Journal of Systems Architecture 160 (2025) 103348

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

StorStack: A full-stack design for in-storage file systems

Juncheng Hu, Shuo Chen, Haoyang Wei, Guoyu Wang, Chenju Pei, Xilong Che *

College of Computer Science and Technology, Jilin University, Chang Chun, 130022, China

ARTICLE INFO

Keywords:
File system
In-storage Computing
Storage-class Memory

ABSTRACT

Due to the increasingly significant cost of data movement, In-storage Computing has attracted considerable attention in academia. While most In-storage Computing works allow direct data processing, these methods do not completely eliminate the participation of the CPU during file access, and data still needs to be moved from the file system into memory for processing. Even though there are attempts to put file systems into storage devices to solve this problem, the performance of the system is not ideal when facing high latency storage devices due to bypassing the kernel and lacking page cache.

To address the above issues, we propose StorStack, a full-stack, highly configurable in-storage file system framework, and simulator that facilitates architecture and system-level researches. By offloading the file system into the storage device, the file system can be closer to the data, reducing the overhead of data movements. Meanwhile, it also avoids kernel traps and reduces communication overhead. More importantly, this design enables In-storage Computing applications to completely eliminate CPU participation. StorStack also designs the user-level cache to maintain performance when storage device access latency is high. To study performance, we implement a StorStack prototype and evaluate it under various benchmarks on QEMU and Linux. The results show that StorStack achieves up to 7x performance improvement with direct access and 5.2x with cache.

1. Introduction

In traditional computing architectures, data must be transferred from storage devices to memory for processing, which not only consumes the computing resources of the host, but also results in high energy consumption and I/O latency. As data scales continue to expand, In-storage Computing has been proposed to alleviate the pressure of data movement [1,2]. The core idea is to perform computations directly where the data is stored, without the need to move the data. The emergence of high-speed storage devices like SSDs [3] and SCMs [4,5] has significantly advanced research in In-storage Computing and transformed computer storage systems. To fully leverage the potential of storage systems and exploit the characteristics of this new computing paradigm, a redesign of storage stack software is required.

As the most essential part of the storage stack software, file systems have been residing in the operating system kernel for a very long time because they need to perform integrity assurance and access control to ensure data security. The kernel is considered a trusted field compared to the user space. However, this seemingly good design has been challenged by new technologies. With the emergence of faster storage devices such as SSDs and SCMs, access latency decreases significantly compared to HDDs [6], leading to the software overhead of file systems [7,8] becoming a major performance bottleneck. Meanwhile, the design and operation of file systems determine their reliance on the CPU when accessing the file system. For In-storage Computing, although researchers are gradually reducing CPU involvement, current file systems still rely on the CPU to handle complex file management tasks and ensure system security and integrity.

On the one hand, to reduce the software overhead of file systems, many works aim at the kernel trap. For example, there are some efforts to move the file system into user space [8–13]. But running in user space may compromise the reliability of the file system, hence bugs or malicious software may cause crashes and data loss. Some of these works try to move the critical parts of the file system back to the kernel. But in most cases, data-plane operations are interleaved with control-plane operations, which may diminish the performance improvement brought by kernel bypassing. In recent years, firmware file systems have been proposed, which move file systems onto the storage device controller [14–16] to completely get rid of the kernel trap. However, those file systems are designed to be strongly coupled with the storage device, making the device lack the replaceability of file system and the compatibility with conventional operating systems. In addition, these firmware file systems do not provide comprehensive security guarantees.

* Corresponding author.
E-mail addresses: jchu@jlu.edu.cn (J. Hu), chenshuo22@mails.jlu.edu.cn (S. Chen), hywei23@mails.jlu.edu.cn (H. Wei), wgy21@mails.jlu.edu.cn (G. Wang), peicj2121@mails.jlu.edu.cn (C. Pei), chexilong@jlu.edu.cn (X. Che).

https://doi.org/10.1016/j.sysarc.2025.103348
Received 29 August 2024; Received in revised form 24 November 2024; Accepted 18 January 2025
Available online 27 January 2025

On the other hand, to fully leverage the advantages of In-storage Computing, it is necessary to eliminate the participation of host-side OS from the storage access path. In-storage Computing advocates for a data-centric approach, where computation units are embedded within the storage devices to enable direct data processing. However, in the process of accessing files, traditional file systems still require CPU involvement. To know which data should be transferred next, file access should be first handled by the host-side file system in the operating system kernel. This CPU intervention limits the computational capacity improvements that In-storage Computing can offer.

Another point worth noting is that numerous studies propose improving system performance by allowing user applications to bypass the kernel and communicate directly with storage devices. This method demonstrates significant performance improvements when dealing with high-speed storage devices. However, due to the diversity of storage devices and their varying latencies, system performance may suffer when bypassing the high-speed cache, especially when using high-latency, low-speed storage devices. Therefore, the impact of cache configuration on performance is also a subject of our further research. In summary, despite various attempts to optimize file systems performance and reduce CPU involvement, current solutions still have several issues.

To further optimize the performance and security of file systems and fully unleash the potential of in-storage computing, we propose StorStack, which is a full-stack, highly configurable, in memory file system framework and simulator on high-speed storage devices such as SSDs and SCMs. Since file systems always have a fixed primary functionality of managing the data mapping, which is similar in function to the flash translation layer (FTL) on the storage controller, we consider it natural and reasonable to run the file system on the storage controller.

StorStack has three main components: a device firmware runtime for file systems enabling file systems to run directly on the storage device, a user library to expose POSIX interfaces to user applications, and a kernel driver to guarantee access control. By moving the file system into the storage, StorStack aims to gain performance improvement from the concept of In-storage Computing that brings the file system closer to the data. Moreover, the file system code is removed from the kernel, which can avoid the latency and context switches caused by kernel traps during file access. More importantly, StorStack can remove the CPU from the storage access path of In-storage Computing applications, maximizing the potential of In-storage Computing. To ensure the security and reliability of the file system, StorStack has designed an efficient security mechanism, introducing a device-side controller as the runtime and retaining control plane operations within the host kernel. By reducing the ratio of control plane to data plane operations, kernel traps are minimized, enhancing performance. StorStack also includes a user-level cache to explore the impact of cache on the performance of in-storage file systems.

We implemented StorStack as a prototype and evaluated it on QEMU and Linux 5.15. Experimental results demonstrate that StorStack performs up to 5.2x faster times than Ext-4 with cache and 7x times with direct access. Regarding the cache, we find that as access latency increases, file systems with cache always maintain high speeds, whereas the speed of file systems without cache decreases significantly.

2. Background and related work

The storage or memory system has changed a lot in the past decades. With the development of speed, capacity, and size, and the emergence of new types of storage, a rethink of both hardware and software is required to exploit the potential of the system in the next era. In this section, we first discuss the trends of two novel high-speed non-volatile storage, and then explored the significance of applying In-storage Computing on these storage devices. Finally, we briefly introduce three file systems in different locations.

2.1. Hardware trends

Compared to the large, slow HDD, solid-state drive (SSD) is a kind of flash-based non-volatile storage with small form factor, high speed, and low energy costs [17,18]. SSDs on the market today can provide up to 30 TB of capacity and 7 GB/s throughput on sequential read/write. To fully exploit the high performance, modern SSDs have switched from SATA to PCIe and NVMe. PCIe 5.0 [19] supports up to 16 lanes and 32 GT/s data rate, which leads to more than 60GB/s bandwidth. NVMe [3] is a communication protocol for non-volatile memories attached via PCIe, supporting up to 65,535 I/O queues each with 65,535 depth. It also supports SSD-friendly operations like ZNS and KV, which can help SSDs further enhance SSDs' throughput capabilities.

Storage class memory (SCM), also referred to as persistent memory (PMEM) or non-volatile memory (NVM), is a different type of storage device that is fast and byte-addressable like DRAM, but can also retain data without power like SSDs. Various technologies such as PRAM [20,21], MRAM [22], and ReRAM [23,24]have been explored to implement SCM, each exhibiting different performance characteristics. SCM provides higher bandwidth than SSD; it offers latency close to DRAM, and its capacity falls between SSD and DRAM [25]. As a new blood in the storage hierarchy, SCM can provide more possibilities to multiple workloads [26–29].

Consequently, while the increased bandwidth and reduced latency of storage devices have substantially boosted the performance of computer systems and enabled novel application scenarios, these advancements also introduce several challenges. These challenges include heightened complexity in data management, the need to balance cost and efficiency, and issues related to technical compatibility and migration.

2.2. In-storage computing

While these new storage devices have significantly altered the memory hierarchy of computer systems, the memory wall between CPU and off-chip memory is still the bottleneck of the whole system, especially with the rise of data-intensive workloads and the slowdown of Moore's law and Dennard scaling. To reduce the overhead of data movement, In-storage Computing(ISC) [30–32]is proposed, gaining increasing attention with advancements in integration technologies. However, most current research predominantly focuses on offloading user-defined tasks to storage devices, and this approach still faces limitations in practice.

First, existing ISC methods exhibit significant shortcomings in terms of compatibility and portability. On the host side, developers must design custom APIs for ISC, which are incompatible with existing system interfaces such as POSIX, demanding substantial modifications to the host code [32]. On the drive side, the drive program either collaborates with the host file system to access the correct file data [33] or manages the drive as a bare block device without a file system. However, most systems still rely on file system-based external storage access, with the file system typically running on the CPU. Consequently, ISC tasks often require CPU involvement when accessing external storage data.

Secondly, current approaches lack adequate protection and isolation for ISC applications. To fully leverage the high speed of modern storage devices, multiple ISC applications may need to execute concurrently. Without proper data protection mechanisms, malicious or erroneous ISC tasks could access unauthorized data. Without isolation, the execution of one ISC task could compromise the performance and security of others. However, most existing research [1,34,35] assumes that ISC tasks operate in an exclusive execution environment, failing to address these concerns effectively. Additionally, when specific code is offloaded to storage devices, attackers can exploit vulnerabilities in in-storage software and hardware firmware, such as buffer overflows [36,37] or bus snooping attacks, to escalate privileges and harm the system.

2.3. File system

The evolution of storage hardware poses higher demands for software systems. As a crucial part of the software stack of the storage system, file systems should be redesigned to minimize software overheads, especially the involvement of the OS kernel on the data path. Many efforts have explored the possibility of different file system locations.

Kernel file systems. Numerous typical file systems are implemented inside kernel as kernel file systems, including Ext4, XFS, etc. Due to the isolation of kernel space, kernel file systems can easily manage data and metadata with reliability guarantees [38]. Recent works on kernel file systems have sought to exploit the capabilities of modern storage devices. For example, F2FS [39] is built on append-only logging to adapt to the characteristics of flash memory. PMFS [38] introduce a new hardware primitive to avoid the consistency issues caused by CPU cache while accessing SCM. DAX [40] bypasses the buffer cache of the system to support direct access to the storage hardware so that the redundant data movement between DRAM and SCM is removed. NOVA [41] explores the hybrid of DRAM and SCM as a specially designed log-structured file system. However, kernel file systems have several limitations. Firstly, the development and debugging process within kernel space is inherently complex and difficult. Furthermore, every file system access necessitates a kernel trap, which inevitably introduces latency. Additionally, the frequent context switching between user processes and the kernel increases CPU overhead.

User-space file systems. User-space file systems are implemented mostly in user space to bypass the kernel and reduce the overhead associated with kernel traps. However, since most user-space file systems are implemented in untrusted environments, ensuring data security and reliability becomes challenging. User-space file systems need sophisticated design, usually the collaboration between kernel space and user space, to keep them reliable. For example, Strata [11] separate the file system into a per-process user space update log for concurrent writing and a read-only kernel space shared area for data persistence. Moneta-D [9] provides a hardware virtual channel support with kernel space file system protection policy and a user space driver to access the hardware. There are also efforts to implement the control-plane of the file system as a trusted user space process [8,12].

Firmware file systems. Works that offload part or the whole of the file system into the storage device firmware are categorized as firmware file systems. There are three representative works on firmware file systems: DevFS [14], CrossFS [15] and FusionFS [16]. DevFS and CrossFS explore the possibility of moving the file system to the storage side to benefit from kernel bypass. FusionFS goes further on the previous two works and attempts to gain performance by combining multiple storage access operations. However, we have identified several problems of these file systems. First, these firmware file systems are tightly coupled with specific storage devices, which makes it hard for users to select alternative file systems or upgrade the software version of the current file system. Second, none of these file systems are designed to operate effectively in scenarios with significant communication latency. Third, the lack of security mechanisms limits their applicability in real-world environments.

2.4. Motivation

Although kernel file systems are well-designed and time-tested, their design principles, which assume high device access latency, are no longer suitable for modern high-speed devices. User-space file systems and firmware file systems have explored new approaches to file system implementation in the era of high-speed storage; however, they may lead to inferior performance with traditional devices, compromised security controls, or inflexible, non-replaceable file systems. To address these issues, we introduce StorStack, a fast, flexible, and secure in-storage file system framework. The detailed comparison between StorStack and previous file systems is shown in Table 1.

3. Design

In this section, we first discuss the design principles of StorStack, followed by an overview of its architecture, connection between host and device, scheduling mechanisms and reliability designs.

3.1. Principles

1. Provide a full-stack framework to enable in-storage file systems without compromising performance. To support in-storage FS, StorStack's design includes a user library, a kernel driver, and a firmware FS runtime. By bringing FS code out of the kernel and closer to the data, StorStack avoids the kernel trap and reduces the communication overhead. StorStack also incorporates a user-level cache to maintain the performance when the access latency of the device is high.

2. Make full use of the heterogeneity of the host CPU and storage device controller. The in-storage FS yields the host CPU time to user application codes and cuts the energy cost, while conflicts due to concurrent access are resolved on the host CPU to maintain the performance. If necessary, the cache is also retained on the host side, and is managed by the user space. Such a heterogeneous system can maximize the overall performance and minimize the power consumption of the system.

3. Guarantee the reliability of the file system with minimal overhead. To provide essential guarantees such as permission checking, StorStack keeps its control plane within the trusted area. Additionally, to enhance performance, a token mechanism is introduced to prevent StorStack from accessing the kernel during data-plane operations.

4. Keep compatible with conventional operating systems. The design of StorStack does not require changes to current operating systems. Instead, the user lib and kernel driver of StorStack are add-ons. Even without them, the StorStack storage device can be accessed with typical block- or byte-based interfaces, just like traditional SSDs or SCMs. StorStack also supports per-partition replaceable file systems, which is a regular function in current operating systems but not supported by firmware file systems.

5. Support heterogeneous computing. By providing a device-level file interface, StorStack may enable multiple advanced heterogeneous access patterns, including In-storage Computing (ISC) [31,32,42,43] and direct I/O access from GPUs [44,45] or NICs [42,46]. In this work, we provide basic support for these patterns and plan to further explore them in future research.

6. Run with reasonable hardware setup on the storage device. Previous research on firmware file systems has assumed that device controller hardware capabilities are severely limited. However, today's high-end storage devices feature up to 4 cores and DRAM capacity that can reach 1% of their storage capacity [47]. As in-storage processing evolves, hardware configurations will continue to improve [30,43,48–50]. In StorStack, we assume that the device possesses sufficient capabilities to run file systems alongside a runtime environment. Future research can investigate the benefits of integrating in-storage file systems with additional device-side capabilities, such as power loss protection capacitors or the flash translation layer.

3.2. Architecture

To support in-storage file systems with compatibility, flexibility, and reliability, StorStack has three major parts distributed over user space, kernel space, and device side.

Table 1
The detailed comparison between StorStack and previous file systems.

	Software access latency	Expected hardware latency	FS position	Host-side cache	Replaceable FS	Isolated access control
Kernel FS	High	High	Host	✓	✓	✓
User-space FS	Low	Low	Host	∘	✓	∘
Prev.Firm FS	Low	Low	Device	×	×	×
StorStack	Low	Either	Device	✓	✓	✓

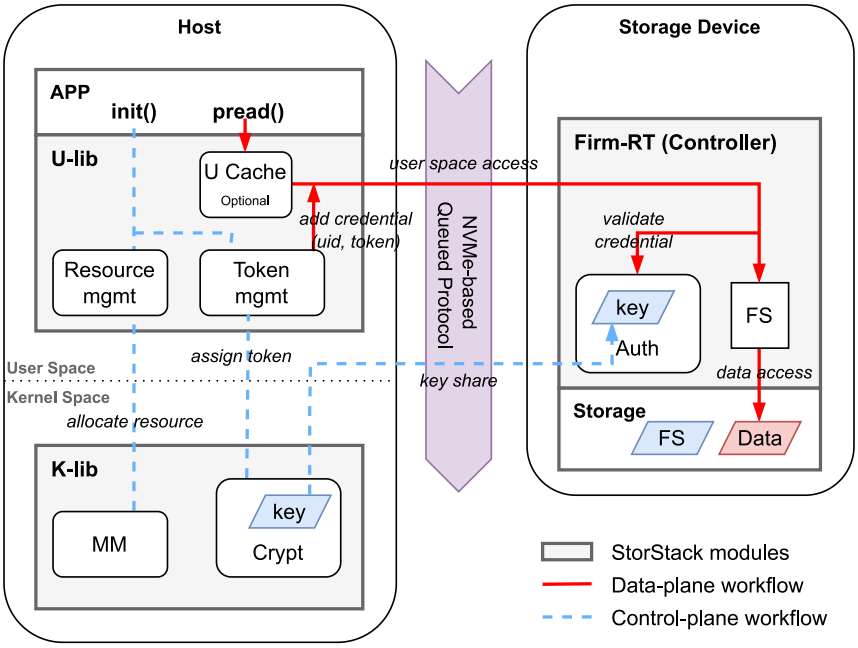

Fig. 1. StorStack Architecture. StorStack consists of three major modules: the `U-lib`, the `K-lib`, and the `Firm-RT`; and there are two workflows: a data-plane workflow, and a control-plane workflow. The interconnection between them is shown in the figure.

3.2.1. High-level design

As shown in Fig. 1, StorStack consists of three major parts: a user lib (`U-lib`), a kernel driver (`K-lib`), and an FS runtime in device firmware (`Firm-RT`).

`U-lib`. The `U-lib` is the interface for user applications to access the in-storage FS, offered as a dynamic link library. The main job of the `U-lib` is to expose POSIX file operations to users, provide user-level cache, and manage the connection with the device. It also cooperates with the `K-lib` and the `Firm-RT` to ensure the reliability of the system.

`K-lib`. The `K-lib` is a kernel module to provide control-plane operations with reliability. Its work includes resource allocation and permission checking. Although it resides in the kernel, the functions of `K-lib` are designed to be rarely called to avoid the performance penalty associated with kernel traps.

`Firm-RT`. The `Firm-RT` is a runtime on the storage firmware that offers essential hardware and software support for in-storage FS to run on the device controller. To serve the FS, `Firm-RT` communicates with both the `U-lib` for data-plane operations, and the `K-lib` for control-plane operations.

3.2.2. StorStack workflow

For clarity, the workflow of StorStack is divided into a data plane and a control plan. The data-plane workflow handles data accesses from user space, and the control plane is responsible for maintaining the system's functionality, safety, and reliability.

For the data plane (red lines in Fig. 1), when a user application calls a file operation in StorStack, the host-side `U-lib` will check the cache if the cache is used. If the cache is bypassed or penetrated, the `U-lib` packs it into an extended NVMe protocol command, and subsequently transmits it to the device-side `Firm-RT`. The `Firm-RT` receives the NVMe command, checks its validity, and then forwards the command to the FS. The FS handles the file operation and then works with the FTL or other hardware instruments to arrange the data blocks on the storage media. The primary distinction between this routine and a typical kernel-based file system lies in the fact that the file system logic is inside the storage device, thus StorStack thereby eliminating the need for kernel traps during data access.

The control plane (blue dashed lines in Fig. 1) provides necessary supports for the data plane to work properly. Control-plane operations on the host side, including memory resource allocation and identity token assignment, are delegated to the kernel to ensure security and reliability. The host-side control-plane operations are designed to be rarely called to reduce kernel trap overhead. On the device, the control plane assists in check the authentication of requests, manage the FS, and deal with other management operations. More detailed security and reliability policies will be described in Section 3.5.

3.2.3. Organization on the storage

In StorStack, file systems are stored in the storage media with pointers originating from partitions, so that the framework can choose the right FS to access a partition. We dedicate a partition to store all the FS binaries that are used by user-created partitions, and each FS in this partition can be indexed by a number. Here we assume that a GUID partition table (GPT) to organize the partitions. Each user-created partition is associated with an FS when it is formatted, and the FS will be added to the FS partition we just mentioned if it was not there yet. To indicate the relation between the user-created partition and its FS, the index number of the FS is added to the attribute flags bits of the partition's GPT entry. The organization is illustrated in Fig. 2. This

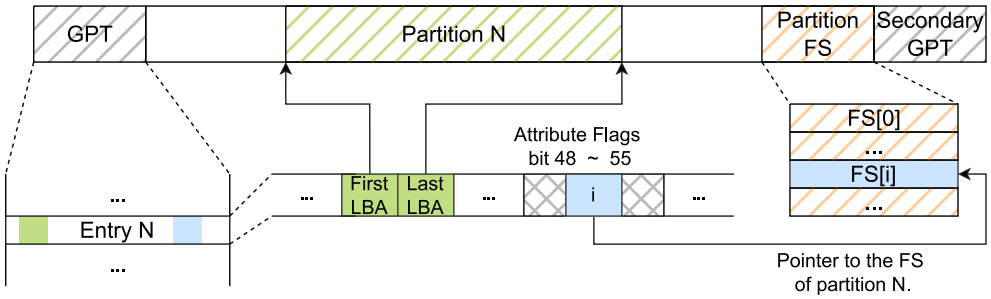

Fig. 2. Partition organization. Figure shows how the FS is stored on the storage and associated with the partition.

design allows StorStack to provide different file systems to different partitions. Meanwhile, the GPT and the partitions are still available for the typical kernel file system routine.

3.3. File access pattern

The `U-lib` provides POSIX IO and AIO interfaces to user applications, and the complicated reliability and performance designs are transparent to users. For regular IO interfaces, the write operations (write, pwrite) act differently with and without cache. When the cache is used, writes will return as soon as an operation passes some simple check and is put into the queue. The interface will not promise that the data is written to the disk before it returns, just like a traditional kernel file system, unless the fsync is called. Without cache, the writes will block the process until the data is written to the storage. The read interfaces (read, pread) will not return until the data is available, regardless of whether there is a cache. The AIO interfaces return immediately when an operation is put into the queue, and the real return value can be fetched by non-blocking check, blocking suspend, or signal.

To make sure that StorStack performs well on high-latency storage devices, an optional user-level per-process cache is provided. Because the reliability of StorStack can only be ensured by the device-side file system but not the `U-lib`, we choose per-process cache to prevent malicious processes from polluting data by writing to a global cache without check. The user-level cache has two ways to deal with write operations: the write-back method returns immediately after the data is put into the cache; the write-around method drops the dirty data in cache and returns after the operation is put into the queue. The write-back cache has a higher performance than the write-around cache, while the write-around cache can provide higher data consistency. In fact, our evaluation shows that the write-back cache in StorStack can outperform the page cache inside the kernel.

3.4. Connectivity

Here we discuss how the host-side `U-lib` and `K-lib` communicate with the device-side `Firm-RT`. StorStack's communication is based on NVMe to take full advantage of high-speed storage devices. We also propose a multi-queue design to improve the performance of device-side FS.

3.4.1. Communication protocol

The communication protocol between the host CPU and StorStack device is a queued protocol extended from NVMe [3]. NVMe is a protocol for accessing non-volatile memories connected via PCIe that supports multiple queues to maximize the throughput, which is suitable for novel high-speed storage devices such as SSDs and SCMs.

To enable the transfer of file operations, we extend the NVMe command list to incorporate the POSIX I/O interface. Meanwhile, the regular data access pattern of NVMe is retained to enable normal disk access when the system does not support StorStack. It is noteworthy that the protocol can be further extended under StorStack to support more paradigms like transactional access [51], log-structured access [52,53], operations fusing [16], or In-storage Computing. We will leave these further explorations to our future work.

With StorStack, heterogeneous hardware like GPUs can implement this extended protocol to access files directly without involving the CPU. For different types of hardware, there are two ways to transmit data. For those who have their own memory (memory-mapped) like GPUs, StorStack can directly place the data to their memory via PCIe bus. For hardware without memory (I/O mapped), StorStack should put the data into the main memory. The manipulation of data destination is directed by the target device driver.

3.4.2. Multi-queue arrangement

NVMe uses multiple queues to improve performance, supporting up to 65,536 I/O queues, with 65,536 commands per queue. Normally, NVMe offers at least a pair of queues (one submission queue and one completion queue) for each core to fully utilize the bandwidth without introducing locks. In StorStack, file operations are processed on the device side, particularly when the storage device features a multi-core controller. To fully utilize the parallelism of the controller cores while minimizing the potential conflicts of concurrent file access, StorStack introduces a special queue organization.

As Fig. 3 shows, every user process in StorStack is assigned a bunch of queue pairs, the number of which is equal to the storage device controller core count. Each queue pair of the queue pair bunch is bound to a controller core of the storage device, so that a process can distribute any file operation to a specific controller core. Meanwhile, each user thread has its exclusive queue pair bunch to avoid queue contention on the host side.

The purpose of this arrangement is to enable the host-side applications to control which operation should be dispatched to which controller core. For example, read intensive applications can issue read operations to all cores with a round robin strategy. For write intensive applications, different threads can send the write operations on the same file to the same controller core to reduce lock contention between controller cores. We will leave the exploration of the scheduling policy for different workloads to future works.

3.5. Security and reliability

From a hardware perspective, the privileged mode (ring 0) that the kernel runs on and the user mode that user applications run on are isolated, which means the access to resources is restricted by hardware. The privileged mode can thus be treated as a trusted area, whereas the user mode as an untrusted area. StorStack introduces the device-side controller as a run-time, which is also isolated from user code and thus viewed as a trusted area.

For safety, everything critical to the correctness of the system should be placed in the trusted area. Typical kernel file systems are placed inside the kernel as they need to manage the data on block devices.

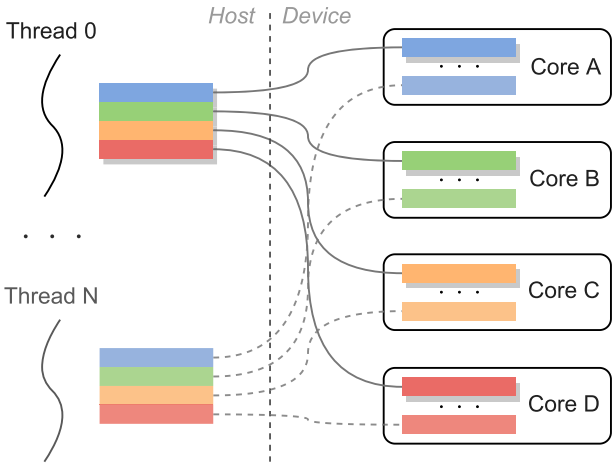

Fig. 3. Queue arrangement and scheduling policies. This figure shows how the queue pairs are mapped between host CPU threads and device controller cores.

StorStack shifts FS to the device side, which is also a trusted area. Meanwhile, as described in Section 3.2.2, StorStack separates the host-side workflow into a control plane and a data plane. The control plane is designed to reside in the host-side trusted area, i.e. the kernel, to cooperate with the device-side FS to ensure security and reliability.

An important design principle of the control plane is to reduce the overhead of the kernel trap. In StorStack, this is done by reducing the proportion of control-plane operations and data-plane operations. There are two types of control-plane workflow on the host side: resource allocation and access control. Both of them are designed to be called rarely.

3.5.1. Resource allocation

The `U-lib` of StorStack is a user-space driver that communicates with the NVMe storage device. It needs to set up VFIO and manage DMA memory mapping to enable direct access from user space. It also needs to allocate areas for caches. These operations involve the kernel but only need to be run once when the device is initialized, so there will not be any performance loss in regular file access.

3.5.2. Permission checking

To provide access control, file systems must check the user's permission to make sure that a file operation is legal. In kernel file systems, the file system can use the process structure in the kernel to validate the process's identity, and then compare it with the permission information stored in the file's inode. In StorStack, however, the file system resides on the device rather than in the kernel, so the kernel needs to share the process's information with the device to support permission checking.

To avoid entering the kernel frequently, DevFS [14] maintains a table that maps CPU IDs to process credentials in the device. All requests are tagged with the CPU's ID that the process runs on before they are sent to the device. The kernel is modified to update the table whenever a process is scheduled on a host CPU. There are two problems with this mechanism. Firstly, it assumes that the CPU ID is unforgeable, but usually a malicious process can potentially exploit the ID of another CPU to escalate its privilege. Secondly, this requires a modification to the process scheduler, which is a core module of the kernel, so making it incompatible with standard OS kernels, and may slow down the system.

In StorStack, we propose a new method to share the credential of the process, with less communication, safer guarantee, and no change to the Linux kernel. The process is shown in Fig. 4. When the `U-lib` is initialized on a process, it calls the `K-lib` (a kernel driver) via `ioctl()` (system call) to get a credential token. The `K-lib` generates a secret key if one has not been set yet, then save and copy it to the device by kernel NVMe driver. Once the key is set, `K-lib` uses it to encrypt the process's credential information (i.e. uid) into MAC (Message Authentication Code). The resulting token, which is the output of the encryption, is then returned to the process. Since the secret key is stored in the kernel, the process cannot forge a token but can only use the one assigned by the kernel, which can prove the authenticity of the uid claimed by the process. Before being sent to the device, every request from the process is tagged with the process's uid and the token, so that the device can use the secret key and the token to verify the uid and check the identity of the request. This mechanism requires only one communication between the kernel and the device to share the secret key, and one kernel trap to initialize the token for each process. Also, the `K-lib` is implemented as a kernel driver, without any modification to the core functions of the kernel, which makes it compatible with conventional operating system.

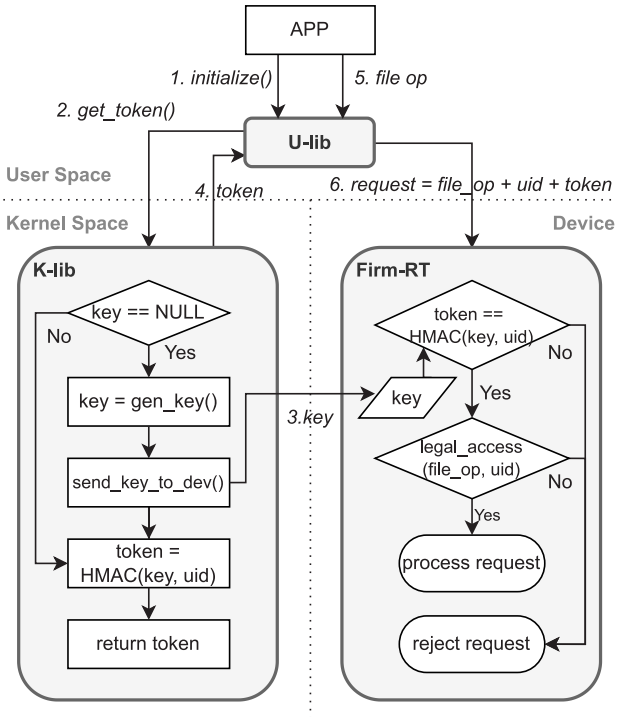

Fig. 4. Permission checking. Figure shows how the user space, the kernel space, and the device work together to check the validity of a request without frequent kernel traps.

3.5.3. Device lock

StorStack is designed to support direct I/O not only from CPUs, but also from different types of heterogeneous computing devices. To prevent concurrent access to the same file from multiple devices, a concurrency control method is required. A common practice is to implement a distributed lock across all devices, but this can be too costly for low-level hardware. In StorStack, we provide in-storage file-level locking mechanisms to protect the files from unexpected access by multiple devices.

StorStack supports two types of lock: (1) spinning lock, an error code will be returned to the caller if the file it accesses is already locked by another device, allowing the caller to continue attempting to acquire the lock until the file is unlocked; (2) sleeping lock, where if the file is locked, any requests from other devices to that file will wait in the submission queue until the file is unlocked. From the perspective of concurrency, StorStack supports both shared lock and exclusive lock, which act exactly the same as those on other systems.

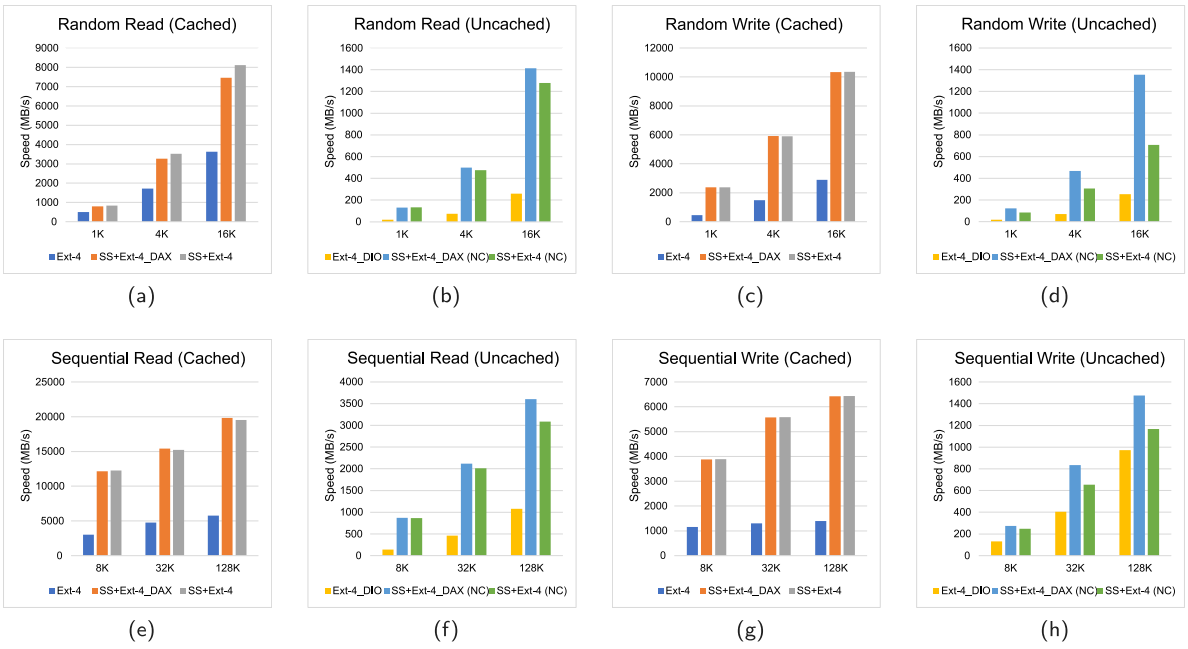

Fig. 5. Random and sequential r/w. Figure shows the basic performance of StorStack compared with Ext-4, under different cache, block size, and in-storage file system settings.

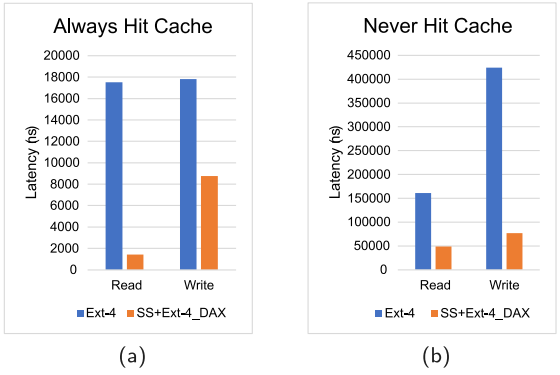

Fig. 6. Time cost for a single operation.

3.6. Implementation

We have implemented a prototype of StorStack, which consists of three parts: the `U-lib`, the `K-lib` and the `Firm-RT`. The source code of this prototype is available at https://anonymous.4open.science/r/StorStack-524F/.

The `U-lib` is implemented under Linux 5.15, utilizing SPDK [54] to access storage devices from user space. The SPDK library is modified in StorStack to transfer POSIX I/O operations over NVMe. The `U-lib` comprises two major components: a dynamic link library that provides interfaces and a user-level cache for accessing the device, and a daemon program responsible for managing the connection to the device.

The `K-lib` is implemented as a simple kernel module in Linux 5.15 kernel. It only takes charge of two things: creating the secret key when the StorStack is initialized so that the `K-lib` and the `Firm-RT` can use it to encrypt and decrypt the MAC token for processes' credentials; generating the MAC token from the uid of the current process with HMAC algorithm when the process initializes, and then return it to the `U-lib`. The interface of the `K-lib` is exposed to the user space through `ioctl`.

The `Firm-RT` is the only component that located on the device side. In this work, the `Firm-RT` is not implemented on actual storage hardware but is instead simulated using QEMU and the system running on its host machine. There are two reasons for the simulation: first, although there are several works regarding programmable storage controllers [49,55–57], these solutions are either expensive or lack high-level programmability as most of them are based on FPGA; second, by simulating with various latency settings, we can evaluate the performance of StorStack on different types of storage devices, which can be costly if done with real hardware. In our prototype, QEMU has been modified to handle extended NVMe POSIX I/O operations and check the token of each operation.

4. Evaluation

In this section, we evaluate the performance of StorStack and compare it with popular file systems to answer the following questions:

- Is StorStack efficient enough compared to widely used kernel file systems?
- How much performance is gained from the kernel trap avoidance?
- How does StorStack perform on different types of devices?
- How is the concurrency performance of StorStack?

4.1. Experimental setup

Our experiment platform is a 20-core 2.4 GHz Intel Xeon server equipped with 64 GB DDR4 memory and 512 GB SSD. Among them, 8 cores with 16 GB memory are assigned to the QEMU VM to simulate the StorStack host; other cores with 16 GB memory are reserved to emulate the StorStack device. Both the StorStack host and the StorStack device runs on Linux 5.15.

StorStack's expected settings on the device require only a minimal embedded system with abstractions of hardware functions and necessary libraries, but due to our simulation requirements, we choose Linux as the device-side environment to support the execution of QEMU.

In this section, we evaluate the performance of StorStack using Filebench [58], a widely used benchmarking suite for testing file system performance. We access StorStack under various configurations, including different cache options, device access latency, thread numbers and read/write ratios, to address the four questions previously raised.

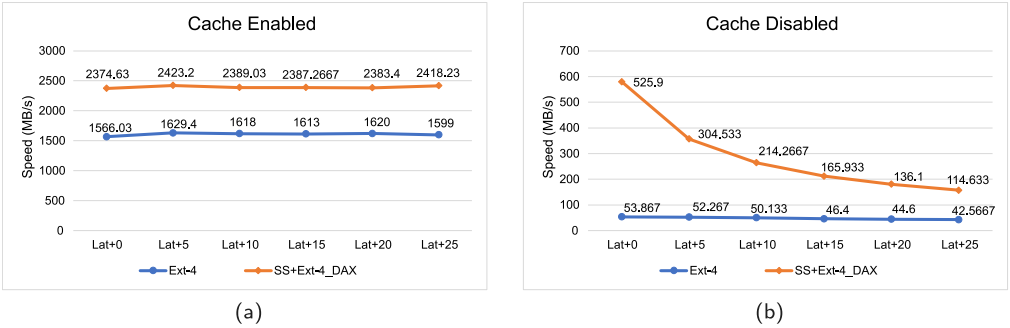

Fig. 7. Performance with simulated latency. This figure shows the change in throughput as a function of simulated device access latency.

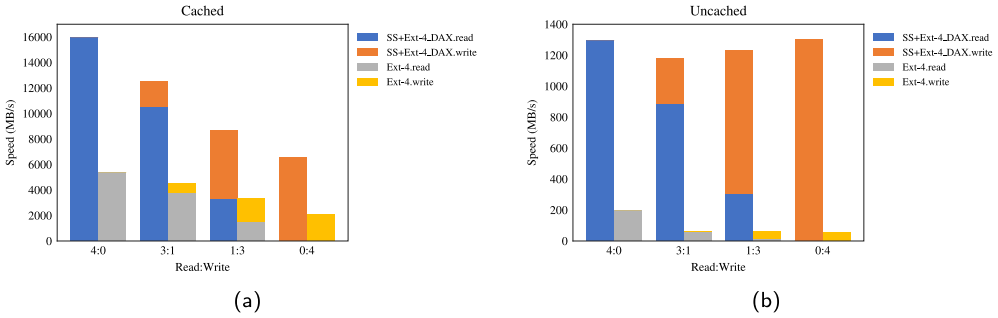

Fig. 8. Multi-thread Performance.

4.2. Random and sequential r/w

First, we evaluate StorStack's performance with single-thread random and sequential read/write tests. The random tests run on a 1 GB file with 1K, 4K, and 16K bytes I/O size. The sequential tests run on a 8 GB file with 8K, 32K, and 128K bytes I/O size. Both of the files are stored on the DRAM memory, which is simulated as a PMEM by `memmap`. The tests are performed on StorStack (referred to as SS) with two different in-storage FS settings: `SS+Ext-4` and `SS+Ext-4_DAX`. Then we compare them with `Ext-4`. We also evaluate the performance of SS without cache (`SS NC`) and Ext-4 with direct IO (`Ext-4_DIO`) to study performance improvement when accessed directly.

Fig. 5 shows the results of the random and sequential tests. In both tests, SS outperforms traditional kernel-level `Ext-4`, due to our kernel-bypass and near-data file system design. `SS+Ext-4_DAX` with user-level write-back cache achieves averagely 1.98x, 4.25x, 3.59x, and 4.08x performance gain on random read, random write, sequential read, and sequential write respectively compared with `Ext-4` with page cache. For direct access, the speed increase is 6.41x, 6.21x, 4.72x, and 1.90x respectively. Another interesting phenomenon is that in cached StorStack, the performances of `SS+Ext-4` and `SS+Ext-4_DAX` are similar, indicating that the choice of the in-storage file system does not matter because most operations are handled by the user-level cache. However, in uncached tests, `SS+Ext-4_DAX` show better results, which means that the in-storage file system may influence the overall performance in direct access.

4.3. Profit of kernel bypassing

We measure the time cost of a single operation to study the profit of kernel bypassing. The cached test demonstrates the impact of kernel trap on the access of in-memory page cache. The uncached test shows the impact of both kernel trap and write amplification on direct access to the storage device. Both tests utilize 4KB block size, and the files are stored on the simulated PMEM. The results in Fig. 6 indicate that compared to `Ext-4`, `SS+Ext-4_DAX` reduces latency by 91.91%, 50.46%, 69.83%, and 81.83% on cached read, cached write, uncached read, and uncached write.

When the cache hits, the data resides in fast DRAM, resulting in low data-fetch latency. In this scenario, traditional Ext-4 exhibits higher access latency, as the kernel trap accounts for most of the latency. In contrast, StorStack shows lower latency because its cache is implemented inside user space eliminating the need for kernel traps. When a cache miss occurs, the primary overhead shifts to the multiple rounds of storage device access, which further increases the performance gap between traditional Ext-4 and StorStack.

4.4. Impact of access latency

Storage devices with different access latencies may influence the performance of file systems. In this experiment, we use multiple latency settings to simulate devices with different access speed. The latency is simulated on the device side by QEMU.

We compare the performance of SS with `Ext-4` under cached and uncached settings using several latency settings. The latency ranges from 0 μs to 25 μs to simulate connection methods from DDR to PCIe to RDMA. Tests run with 4KB block size.

Fig. 7 shows the result of this test. With a cache, both SS and `Ext-4` are not susceptible to the rise of latency. However, without cache, the performance of SS has a 78.20% degrade from 526MB/s at 0 simulated latency to 115 MB/s at 25 μs latency. The performance of `Ext-4` also cuts 20.98% from 54MB/s to 43MB/s. Note that the experiment introduces extra latency due to QEMU, so the simulated 0 latency is larger than 0 actually, meaning that the curve can even go higher on the left side of the graph. The result illustrates that direct access of SS should only be enabled on ultra-low latency devices. For other hardware, it is better to enable the cache.

4.5. Multi-thread performance

To study the performance of StorStack under multiple threads, we evaluate SS and `Ext-4` under a multi-thread micro-benchmark. The benchmark is to perform parallel 4KB file operations on one file with 4 threads, each thread is a reader or a writer, and the ratio of readers and

writers is set to 4:0, 3:1, 1:3, and 0:4. Fig. 8 shows the result. StorStack is faster than Ext-4 in all concurrent read and write scenarios of our test. For cached scenario, SS is on average 2.88x faster than Ext−4 in all read-write ratios. For uncached scenario, the speed up is 17.34x.

5. Conclusion

In this paper, we present StorStack, a full-stack design for in-storage file systems framework and simulator. The StorStack components across user space, kernel space, and device space collaborate to enable file systems to run inside the storage device efficiently and reliably. We implement a prototype of StorStack and evaluate it with various settings. Experimental results show that StorStack outperforms current kernel file systems in both cached and uncached scenes. Some further performance optimizations, such as the combination of file systems and storage hardware capabilities, the exploration of multi-queue scheduling strategies for different workloads, and the performance of direct access from heterogeneous devices, are left to future work.

CRediT authorship contribution statement

Juncheng Hu: Writing – review & editing, Writing – original draft. **Shuo Chen:** Formal analysis, Data curation. **Haoyang Wei:** Formal analysis, Data curation. **Guoyu Wang:** Writing – review & editing, Writing – original draft. **Chenju Pei:** Formal analysis, Data curation. **Xilong Che:** Methodology, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was funded by the National Key Research and Development Programme No. 2024YFB3310200, and by Key scientific and technological R&D Plan of Jilin Province of China under Grant No. 20230201066GX, and by the Central University Basic Scientific Research Fund Grant No.2023-JCXK-04.

References

[1] G. Koo, K.K. Matam, T. I, H.K.G. Narra, J. Li, H.-W. Tseng, S. Swanson, M. Annavaram, Summarizer: trading communication with computing near storage, in: Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, 2017, pp. 219–231.

[2] S.S.M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, S. Swanson, Willow: A User-Programmable ssd, OSDI, 2014.

[3] NVMe specifications, https://nvmexpress.org/specifications/.

[4] Intel, Intel® Optane™ Persistent Memory, https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html.

[5] S. Mittal, J.S. Vetter, A survey of software techniques for using non-volatile memories for storage and main memory systems, IEEE Trans. Parallel Distrib. Syst. 27 (5) (2016) 1537–1550, http://dx.doi.org/10.1109/TPDS.2015.2442980.

[6] M. Wei, M. Bjørling, P. Bonnet, S. Swanson, I/O speculation for the microsecond era, in: 2014 USENIX Annual Technical Conference, USENIX ATC 14, 2014, pp. 475–481.

[7] S. Peter, J. Li, I. Zhang, D.R.K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, T. Roscoe, Arrakis: the operating system is the control plane, in: 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI 14, 2014, pp. 1–16.

[8] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, M.M. Swift, Aerie: Flexible file-system interfaces to storage-class memory, in: Proceedings of the Ninth European Conference on Computer Systems, in: EuroSys '14, Association for Computing Machinery, New York, NY, USA, 2014, pp. 1–14, http://dx.doi.org/10.1145/2592798.2592810.

[9] A.M. Caulfield, T.I. Mollov, L.A. Eisner, A. De, J. Coburn, S. Swanson, Providing safe, user space access to fast, solid state disks, ACM SIGPLAN Not. 47 (4) (2012) 387–400, http://dx.doi.org/10.1145/2248487.2151017.

[10] M. Dong, H. Bu, J. Yi, B. Dong, H. Chen, Performance and protection in the ZoFS user-space NVM file system, in: Proceedings of the 27th ACM Symposium on Operating Systems Principles, ACM, Huntsville Ontario Canada, 2019, pp. 478–493, http://dx.doi.org/10.1145/3341301.3359637.

[11] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, T. Anderson, Strata: A cross media file system, in: Proceedings of the 26th Symposium on Operating Systems Principles, in: SOSP '17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 460–477, http://dx.doi.org/10.1145/3132747.3132770.

[12] J. Liu, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, S. Kannan, File systems as processes, in: 11th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 19, USENIX Association, Renton, WA, 2019.

[13] S. Zhong, C. Ye, G. Hu, S. Qu, A. Arpaci-Dusseau, R. Arpaci-Dusseau, M. Swift, MadFS: per-file virtualization for userspace persistent memory filesystems, in: 21st USENIX Conference on File and Storage Technologies, FAST 23, 2023, pp. 265–280.

[14] S. Kannan, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, Y. Wang, J. Xu, G. Palani, Designing a true direct-access file system with devfs, in: 16th USENIX Conference on File and Storage Technologies, FAST 18, USENIX Association, Oakland, CA, 2018, pp. 241–256.

[15] Y. Ren, C. Min, S. Kannan, Crossfs: A cross-layered direct-access file system, in: 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 20, USENIX Association, 2020, pp. 137–154.

[16] J. Zhang, Y. Ren, S. Kannan, FusionFS: fusing I/O operations using ciscops in firmware file systems, in: 20th USENIX Conference on File and Storage Technologies, FAST 22, USENIX Association, Santa Clara, CA, 2022, pp. 297–312.

[17] N. Agrawal, V. Prabhakaran, T. Wobber, J.D. Davis, M. Manasse, R. Panigrahy, Design tradeoffs for SSD performance, in: USENIX 2008 Annual Technical Conference, in: ATC'08, USENIX Association, USA, 2008, pp. 57–70.

[18] F. Chen, D.A. Koufaty, X. Zhang, Understanding intrinsic characteristics and system implications of flash memory based solid state drives, in: Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems, in: SIGMETRICS '09, Association for Computing Machinery, New York, NY, USA, 2009, pp. 181–192, http://dx.doi.org/10.1145/1555349.1555371.

[19] Welcome to PCI-SIG | PCI-SIG, https://pcisig.com/.

[20] Y. Choi, I. Song, M.-H. Park, H. Chung, S. Chang, B. Cho, J. Kim, Y. Oh, D. Kwon, J. Sunwoo, J. Shin, Y. Rho, C. Lee, M.G. Kang, J. Lee, Y. Kwon, S. Kim, J. Kim, Y.-J. Lee, Q. Wang, S. Cha, S. Ahn, H. Horii, J. Lee, K. Kim, H. Joo, K. Lee, Y.-T. Lee, J. Yoo, G. Jeong, A 20nm 1.8V 8gb PRAM with 40mb/s program bandwidth, in: 2012 IEEE International Solid-State Circuits Conference, 2012, pp. 46–48, http://dx.doi.org/10.1109/ISSCC.2012.6176872.

[21] H. Volos, A.J. Tack, M.M. Swift, Mnemosyne: Lightweight persistent memory, ACM SIGARCH Comput. Archit. News 39 (1) (2011) 91–104, http://dx.doi.org/10.1145/1961295.1950379.

[22] S.-W. Chung, T. Kishi, J.W. Park, M. Yoshikawa, K.S. Park, T. Nagase, K. Sunouchi, H. Kanaya, G.C. Kim, K. Noma, M.S. Lee, A. Yamamoto, K.M. Rho, K. Tsuchida, S.J. Chung, J.Y. Yi, H.S. Kim, Y. Chun, H. Oyamatsu, S.J. Hong, 4Gbit density STT-MRAM using perpendicular MTJ realized with compact cell structure, in: 2016 IEEE International Electron Devices Meeting, IEDM, 2016, pp. 27.1.1–27.1.4, http://dx.doi.org/10.1109/IEDM.2016.7838490.

[23] H. Akinaga, H. Shima, Resistive random access memory (ReRAM) based on metal oxides, Proc. IEEE 98 (12) (2010) 2237–2251, http://dx.doi.org/10.1109/JPROC.2010.2070830.

[24] K. Kawai, A. Kawahara, R. Yasuhara, S. Muraoka, Z. Wei, R. Azuma, K. Tanabe, K. Shimakawa, Highly-reliable TaOx reram technology using automatic forming circuit, in: 2014 IEEE International Conference on IC Design & Technology, 2014, pp. 1–4, http://dx.doi.org/10.1109/ICICDT.2014.6838600.

[25] K. Suzuki, S. Swanson, The Non-Volatile Memory Technology Database (NVMDB), Tech. Rep. CS2015-1011, Department of Computer Science & Engineering, University of California, San Diego, 2015.

[26] S. Matsuura, Designing a persistent-memory-native storage engine for SQL database systems, in: 2021 IEEE 10th Non-Volatile Memory Systems and Applications Symposium, NVMSA, IEEE, Beijing, China, 2021, pp. 1–6, http://dx.doi.org/10.1109/NVMSA53655.2021.9628842.

[27] R. Tadakamadla, M. Patocka, T. Kani, S.J. Norton, Accelerating database workloads with DM-WriteCache and persistent memory, in: Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, in: ICPE '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 255–263, http://dx.doi.org/10.1145/3297663.3309669.

[28] W. Wang, C. Yang, R. Zhang, S. Nie, X. Chen, D. Liu, Themis: malicious wear detection and defense for persistent memory file systems, in: 2020 IEEE 26th International Conference on Parallel and Distributed Systems, ICPADS, 2020, pp. 140–147, http://dx.doi.org/10.1109/ICPADS51040.2020.00028.

[29] B. Zhu, Y. Chen, Q. Wang, Y. Lu, J. Shu, Octopus+ : An RDMA-enabled distributed persistent memory file system, ACM Trans. Storage 17 (3) (2021) 1–25, http://dx.doi.org/10.1145/3448418.

[30] J. Do, V.C. Ferreira, H. Bobarshad, M. Torabzadehkashi, S. Rezaei, A. Heydarigorji, D. Souza, B.F. Goldstein, L. Santiago, M.S. Kim, P.M.V. Lima, F.M.G. França, V. Alves, Cost-effective, energy-efficient, and scalable storage computing for large-scale AI applications, ACM Trans. Storage 16 (4) (2020) 21:1–21:37, http://dx.doi.org/10.1145/3415580.

[31] L. Kang, Y. Xue, W. Jia, X. Wang, J. Kim, C. Youn, M.J. Kang, H.J. Lim, B. Jacob, J. Huang, IceClave: A trusted execution environment for in-storage computing, in: MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture, in: MICRO '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 199–211, http://dx.doi.org/10.1145/3466752.3480109.
[32] Z. Ruan, T. He, J. Cong, INSIDER: designing in-storage computing system for emerging high-performance drive, in: 2019 USENIX Annual Technical Conference, USENIX ATC 19, USENIX Association, Renton, WA, 2019, pp. 379–394.
[33] A.M. Caulfield, T.I. Mollov, L.A. Eisner, A. De, J. Coburn, S. Swanson, Providing safe, user space access to fast, solid state disks, ACM SIGPLAN Not. 47 (4) (2012) 387–400.
[34] S. Cho, C. Park, H. Oh, S. Kim, Y. Yi, G.R. Ganger, Active disk meets flash: A case for intelligent ssds, in: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, 2013, pp. 91–102.
[35] J. Do, Y.-S. Kee, J.M. Patel, C. Park, K. Park, D.J. DeWitt, Query processing on smart ssds: Opportunities and challenges, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, 2013, pp. 1221–1230.
[36] C. Cowan, S. Beattie, J. Johansen, P. Wagle, {PointGuard.}: Protecting pointers from buffer overflow vulnerabilities, in: 12th USENIX Security Symposium, USENIX Security 03, 2003.
[37] L. Szekeres, M. Payer, T. Wei, D. Song, Sok: Eternal war in memory, in: 2013 IEEE Symposium on Security and Privacy, IEEE, 2013, pp. 48–62.
[38] S.R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, J. Jackson, System software for persistent memory, in: Proceedings of the Ninth European Conference on Computer Systems - EuroSys '14, ACM Press, Amsterdam, The Netherlands, 2014, pp. 1–15, http://dx.doi.org/10.1145/2592798.2592814.
[39] C. Lee, D. Sim, J. Hwang, S. Cho, F2FS: A new file system for flash storage, in: 13th USENIX Conference on File and Storage Technologies, FAST 15, USENIX Association, Santa Clara, CA, 2015, pp. 273–286.
[40] DAX, https://www.kernel.org/doc/Documentation/filesystems/dax.txt.
[41] J. Xu, S. Swanson, NOVA: A log-structured file system for hybrid volatile/non-volatile main memories, in: Proceedings of the 14th Usenix Conference on File and Storage Technologies, in: FAST'16, USENIX Association, USA, 2016, pp. 323–338.
[42] M. Torabzadehkashi, S. Rezaei, A. HeydariGorji, H. Bobarshad, V. Alves, N. Bagherzadeh, Computational storage: An efficient and scalable platform for big data and HPC applications, J. Big Data 6 (1) (2019) 100, http://dx.doi.org/10.1186/s40537-019-0265-5.
[43] W. Cao, Y. Liu, Z. Cheng, N. Zheng, W. Li, W. Wu, L. Ouyang, P. Wang, Y. Wang, R. Kuan, Z. Liu, F. Zhu, T. Zhang, POLARDB meets computational storage: efficiently support analytical workloads in cloud-native relational database, in: Proceedings of the 18th USENIX Conference on File and Storage Technologies, in: FAST'20, USENIX Association, USA, 2020, pp. 29–42.
[44] Nvidia, NVIDIA RTX IO: GPU accelerated storage technology, https://www.nvidia.com/en-us/geforce/news/rtx-io-gpu-accelerated-storage-technology/.
[45] AMD, Radeon™ Pro SSG graphics, https://www.amd.com/en/products/professional-graphics/radeon-pro-ssg.
[46] Z. An, Z. Zhang, Q. Li, J. Xing, H. Du, Z. Wang, Z. Huo, J. Ma, Optimizing the datapath for key-value middleware with NVMe SSDs over RDMA interconnects, in: 2017 IEEE International Conference on Cluster Computing, CLUSTER, 2017, pp. 582–586, http://dx.doi.org/10.1109/CLUSTER.2017.69.
[47] Samsung, Samsung 990 PRO with heatsink, https://semiconductor.samsung.com/content/semiconductor/global/consumer-storage/internal-ssd/990-pro-with-heatsink.html.
[48] A. Ltd, ARM computational storage solution, https://www.arm.com/solutions/storage/computational-storage.
[49] Samsung, Samsung SmartSSD, https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html.
[50] ScaleFlux, ScaleFlux, https://scaleflux.com/.
[51] E. Gal, S. Toledo, A transactional flash file system for microcontrollers, in: 2005 USENIX Annual Technical Conference, USENIX ATC 05, 2005.
[52] J. Koo, J. Im, J. Song, J. Park, E. Lee, B.S. Kim, S. Lee, Modernizing file system through in-storage indexing, in: Proceedings of the 15th Usenix Symposium on Operating Systems Design and Implementation, Osdi '21, Usenix Assoc, Berkeley, 2021, pp. 75–92, http://dx.doi.org/10.5281/zenodo.4659803.
[53] LevelDB, https://github.com/google/leveldb.
[54] Storage performance development kit, https://spdk.io/.
[55] DFC open source, https://github.com/DFC-OpenSource.
[56] M. Jung, OpenExpress: fully hardware automated open research framework for future fast NVMe devices, in: 2020 USENIX Annual Technical Conference, USENIX ATC 20, 2020, pp. 649–656.
[57] J. Kwak, S. Lee, K. Park, J. Jeong, Y.H. Song, Cosmos+ OpenSSD: rapid prototype for flash storage systems, ACM Trans. Storage 16 (3) (2020) 15:1–15:35, http://dx.doi.org/10.1145/3385073.
[58] Filebench, https://github.com/filebench/filebench.

Juncheng Hu received the bachelor's degree and doctor of Engineering degree from Jilin University in 2017 and 2022, where he is current a lecturer in Jilin University. His research interests include data mining, machine learning, computer network and parallel computing.
jchu@jlu.edu.cn

Shuo Chen is currently working toward the master's degree with College of Computer Science and Technology, Jilin University since 2022. His research field is computer architecture, mainly focusing on optimization for caching systems.
chenshuo22@mails.jlu.edu.cn

Wei Haoyang a 23rd-year Master's student in Computer Science and Technology at Jilin University, focuses on computer architecture research, with a primary interest in the application of new storage devices.
hywei23@mails.jlu.edu.cn

Guoyu Wang is currently working toward the doctor's degree with College of Computer Science and Technology, Jilin University.
wgy21@mails.jlu.edu.cn

Pei Chenju is an undergraduate student at the School of Computer Science and Technology at Jilin University. His field of research is computer system architecture, and he is currently investigating new L7 load balancing solutions.
peicj2121@mails.jlu.edu.cn

Che Xilong Received the M.S. and Ph.D. degrees in Computer Science from Jilin University, in 2006 and 2009 respectively.

Currently, He is a full professor and doctoral supervisor at the College of Computer Science and Technology, Jilin University, China.

His current research areas are Parallel & Distributed Computing, High Performance Computing Architectures, and related optimizations.

He is a member of the China Computer Federation. Corresponding author of this paper.
chexilong@jlu.edu.cn