criteria for page scheduling.

We compared the performance of TagFS with differentiated storage services enabled and disabled. The workload involves a zero-think-time replay of a departmental web server trace, which contains 8.6M file references to 1.04TB of data, among which 1.53M files are unique with 121GB of unique data. We classified files under 18KB (75% of files) as small.

Figure 3.14 shows that by the end of the trace replay, up to 95% of the cache is populated with small files and metadata. Figure 3.15 shows that the overall elapsed time for the trace replay is

improved by 40% for both HDD and SSD. Figures 3.16 and 3.17 show that the request completion times shift lower for both HDD and SSD.

3.2.5 Per-file secure deletion

To demonstrate cross-layer coordination and tracking, we augmented TagFS with a per-file secure-deletion feature akin to that of TrueErase [Diesburg et al. 2012]. First, a user can use chattr +s to set the secure-deletion bit of a file at the file-system layer. However, by the time a storage request arrives at the device-driver layer, the layer can no longer identify the file membership of a block.

In TagFS, since each group of nodes is governed by an s-node to manage the permission, any node (e.g., a data block node) under an s-node can reach the s-node (see Section 3.4). Consequently, TagFS can access the permission. The secure-deletion bit indicates that the corresponding overwrite or truncate/delete should be handled securely.

We handled the disk case by zeroing out data blocks that needed to be securely overwritten and truncated at the block layer. We borrowed the `ioctl` call mechanism from FUSE to allow the block layer to issue calls from the kernel space to Tags to query if a block belongs to a file that has the secure-deletion bit set. If so, the block layer would perform triple writes of random bits to ensure secure deletion. If the secure-deletion configuration is disabled (through the Tags configuration file or an instruction), for such `ioctl` calls, TagFS would return a special value to the block layer to indicate this. Updates to blocks belonging to a file marked for secure deletion are prepended with triple writes of random bits to securely delete the previous content. We also modified the truncate mechanism, so that it will issue calls to erase data blocks. Without the open FTL and raw flash setup, we did not implement this feature for SSD. Note that the TRIM command

is insufficient; it only specifies which pages are obsolete to prevent migrating these as live pages during the garbage collection process [Shu and Obr 2007].

3.3 Summary

Based on the design mentioned in Chapter 2, we implemented libraries to be used to construct various applications on top of Tags to demonstrate its usage and advantage. The applications include new data workload types such as key-value stores, direct data-structure representation such as B+trees, differentiated storage services, and traditional file access, including extended secure-deletion function support. The evaluation results validate the goals we set in Section 3.2, and the benchmark numbers show comparable or even superior performance.

Table 3.1: Tags initialization variables

Variable Name	Variable Type	Meaning
IMAGE_MAP_START_ADDR_MEM	64 bit integer	Start Address of the Tags memory region in image
STATIC_VAR_HT_POINTER_MEM	64 bit integer	Start address of critical start up variables
IMAGE_DIRTY_SHUTDOWN_PHASE	8 bit char	If previous shutdown of the Tags system is an unexpected failure/crash, indicates different phase between shutdown
IMAGE_IDENTITY_VER_MEM	64 bit integer	Version number information of Tags
…….		

Table 3.2: System configuration

CPU	2.2Ghz Intel® Xeon® E5-2430, 15MB cache
Memory	32 GB RDIMM 1333 MT/s
HDD	Seagate® SAS 146GB 15K RPM
SSD	Intel® S3500 200 GB SATA Value MLC
Operating system	Linux Mint 3.19

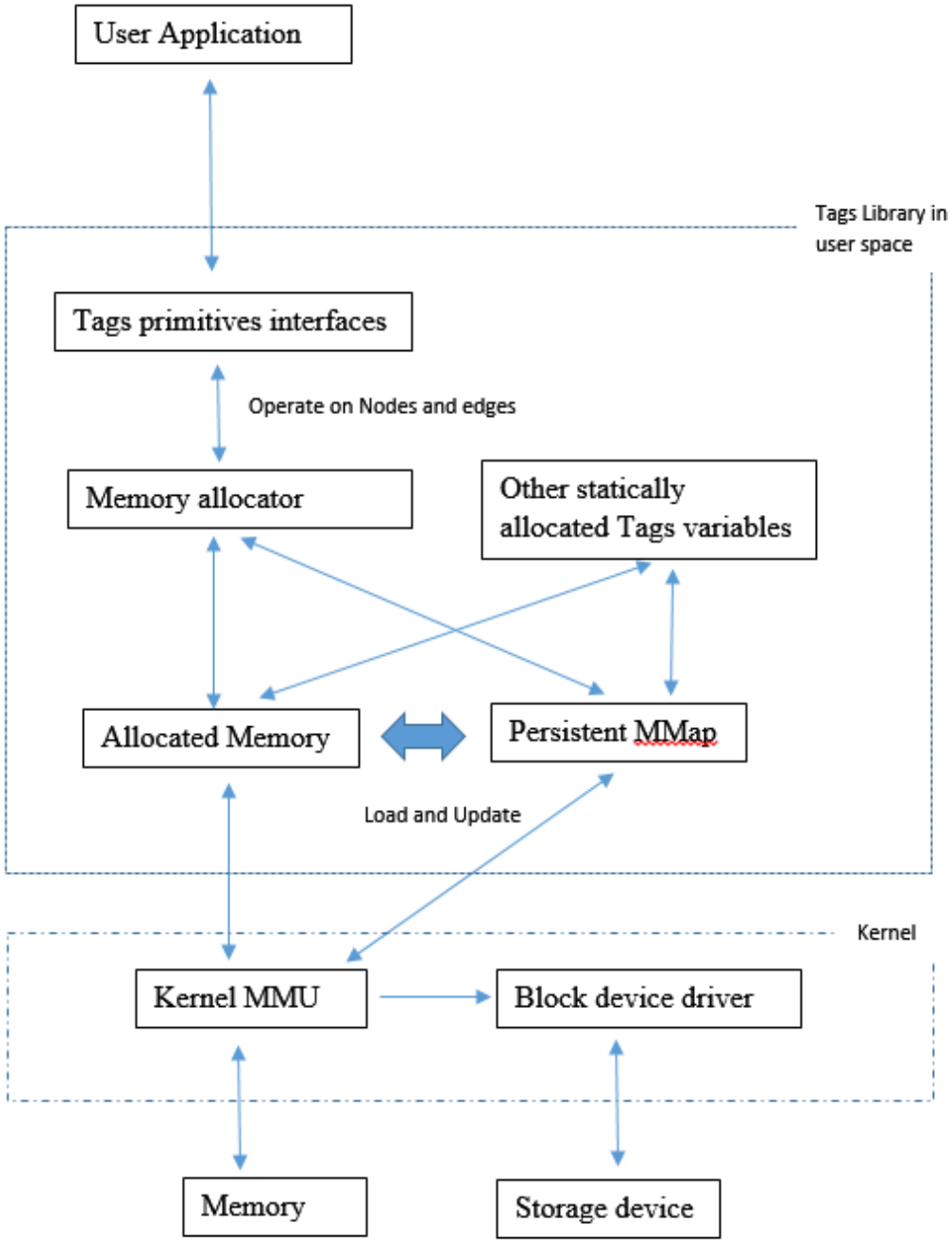


Figure 3.1: Main components of Tags prototype

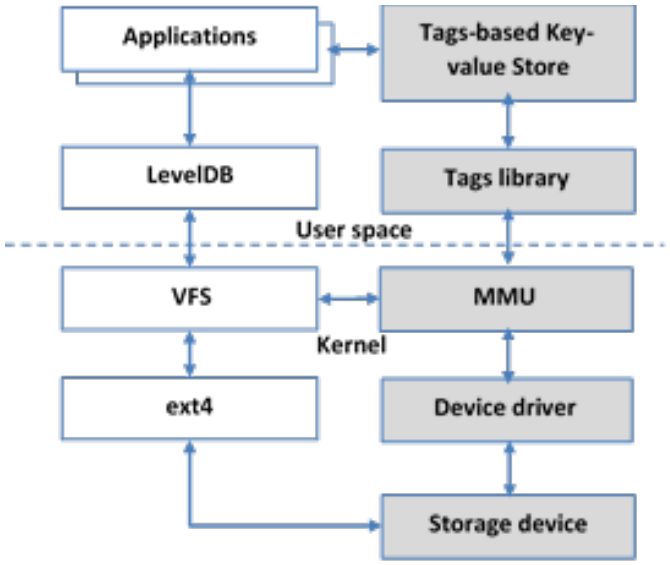


Figure 3.2: Storage data paths for Tags-based key-value store (shaded boxes) and LevelDB [Ghemawat and Dean 2019]

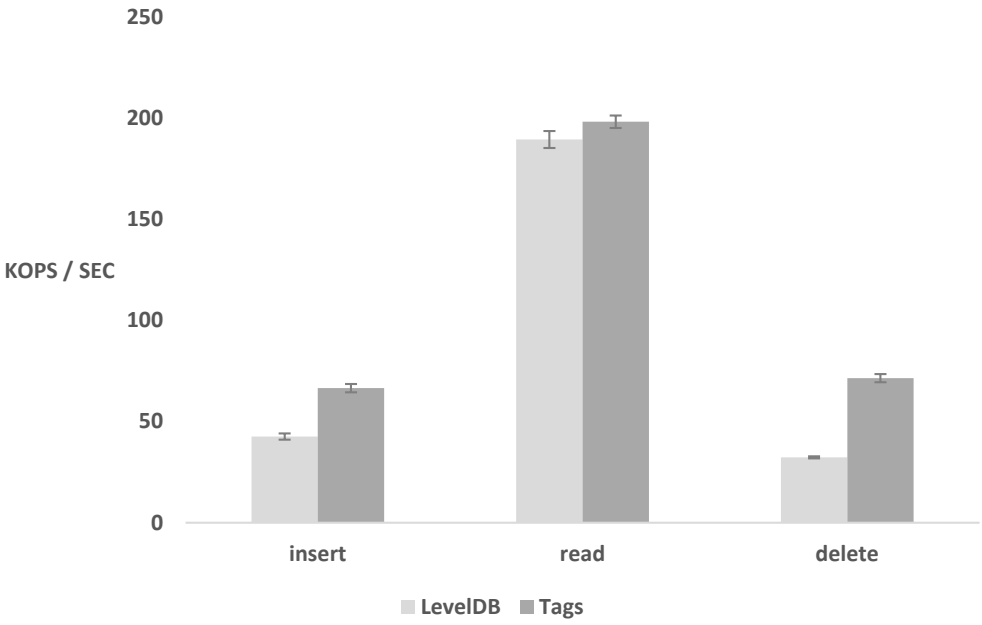


Figure 3.3: Key-value store performance for HDD

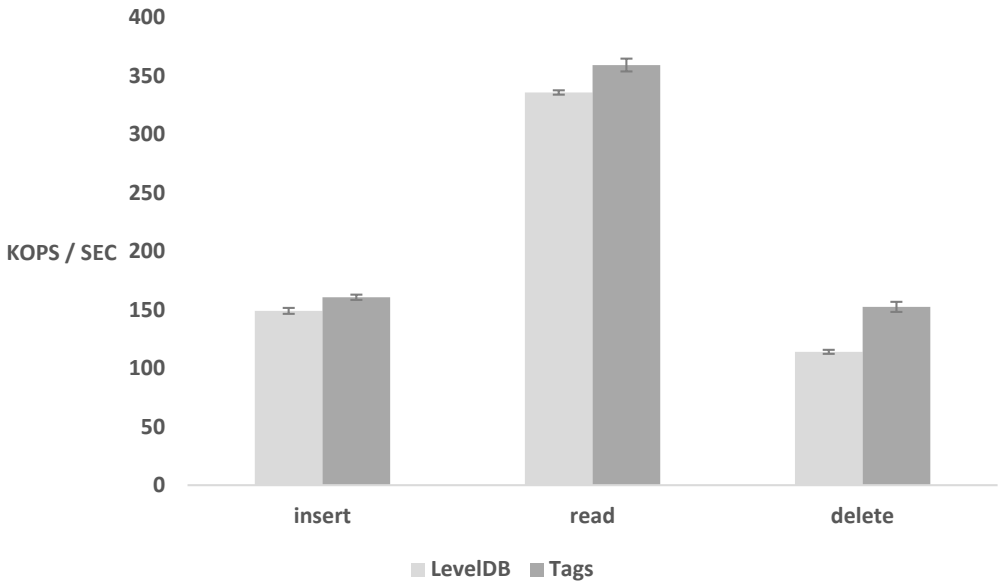


Figure 3.4: Key-value store performance for SSD

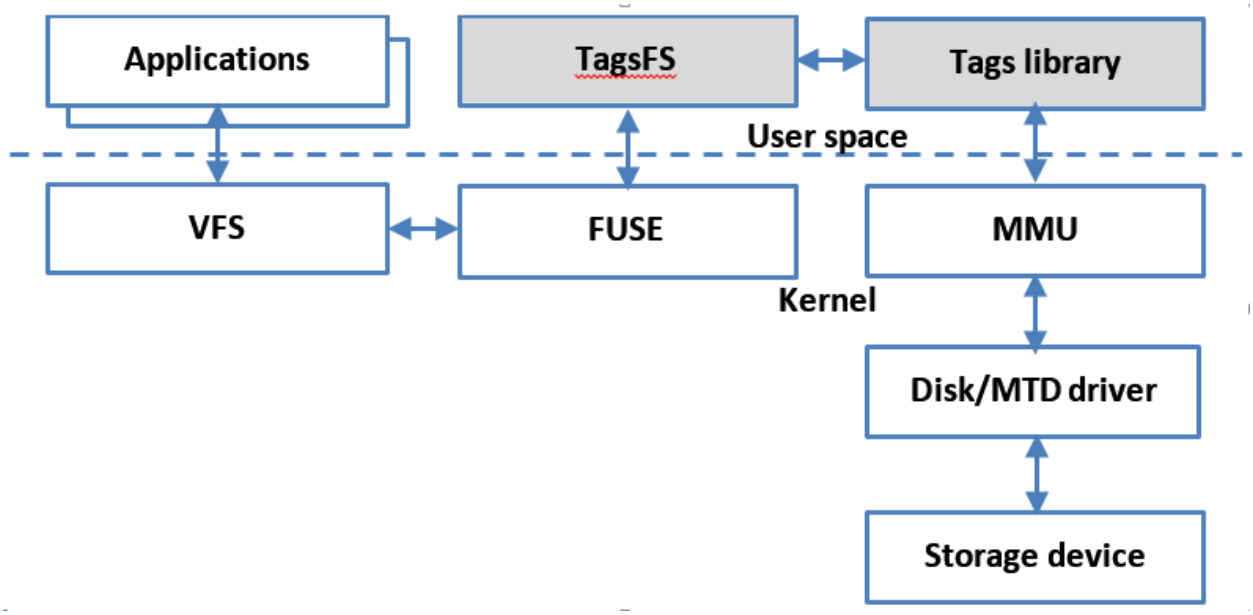


Figure 3.5: TagFS and the Tags library (shaded)

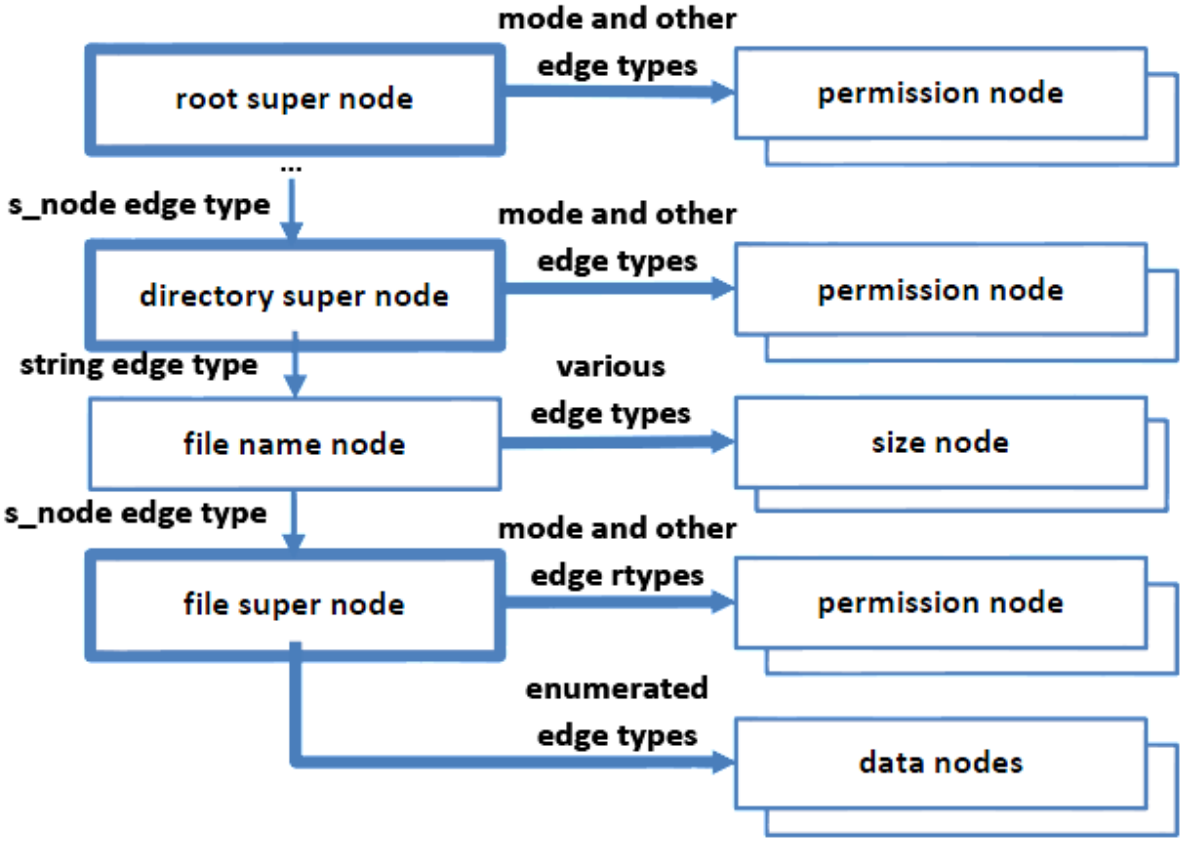


Figure 3.6: The Tags representation of file system

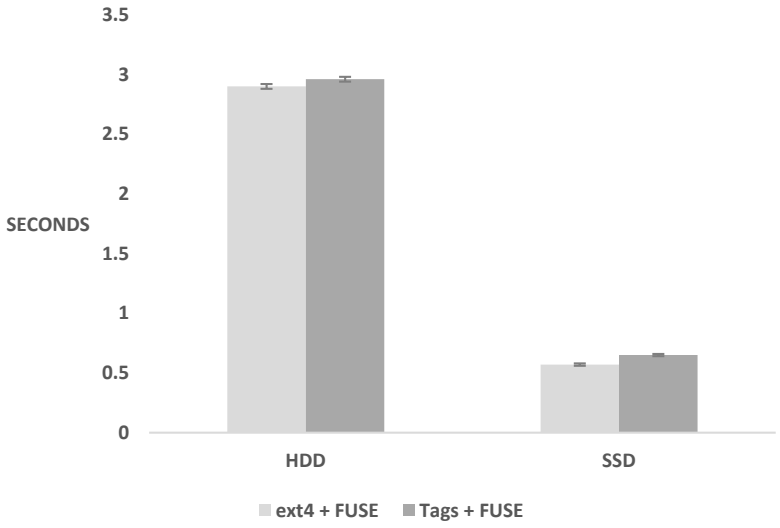


Figure 3.7: Time to recursively list Linux build

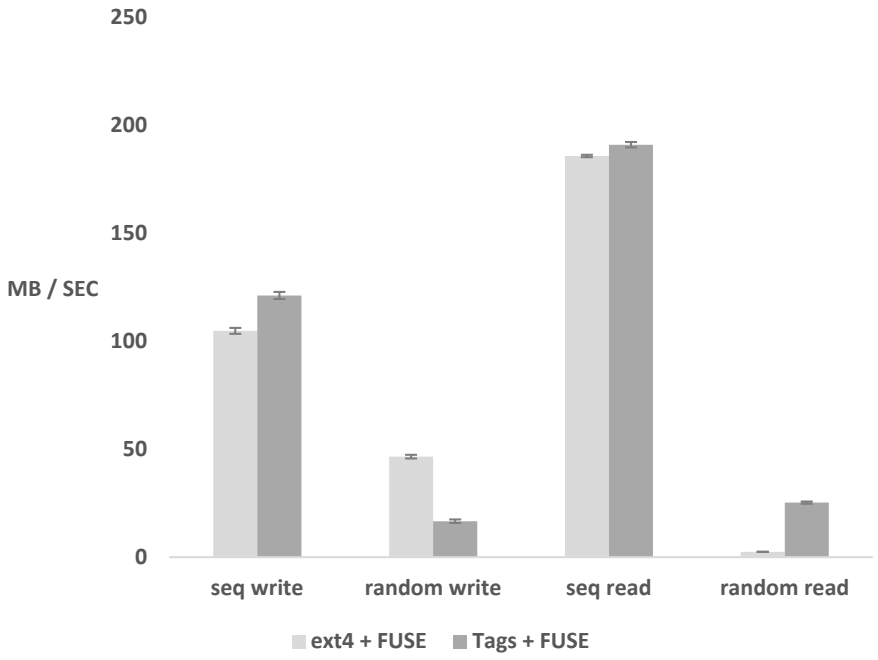


Figure 3.8: LFS large-file benchmark numbers with one 2GB file for HDD

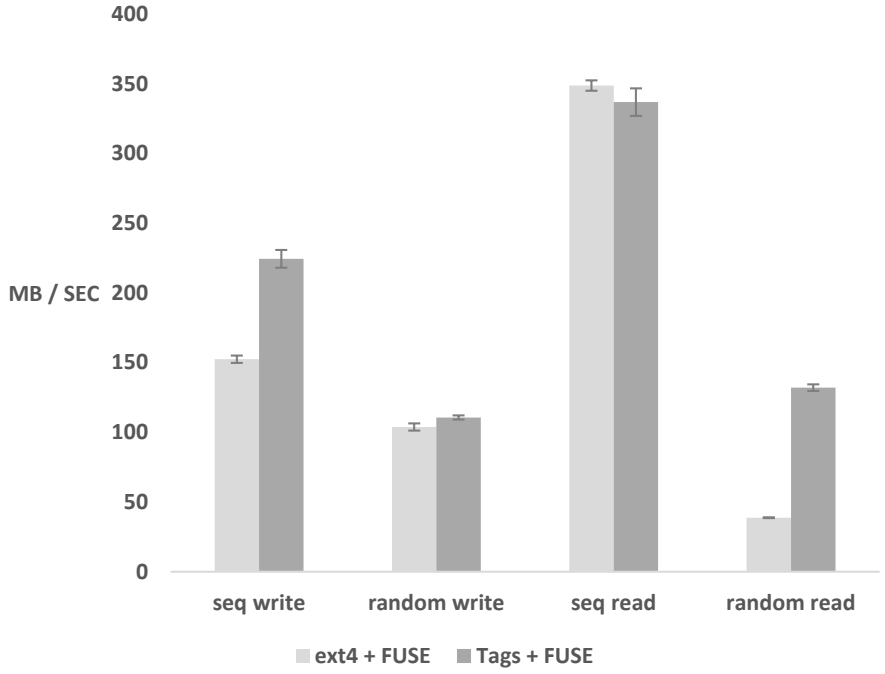


Figure 3.9: LFS large-file benchmark numbers with one 2GB file for SSD

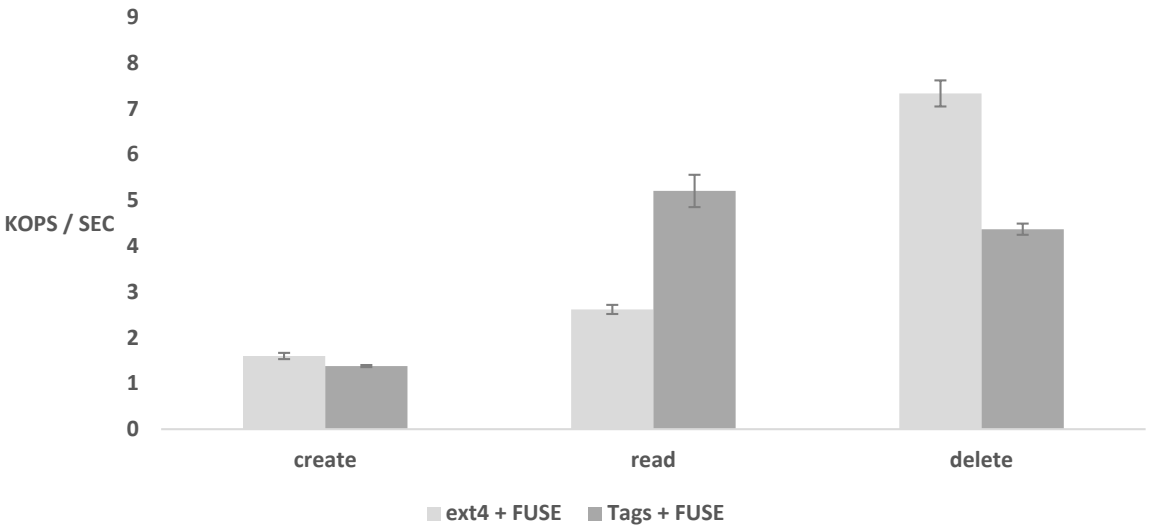


Figure 3.10: LFS small-file benchmark numbers with 20K 16KB file for HDD

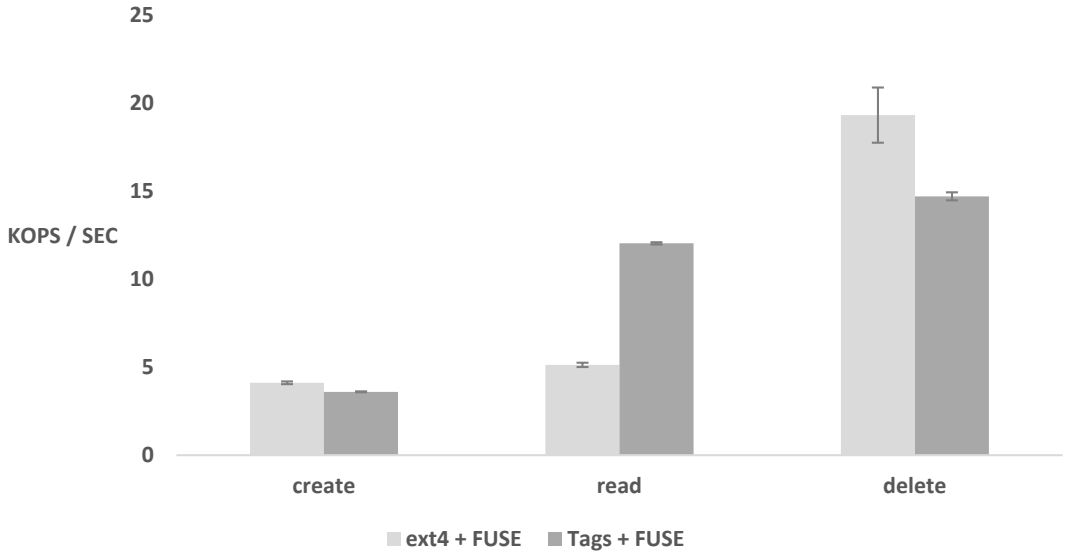


Figure 3.11: LFS small-file benchmark numbers with 20K 16KB file for SSD

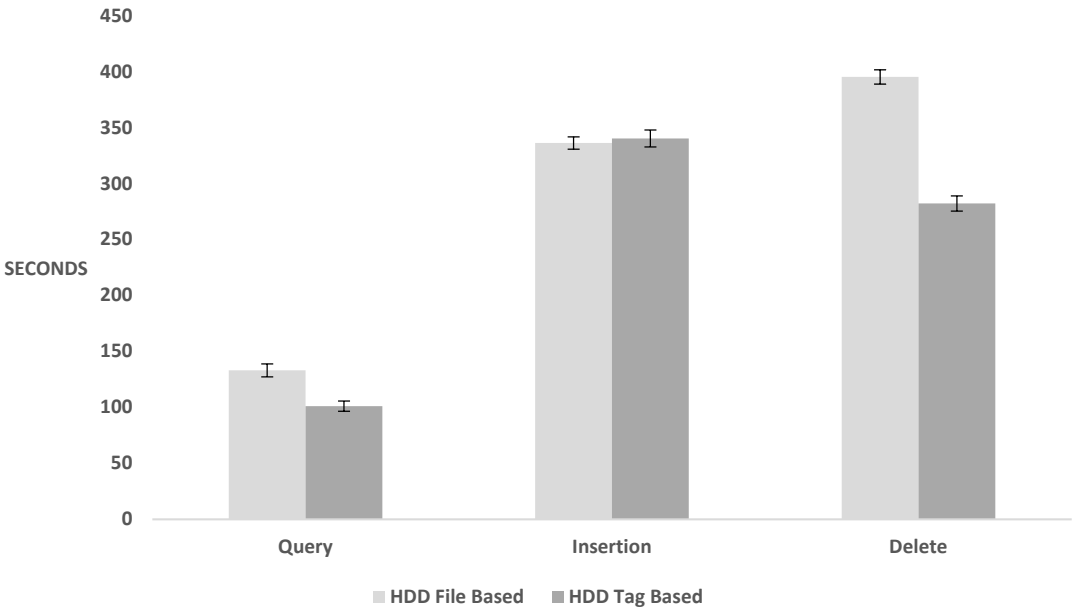


Figure 3.12: Query/insertion/deletion performance (time consumed) comparisons for HDD: Tags-based B+ tree vs file based B+ tree

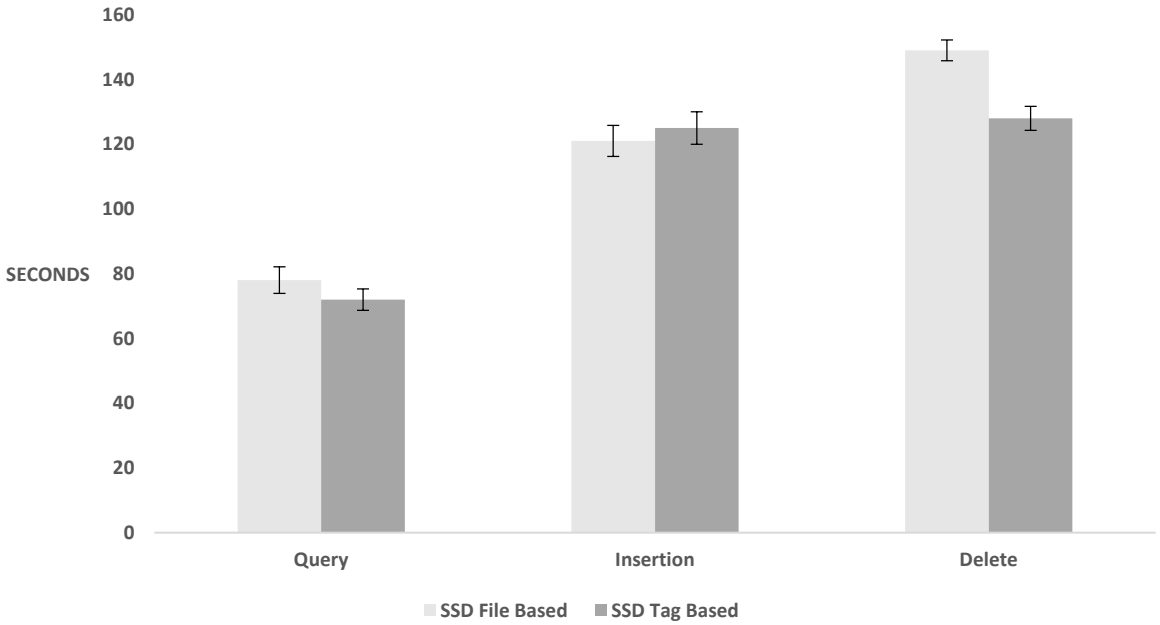


Figure 3.13: Query/insertion/deletion performance (time consumed) comparisons for SSD: Tags-based B+ tree vs file based B+ tree

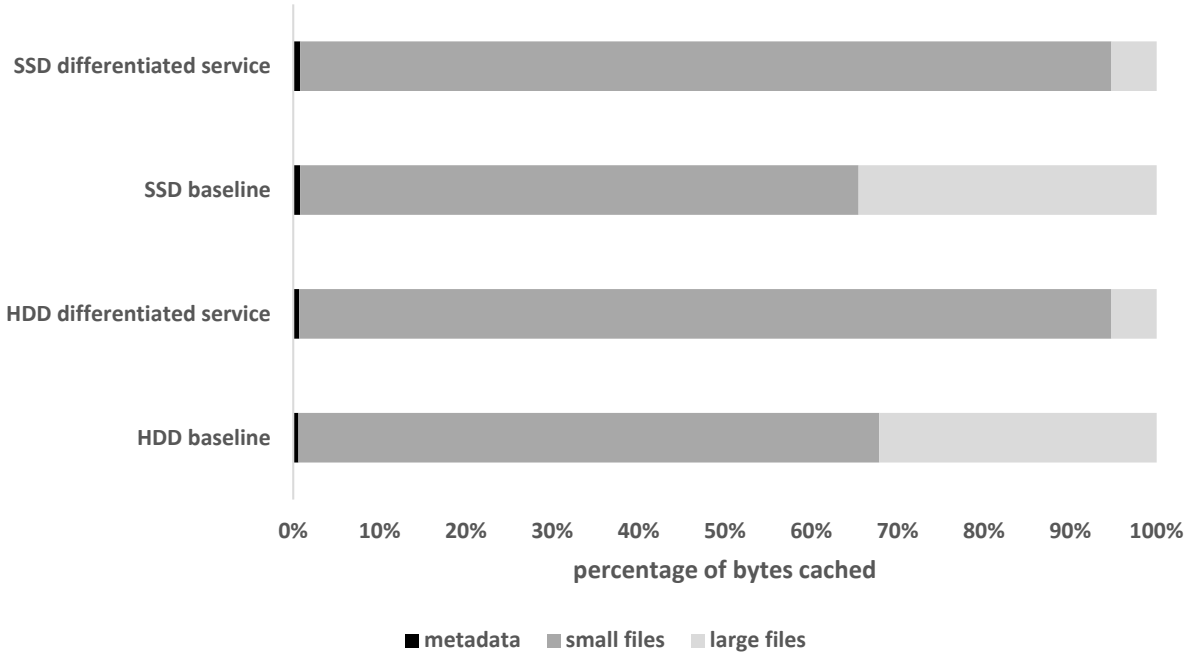


Figure 3.14: Percentage of bytes cached for each IO class

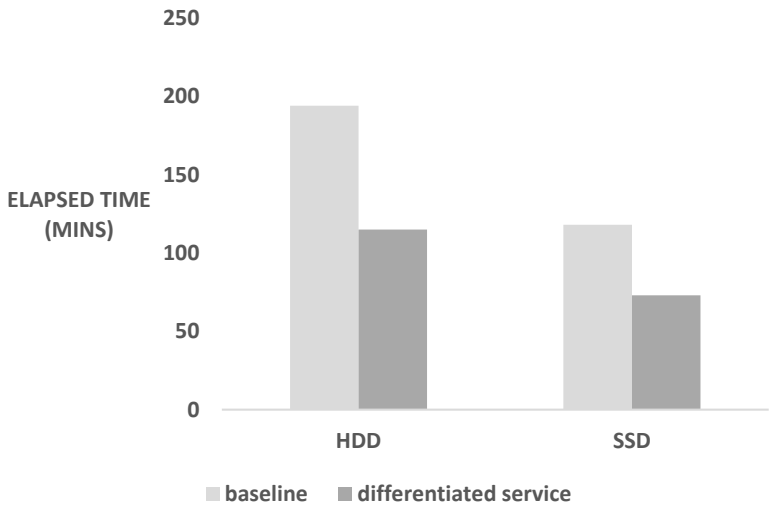


Figure 3.15: Elapsed times for web trace replay

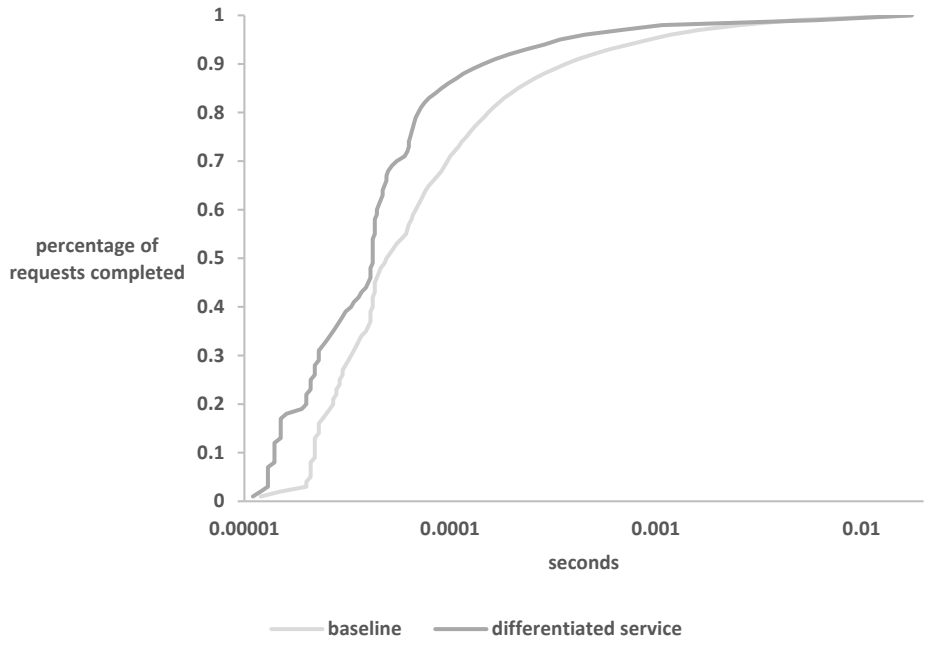


Figure 3.16: CDF for HDD with and without differentiated storage services

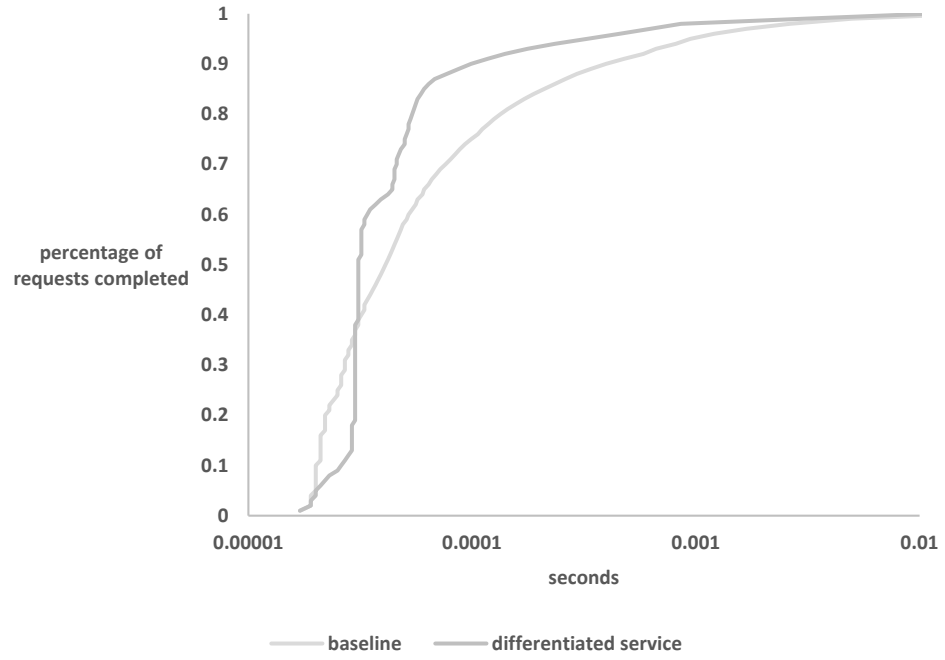


Figure 3.17: CDF for SSD with and without differentiated storage services

CHAPTER 4

CONCLUSION, LESSONS LEARNED AND FUTURE WORK

4.1 Conclusion

We have presented Tags, a white-box approach to addressing constraints of the legacy storage data path. Using a unifying primitive and an API of nodes and edges, we have shown how Tags can be used to build applications as complex as a file system and robust enough to compile the Linux kernel. The Tags-based key-value store shows how direct system support and bypassing redundant services can significantly improve performance for both disks and SSDs. Tags also eases data-path-wide tracking and coordination to support features such as differentiated storage services and per-file secure deletion. Upon benchmark and assessment, the Tags system shows promising performance and scores. However, Tags does have its own limitations. By thinking about these limitations and reflection on lessons we learned during the process, we gain knowledge on whole picture of storage evolution, as well as direction in which we would improve Tags in the future.

4.2 Tags Limitations

Tags provides a framework for different layers to coordinate through a single unified interface, but it does not necessarily mean such coordination is possible. For example, consider two databases that want to share the same dataset to optimize the data layout: one for row access, and the other for column access. How to reconcile conflicting goals across storage components that want to coordinate is beyond the scope of this work.

The white-box approach will also interact with storage software components developed by people who wish to keep their internal representations proprietary. One possible solution is to enrich the

session semantics, so that only a limited subset of translated edge-type IDs is made available for external use.

4.3 Lessons Learned and Future Work

Tags takes a minimalist approach to designing and building a storage data path. The idea seemed simple. However, Tags began as an analogue of sticky notes and was transformed into graph nodes and edges, implemented with the semantics of single-level stores and representations akin to those of key-value stores. Little did we know that this journey would lead us to revisit numerous legacy concepts and design decisions and give us a better appreciation of storage advances.

The following are several thoughts concluded from our experience developing Tags.

- *Low-level single-level store model is tricky to program:*

When building the core Tags, low-level single-level-store style programming was confusing at times. Since the memory allocator and all of its allocated memory regions are persistent, all changes to memory data structures may result in unintended IOs. To overcome this hurdle, we separated persistent and transient data structures. Fortunately, users need only to handle node and edge IDs.

- *Locality is still important for hashing:*

We avoided hashing repeated path prefixes using the parent path ID as a seed to short-circuit the hash functions; however, effectively, this scheme made hashing hierarchical. Our future work will find additional ways to improve the locality of hashing.

- *Access control dictates the unit of access:*

Although Tags allows fine-grained data representation and organization beyond the granularity of legacy data structures, the access control dictates which groups of tags are accessed together and how they can form graphs.

- *Convoluted path forward:*

The Tags design reintroduced certain aspects of the legacy storage data path components (e.g., group operations). However, once we pierce through the legacy data structures, with fine-grained system calls, we can directly support data and metadata layouts not previously possible [Zhang et al. 2016].

- *Multiuser and multithread support:*

Tags needs to expand support for multithread to be full-fledged storage system. Also, adding multithread support would potentially offer opportunities for optimization. As stated in Section 1.3, another task for future implementation is trying to exploit parallelism in modern hardware. For a coarse-grained traditional file-based approach, saving calculation and computation for metadata would not present a significant benefit, reflective of hardware resources and capability at that time. But Tags has many more metadata items such as edge entries. Saving overhead on each operation, such as hash table operation/hash function, would add up to a substantial gain. Examples to take advantage of hardware include the following:

  - Take Intel TSX [Intel TSX] capability from eligible Intel CPU to enhance transaction – hash table operation performance.
  - Take GPU capability to enhance hash value calculation.
  - Take new CPU internal SHA functions to enhance hash value calculation.

These may make a big difference in the Tags performance demonstration.

- *Tags in kernel*

There are pros and cons to porting Tags into kernel. The code would be exposed to many constraints, including a memory space limit and an available function set much less than in user space. However, it would be much easier to demonstrate Tags' design goal for cross-layer coordination and tracking, and because all underlying layers are in kernel space, Tags would be able to communicate with them without breaking the user space–kernel space barrier. For example, it would be easy to implement differentiated storage services and per-file secure deletion, as in Sections 3.2.4 and 3.2.5.

REFERENCES

[Ames et al. 2006] Ames S, Bobb N, Greenan KM, Hofmann OS, Storer MW, Maltzahn C, Miller EL, Brandt SA. LiFS: An Attribute-Rich File System for Storage Class Memories. *Proceedings of the 23rd IEEE/14th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)*, 2006.

[Arpaci-Dusseau and Arpaci-Dusseau 2001] Arpaci-Dusseau AC, Arpaci-Dusseau RH. Information and Control in Gray-box Systems. *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP)*, 2001.

[Arpaci-Dusseau et al. 2006] Arpaci-Dusseau AC, Arpaci-Dusseau RH, Bairavasundaram LN, Denehy TE, Popovici FI, Prabhakaran V, Sivathanu M. Semantically-smart Disk Systems: Past, Present, and Future. *Proceedings of the 2006 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2006.

[Bez and Pirovano 2004] Bez R, Pirovano A. Non-volatile memory technologies: emerging concepts and new materials. *ElsevierMaterials in Semiconductor Processing*, 7(4):349-355, November 2004.

[Bjorling et al. 2017] Bjorling M, Gonzalez J, Bonnet P. LightNVM: The Linux Open-Channel SSD SubSystem. *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, 2017.

[Bonwick 1994] Bonwick J. The Slab Allocator: An Object-Caching Kernel Memory Allocator. *Proceedings of the USENIX Summer 1994 Technical Conference (ATC)*, 1994.

[Diesburg et al. 2012] Diesburg S, Meyers C, Stanovich M, Mitchell M, Marshall J, Gould J, Wang AIA, Kuenning G. TrueErase: Per-file Secure Deletion for the Storage Data Path. *Proceedings of the 2012 ACM Annual Computer Security Applications Conference (ACSAC)*, 2012.

[Duller et al. 2014] Dulloor SR, Kumar S, Keshavamurthy A, Lantz P, Reddy D, Sankaran R, Jackson J. System Software for Persistent Memory. *Proceedings of 2014 European Conference on Computer Systems (EuroSys)*, 2014.

[Fagin et al. 1979] Fagin R, Nievergelt J, Pippenger N, Strong HR. Extensible Hashing—A Fast Access Method for Dynamic Files. *ACM Transactions on Database Systems*, 4(3):315-344, 1979.

[Ghemawat and Dean 2019] Ghemawat S, Dean J, LevelDB, https://github.com/google/leveldb, 2018.

[Intel 2016] Introduction to the Storage Performance Development Kit (SPDK), Intel Developer Zone, https://software.intel.com/en-us/articles/introduction-to-the-storage-performance-development-kit-spdk, 2016

[Intel TSX] Intel® Transactional Synchronization Extensions (Intel® TSX) Overview, Intel Compiler Documentation, https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-intel-transactional-synchronization-extensions-intel-tsx-overview

[Kannan et al. 2018] Kannan S, Arpaci-Dusseau AC, Arpaci-Dusseau RH, Wang Y, Xu J, Palani G. Designing a True Direct-Access File System with DevFS. *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, 2018.

[Kwon et al. 2017] Kwon Y, Fingler H, Hunt T, Peter S, Witchel E, Anderson T. Strata: A Cross Media File System. *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, 2017.

[Lakshman and Malik 2010] Lakshman A, Malik P 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2): 35-40, 2010.

[Liu et al. 2019] Liu J, Arpaci-Dusseau AC, Arpaci-Dusseau RH, Kannan S. *Proceedings of the 11th USNEIX Workshop on Hot Topics in Storage and File Systems*, 2019.

[Lu et al. 2016] Lu L, Pillai TS, Arpaci-Dusseau AC, Arpaci-Dusseau RH, WiscKey: Separating Keys from Values in SSD-Conscious Storage. *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[Mesnier et al. 2011] Mesnier M, Chen F, Luo T, Akers JB. Differentiated Storage Services. *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[OpenSSL 2019] OpenSSL Cryptography and SSL/TLS Toolkit. https://www.openssl.org/news/openssl-1.1.0-notes.html, 2019.

[Ousterhout et al. 2010] Ousterhout J, Agrawal P, Erickson D, Kozyrakis C, Leverich J, Mazieres D, Mitra S, Narayanan A, Parulkar G, Rosenblum M, Rumble SM, Stratmann E, Stutsman R. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review*, 43(4):92-105, 2010.

[Peter et al. 2014] Peter S, Li J, Zhang I, Ports DRK, Woos D, Krishnamurthy A, Anderson T, Roscoe T. Arrakis: The Operating System is the Control Plane. *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[Peterson and Norman 1997] Peterson JL, Norman TA. Buddy Systems. *Communications of the ACM* 20(6):421-431, 1997.

[Ren and Gibson 2013] Ren K, Gibson G. TABLEFS: Enhancing Metadata Efficiency in the Local File System. *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, 2013.

[Rodeh et al. 2013] Rodeh O, Bacik J, Mason C. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3), Article No. 9, 2013.

[Rosenblum and Ousterhout 1992] Rosenblum M, Ousterhout JK. The Design and Implementation of a Log-structured File System. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26-52, 1992.

[Rumble et al. 2014] Rumble SM, Kejriwal A, Ousterhout J. Log-structured Memory for DRAM-based Storage. *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[Seshadri et al. 2014] Seshadri S, Gahagan M, Bhaskaran S, Bunker T, De A, Jin Y, Liu Y, Swanson S, 2014. Willow: A User-programmable SSD. *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[Shen et al. 2014] Shen K, Park S, Zhu M. Journaling of Journal is (Almost) Free. *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[Shetty et al. 2013] Shetty PJ, Spillane RP, Malpani RR, Andrews B, Seyster J, Zadok E, Building workload-independent storage with VT-trees. *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[Shin 2005] Shin Y. Non-volatile memory technologies for beyond 2010. *Digest of Technical Papers. Proceedings of the IEEE 2005 Symposium on VLSI Circuits*, 2005.

[Shu and Obr 2007] Shu F, Obr N. Data set management commands proposal for ATA8-ACS2. http://www.t13.org/documents/UploadedDocuments/docs2007/e07154r2-Data_Set_Management_Proposal_for_ATA-ACS2.pdf, 2007.

[Sivathanu et al. 2003] Sivathanu M, Prabhakaran V, Popovici FI, Denehy TE, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Semantically-Smart Disk Systems. *Proceedings of the Second USENIX Symposium on File and Storage Technologies (FAST)*, March 2003.

[Sivathanu et al. 2004] Sivathanu M, Bairavasundaram LN, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Life or Death at Block Level. *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[Sivathanu et al. 2005] Sivathanu M, Arpaci-Dusseau AC, Arpaci-Dusseau RH, Jha S. A Logic of File Systems. *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)*, 2005.

[Statista, 2017] *https://www.researchgate.net/figure/Shipments-of-hard-and-solid-state-disk-drives-HDD-SSD-in-millions-worldwide-in_fig1_323005275*

[Sun et al. 2018] Sun K, Fryer D, Chu J, Lakier M, Brown AD, Goel A. Spiffy: Enabling File-System Aware Storage Applications. *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*, 2018.

[Sun Microsystems 2004] Sun Microsystems. In a Class by Itself—the Solaris 10 Operating System, A Technical White Paper, November 2004.

[Swanson and Caulfield 2013] Swanson S, Caulfield A. Refactor, Reduce, Recycle: Restructuring the I/O Stack for the Future of Storage. *Computer*, 46(8):52-59, August 2013.

[Szeredi 2005] Szeredi M. Filesystem in Userspace. http://fuse.sourceforge.net, 2005.

[Volos et al. 2014] Volos H, Nalli S, Panneerselvam S, Varadarajan V, Saxena P, Swift MM. *Proceedings of the 2014 European Conference on Computer Systems (EuroSys)*, 2014.

[Wang et al. 2002] Wang AI, Reiher PL, Popek GJ, Kuenning GH, Conquest: Better Performance through a Disk/Persistent-RAM Hybrid File System. *Proceedings of the 2002 USENIX Annual Technical Conference (ATC)*, 2002.

[Wilcox 2014] Wilcox M. DAX: Page Cache Bypass for Filesystems on Memory Storage. https://lwn.net/Articles/618064, 2014.

[Woodhouse 2001] Woodhouse D. JFFS: The Journaling Flash File System. *Proceedings of the Ottawa Linux Symposium*, 2001.

[Zhang et al. 2016] Zhang S, Catanese H, Wang AIA. The Composite-file File System: Decoupling the One-to-one Mapping of Files and Metadata for Better Performance. *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.

BIOGRAPHICAL SKETCH

Weisu Wang received his M.S. and B.S. in Electrical Engineering in 2004 and 2001 from Tsinghua University. He was a software engineer at Alcatel-Lucent from 2004 to 2008. He received his M.S. in Computer Science at Florida State University in 2014, where he continued with his pursuit of PhD. His research interests include file systems, operating systems, storage architecture, emerging storage hardware, and distributed systems.