Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Review:

A survey on design and application of open-channel solid-state drives*

Junchao CHEN[1,2,3], Guangyan ZHANG†‡[1,3], Junyu WEI[1,3]

[1]*Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China*

[2]*Xi'an Satellite Control Center, Xi'an 710043, China*

[3]*Beijing National Research Center for Information Science and Technology, Tsinghua University, Beijing 100084, China*

†E-mail: gyzh@tsinghua.edu.cn

Received July 25, 2022; Revision accepted Nov. 16, 2022; Crosschecked Mar. 3, 2023

Abstract: Compared with traditional solid-state drives (SSDs), open-channel SSDs (OCSSDs) expose their internal physical layout and provide a host-based flash translation layer (FTL) that allows host-side software to control the internal operations such as garbage collection (GC) and input/output (I/O) scheduling. In this paper, we comprehensively survey research works built on OCSSDs in recent years. We show how they leverage the features of OCSSDs to achieve high throughput, low latency, long lifetime, strong performance isolation, and high resource utilization. We categorize these efforts into five groups based on their optimization methods: adaptive interface customizing, rich FTL co-designing, internal parallelism exploiting, rational I/O scheduling, and efficient GC processing. We discuss the strengths and weaknesses of these efforts and find that almost all these efforts face a dilemma between performance effectiveness and management complexity. We hope that this survey can provide fundamental knowledge to researchers who want to enter this field and further inspire new ideas for the development of OCSSDs.

Key words: Domain-specific storage; Flash translation layer; Garbage collection; Internal parallelism; Open-channel solid-state drives (OCSSDs)

https://doi.org/10.1631/FITEE.2200317 **CLC number:** TP302

1 Introduction

In recent years, solid-state drives (SSDs) have been widely used in data centers (Bjørling et al., 2017), cloud storage systems (Zhang XY et al., 2021), and mobile devices (Lee C et al., 2015) due to their better performance such as higher input/output (I/O) throughput, lower access latency, and lower power consumption than hard-disk drives (HDDs).

However, the design principle of traditional file systems targets the physical media of HDDs and does not consider the features of flash memory, which leads to inefficient utilization of SSDs. Even worse, frequent random writes and flush operations to an SSD can seriously aggravate the wearing, cause internal fragmentation, and reduce its lifetime (Lee C et al., 2015). Storing data on an aged SSD may trigger garbage collection (GC) operations frequently, which will cause a bad quality of service (QoS).

Some traditional file systems are designed to overcome the challenges of frequent random writes and flush operations. For example, the log-structured file system (LFS) (Rosenblum and Ousterhout, 1991) writes data sequentially to disks in a log-structured manner, which converts small synchronous random writes into large asynchronous sequential writes, while designed for HDDs, it can eliminate the negative effect of frequent small writes to SSDs and significantly improve the write

‡ Corresponding author

* Project supported by the National Natural Science Foundation of China (No. 62025203)

ORCID: Junchao CHEN, https://orcid.org/0000-0002-6273-075X; Guangyan ZHANG, https://orcid.org/0000-0002-3480-5902

throughput. The copy on write (COW) strategy of BTRFS (Rodeh et al., 2013) and the delayed allocation mechanism of Ext4 (Mathur et al., 2007) are able to improve the resistance to fragmentation and thus can mitigate the aging problem of flash media. However, these works do not consider the physical features of SSDs and fail to exploit their strengths.

Some research works try to design tailored file systems or optimize specific methods to exploit the potential of SSDs. For example, F2FS (Lee C et al., 2015) is a Linux kernel file system designed for flash storage devices; it uses a flash-friendly data layout and an effective GC mechanism to enhance flash storage performance and prolong its lifetime. Park et al. (2010) proposed external request rescheduling and dynamic write mapping approaches to exploit SSD's internal plane- and die-level parallelism. Yang LH et al. (2019) observed the external behaviors of SSDs and exploited their internal parallelism for specific applications. However, these works still do not fully explore the performance advantage of SSDs mainly because they treat an SSD as a black box, and instead of directly building upon raw flash memory, they are based on the block interface to the built-in flash translation layer (FTL). This results in a huge semantic gap between file systems and flash devices, and additionally causes the inability to fully leverage the parallelism and scheduling mechanism inside SSDs.

Therefore, more researchers have shifted their attention to the key ideas of treating an SSD as a white box, namely, moving the FTL from the device side to the host side and letting the external software directly interact with raw flash memory. These ideas give birth to open-channel SSDs (OCSSDs). An OCSSD exposes its internal physical layout and moves the FTL to the host side, allowing the FTL to be controlled and co-designed by the host side. OCSSD's openness concept allows the host side to fully exploit its internal parallelism and design optimization methods, such as rational I/O scheduling policies and efficient GC operations, to achieve high performance, such as high I/O throughput, stable performance, low tail latency, and high resource utilization.

This paper presents a review of nearly 10 years of exploring the advantages of OCSSDs in academia and industry; the representative papers we surveyed are shown in Table 1. These works are roughly divided into three time periods: works before 2015 initially explore the open-channel advantages of SSDs and gradually form and perfect the concept of OCSSDs; works between 2016 and 2018 focus mainly on the systematic design of OCSSDs and form some relatively mature file systems; works after 2018 mainly try to address domain-specific problems based on LightNVM or simulation platforms, and meanwhile optimize them for better performance.

Table 1 Representative works in 2013–2022

Year	Work
2013	OFTL (Lu et al., 2013)
2014	SDF (Ouyang et al., 2014)
	LOCS (Wang P et al., 2014)
	SOFA (Chiueh et al., 2014)
2015	NVMKV (Marmol et al., 2015)
2016	ParaFS (Zhang JC et al., 2016)
	AMF (Lee S et al., 2016)
2017	FlashKV (Zhang JC et al., 2017)
	FlashBlox (Huang et al., 2017)
	Multi-Partition (González and Bjørling, 2017)
	LightNVM (Bjørling et al., 2017)
	uFLIP-OC (Picoli et al., 2017)
	TTFLASH (Yan et al., 2017)
	DIDACache (Shen et al., 2017)
2018	OC-Cache (Wang HT et al., 2018)
2019	ThermAlloc (Wang Y et al., 2019)
	PATCH (Chen et al., 2019)
	StageFS (Lu et al., 2019a)
	OCStore (Lu et al., 2019b)
	hWA-BC (Lee S et al., 2019)
	QBLK (Qin et al., 2019)
2020	SSW (Du et al., 2020)
	OX-FTL (Picoli et al., 2020)
2021	NBFS (Qin et al., 2021a)
	QBLKe (Qin et al., 2021b)
	AOCBLK (Zhang XY et al., 2021)
	MT-Cache (Oh and Ahn, 2021)
	IODA (Li et al., 2021)
	oNFC (Qiu et al., 2021)
	JS-Pblk (Son and Ahn, 2021)
2022	Prism-SSD (Shen et al., 2022)
	PVSensing (Wang Y et al., 2022)

2 Overview of open-channel solid-state drives

SSD is a type of persistent storage mainly with NAND flash as its media. Different from HDDs, SSDs are made up of pure electronic circuits and store data in semiconductors, avoiding the bottleneck of physical mechanical movement; they can reach a better performance under the automated management of embedded FTL such as GC, address

mapping, and error handling (Fig. 1a).

OCSSDs have evolved from SSDs (SSD refers to traditional SSD in the remainder of this paper unless explicitly stated otherwise) in recent years. Unlike SSDs, they move the FTL to the host side and expose the internal physical features of SSDs. To be concrete, as shown in Fig. 1b, OCSSDs move application-related functions such as GC and I/O scheduling to the host-side FTL, and in most cases leave the device-closely-related functions, such as wear leveling (WL) and error correcting code (ECC), in a simple FTL at the device side. By doing so, they expose the physical page addresses (PPAs) of the space-arranged NAND storage units rather than logical block addresses (LBAs) to the host-side software for better hardware access. OCSSDs enable customizing the FTL for specific scenarios and inherit the advantages of SSDs while having the potential to avoid SSDs' shortcomings. A typical architecture of OCSSDs is shown in Fig. 2.

2.1 Physical layout

The physical layout of the storage media of a typical OCSSD is organized as a hierarchy of channel, chip, die, plane, block, and page. This physical layout brings an opportunity to exploit the internal parallelism of OCSSDs. Existing works have explored the parallelism from four levels (Hu et al., 2013): channel level (e.g., Lu et al., 2019b), chip level (e.g., Wang Y et al., 2019), die level (e.g., Bjørling et al., 2017), and plane level (e.g., Ouyang et al., 2014). From the software perspective, the open-channel specification (The Open-Channel SSD Community, 2023) suggests exposing three levels on channels, parallel units (PUs), and planes, which integrates the chip level and die level as the PU level.

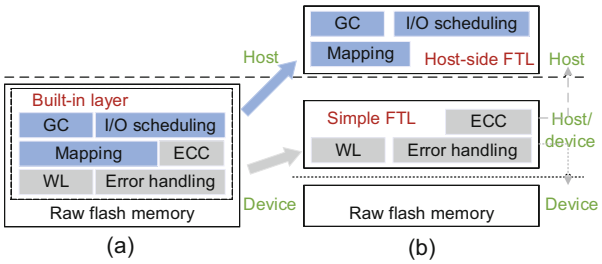

Fig. 1 FTL in SSDs (a) compared to FTL in OCSSDs (b). FTL: flash translation layer; SSD: solid-state drive; OCSSD: open-channel SSD; GC: garbage collection; I/O: input/output; ECC: error correcting code; WL: wear leveling

2.2 Properties of the flash translation layer

FTL is the core component of an OCSSD and is placed over the hardware. The design quality of an FTL directly determines the quality of OCSSDs in terms of performance, reliability, durability, and so on. An FTL can be of two forms: partly on the device side and partly on the host side, where the device-side light-weight FTL helps OCSSDs manage functions related to the hardware; or fully on the host side, to allow effective design and management. The FTL on the host side can be in the kernel space to achieve efficient I/O interactions or be in the user space to coordinate with applications conveniently.

A well-designed FTL can bridge the semantic gap between file systems and flash devices, co-design with host software, and eliminate redundant functions, such as space allocation and GC operations in multiple layers, which can further reduce the performance overhead.

2.3 Interface

OCSSDs often provide three types of interfaces for interaction: a general block interface, a portable operating system interface (POSIX), and a user-customized interface often co-designed with host-side software. The FTL can access the raw flash

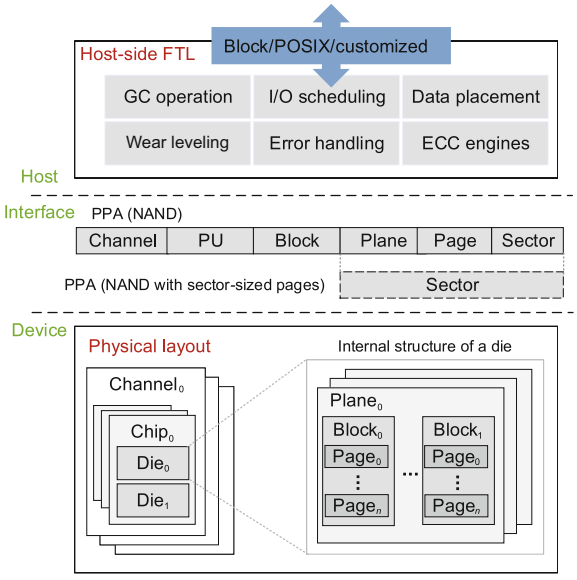

Fig. 2 A typical OCSSD architecture. OCSSD: open-channel solid-state drive; POSIX: portable operating system interface; FTL: flash translation layer; GC: garbage collection; I/O: input/output; ECC: error correcting code; PU: parallel unit; PPA: physical page address

memory through hardware interfaces such as the PPA (Fig. 2). The PPA relies on a hierarchical address space (Picoli et al., 2017), which allows the host side to see the media in a fine-grained manner and helps developers directly access the NAND flash cells using data commands including read/write/erase operations.

Generally, these operations have different granularities: page is the minimum unit for reads and writes, while block is the minimum unit for erases; different granularities of writes and erases may lead to write amplification (WA). These operations also have different access latencies: read is an order of magnitude faster than write/erase, and if a read operation is scheduled after performing a write/erase operation, it may lead to higher read latency. Therefore, a well-designed interface for monitoring and scheduling these operations is very important, which determines the high and stable performance of OCSSDs.

3 Opportunities

3.1 Key challenges of solid-state drives

SSDs have many performance advantages; however, due to the erase-before-write limitation (Lee C et al., 2015), the different write/erase granularities, and the disk-like block interface (Bjørling et al., 2017), there are several well-documented shortcomings that prevent SSDs from providing high QoS and fully exploiting the benefits of flash memory, as listed below:

1. Log-on-log problem (Yang JP et al., 2014)

Due to the functional completeness considerations, each software layer may prefer to perform the same function such as data mapping and GC, resulting in additional performance overhead.

2. Flash endurance reduction (Lu et al., 2013)

Extra write operations from outer systems and valid page migration from GC may incur WA and frequent program/erase (P/E) cycles, resulting in expensive bandwidth consumption and flash lifetime reduction.

3. Long tail latency (Hao et al., 2016; Kang et al., 2017)

This is usually caused by time-consuming GC operations and irrational I/O scheduling, which may incur busy writes/erases and hungry reads, resulting in long and unpredictable tail latencies.

4. Inadequate resource utilization (Ouyang et al., 2014)

SSDs may provide only $<50\%$ of the raw bandwidth to applications and often reserve 20%–30% or even larger space as over-provisioning (OP) for GC and other operations. This extremely low resource utilization may result in a significant waste of expensive resources on both initial and recurring costs.

These shortcomings present great challenges to SSDs and are often difficult to address because SSDs are treated as black boxes and are managed by the built-in FTL. An FTL can provide interoperability (Lee S et al., 2016) and automated management, but it also hides the design details of SSDs such as the execution of erase operations and internal parallelism, and has become a bottleneck to further improve SSDs' performance.

Compared to SSDs, OCSSDs can provide new ideas for many research breakthroughs and problem solutions, especially in domain-specific scenarios. OCSSDs expose their internal physical features, allowing host-side applications to customize the FTL according to their own needs, which brings unprecedented opportunities for developers to fully leverage the performance potential of OCSSDs while also posing some challenges for how to study them well under complex management costs.

Some studies argue that the internal structure of SSDs should not be opened (Yang LH et al., 2019) since it may reduce the security and flexibility of application accesses and raise design complexity. However, existing works on OCSSDs suggest that open designs for domain-specific scenarios can bring obvious performance benefits when properly designed.

3.2 Opportunities for open-channel solid-state drives

OCSSDs bring many opportunities for researchers in academia and industry to achieve specific performance goals or address practical problems in different scenarios. We focus mainly on the following five metrics: throughput, latency, lifetime, isolation, and resource utilization. In this subsection, we give only a rough discussion of some representative works to illustrate the performance opportunities presented by OCSSDs. More systematic methods will be analyzed in detail in Section 4 for clarity. More works on leveraging these opportunities to

achieve better performance are summarized in Table 2.

3.2.1 High throughput

SSDs can achieve only sub-optimal I/O throughput because of inefficient GC and under-utilized internal parallelism. Even flash-optimized file systems such as F2FS do not perform well under heavy write traffic. OCSSDs can address this shortcoming by allowing users to optimize the GC mechanism, redesign I/O scheduling, and fully exploit the parallelism to significantly improve the throughput.

Compared to F2FS, ParaFS (Zhang JC et al., 2016) proposes a two-dimensional (2D) data allocation to fully leverage OCSSDs' channel-level parallelism while reducing valid page migrating during GC operations, which may impact the data read/write throughput. Evaluations show that ParaFS can outperform F2FS by 1.6–3.1 times under write-intensive workloads.

FlashKV (Zhang JC et al., 2017) and LOCS (Wang P et al., 2014) leverage multi-channel parallelism to improve the throughput of log-structured merge tree (LSM-tree) based key-value (KV) stores. FlashKV uses a parallelism-friendly data layout to manage the raw flash space in the user space, which improves the throughput by 1.5–4.5 times compared with LevelDB. LOCS directly uses software-defined flash (SDF) (Ouyang et al., 2014) as the underlying hardware and adopts a dynamic I/O dispatching policy to fully exploit the channels, which improves throughput by >4 times. NVMKV (Marmol et al., 2015) leverages the advanced host-side FTL capabilities to enable its hash-based KV store, achieving high throughput and low WA while ensuring scalable and ACID (atomicity, consistency, isolation, and durability) compliant KV operations.

3.2.2 Low latency

Enterprise-grade systems need to provide high QoS for their tenants since unpredictable or unstable latency will hurt users' productivity. Even worse,

Table 2 Summary of the main optimization goals that some existing works try to achieve

Work	Throughput	Latency		Lifetime			Isolation	Resource utilization	
		Stability	Predictability	WL	WA	GC		Bandwidth	Miscellaneous*
OFTL				✓	✓				
SDF	✓	✓			✓	✓		✓	1
LOCS	✓	✓				✓			
SOFA	✓	✓		✓		✓			
NVMKV	✓				✓				
ParaFS	✓	✓				✓			
AMF	✓	✓			✓	✓		✓	2
FlashKV	✓	✓			✓	✓			
Multi-Partition							✓		
LightNVM	✓		✓	✓		✓			
TTFLASH			✓			✓			
DIDACache	✓	✓		✓		✓			
OC-Cache							✓	✓	1
ThermAlloc						✓			3
OCStore	✓	✓				✓			
StageFS	✓			✓	✓	✓			
hWA-BC							✓		
SSW	✓					✓			
QBLKe	✓					✓		✓	2
AOCBLK	✓		✓			✓		✓	
IODA			✓		✓	✓			
oNFC		✓						✓	1
JS-Pblk					✓			✓	
Prism-SSD	✓	✓		✓		✓			
PVSensing		✓							3

*Includes the following: (1) device-side capacity utilization; (2) host-side CPU/memory utilization; (3) 3D flash reliability. WL: wear leveling; WA: write amplification; GC: garbage collection (can also improve throughput and latency; here, we put it under lifetime for simplification)

long tail latency may slow down the response time or even cause service timeout which fails to meet the requirements of real-time and quality-critical systems (Kang et al., 2017). Fortunately, OCSSDs can solve these problems explicitly by adopting rational GC mechanisms and I/O scheduling strategies.

1. Predictability

LightNVM (Bjørling et al., 2017) is the first generic subsystem designed for OCSSDs. It exposes the physical layout of OCSSDs by the physical block device (PBLK) and narrows the semantic gap by a high-level I/O interface. It allows the developers to explicitly manage the OCSSD's PUs and design the GC and I/O scheduling according to workloads, which helps achieve predictable latency. By controlling the GC process in the flash array, IODA (Li et al., 2021) can achieve strong latency predictability and reduce the P95–P99.99 latencies to be under 2%.

2. Stability

ParaFS adopts a two-phase scheduling policy to fairly assign read/write/erase requests to the flash channels and achieves a consistent performance under write-intensive workloads. TTFLASH (Yan et al., 2017) introduces four key strategies to remove GC blocking from all software stacks and can achieve a guaranteed high performance. Evaluation results show that TTFLASH is 2.5–7.8 times faster compared to the base in average latencies.

3.2.3 Long flash lifetime

SSDs face severe WA and inefficient GC problems, which reduce flash endurance excessively. Fortunately, by using host-side FTL and exposing the physical layout of OCSSDs, developers can control data dispatching actions and data placements to prolong flash lifetime by reducing flash WA, achieving efficient GC, and addressing WL issues.

1. WA

For example, OFTL (Lu et al., 2013) uses an in-kernel FTL co-designed with the file system to reduce data updates and significantly reduce WA to 47.4%–89.4% in the SYNC mode and 19.8%–64.0% in the ASYNC mode compared with ext2, ext3, and btrfs. By bypassing the file system layer and avoiding partial page updates, FlashKV uses the in-file indexing to eliminate the cascading update and can greatly reduce the WA from the file system and FTL.

2. GC

SSW (Du et al., 2020) uses strictly sequential writes to make sure that the writes are not shuffled, which can reduce the number of GC operations on average by 26.65%. Some other works such as StageFS (Lu et al., 2019a) and QBLKe (Qin et al., 2021b) improve the flash lifetime by reducing valid page migrations during the GC process.

3. WL

To prevent flash memory cells from wearing out, Prism-SSD (Shen et al., 2022) uses a global WL module by shuffling hot/cold PUs based on the "update likelihood" (or hotness) (Lee C et al., 2015), and SOFA (Chiueh et al., 2014) uses a global FTL that provides both intradisk and interdisk global WL to prolong the lifetime of the flash array.

3.2.4 Strong performance isolation

In multi-tenant cloud environments, cloud providers need to provide isolated and stable services to their tenants. Dedicated devices for each tenant may provide great performance while incurring unacceptable cost and management overhead as the tenant scales expand dramatically. Many providers choose SSDs as their storage devices. They use software methods such as Dockers to create isolation space for each tenant or hardware techniques to separate tenants to different flash blocks. However, the unpredictable global GC operations caused by one tenant may affect other tenants' experiences. Meanwhile, tenants still share the flash channels, which may cause excessive conflict in heavy workloads and reduce the overall performance. OCSSDs bring new opportunities for strong performance isolation because of their exposed device geometry.

Multi-Partition (González and Bjørling, 2017) instantiates several PBLKs (Bjørling et al., 2017) to extend LightNVM with multi-target support and to separate tenants to dedicated instances and PUs (Fig. 3a). Each instance executes independent GC operations and I/O scheduling as if every tenant owns the hardware resource and will not be disturbed by other tenants.

On the other hand, OC-Cache (Wang HT et al., 2018) achieves strong performance isolation by using OCSSDs as I/O caches. As shown in Fig. 3b, it assigns one or more flash channels exclusively to one tenant to ensure strong performance isolation and reserves some shared channels for dynamical assignment according to the tenant's miss ratio curves of

workloads, which can provide better QoS and leverage the OCSSD's internal parallelism.

Lee S et al. (2019) aimed to achieve performance isolation between multiple processes rather than between tenants. They proposed a host-level workload-aware budget compensation (hWA-BC) scheduler based on LightNVM. The scheduler focuses mainly on the application's contribution to GC and takes two factors (the read–write request ratio and the valid–invalid flash page ratio) into consideration. Evaluation results show that hWA-BC succeeds in compensating/penalizing the workloads and ensures performance isolation.

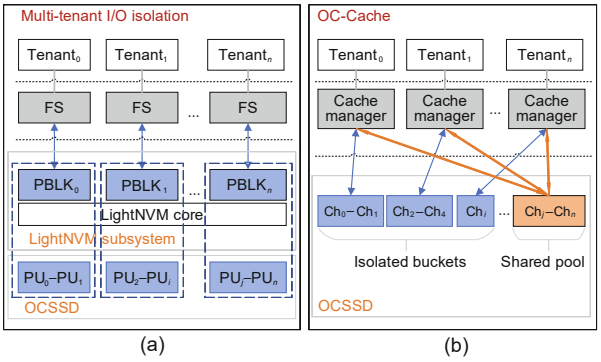

Fig. 3 Design of strong performance isolation: (a) Multi-Partition; (b) OC-Cache. I/O: input/output; FS: file system; PBLK: physical block device; PU: parallel unit; Ch: channel; OCSSD: open-channel solid-state drive

3.2.5 High resource utilization

By exposing the physical layout and removing the FTL to the host side, OCSSDs can reach higher resource utilization than SSDs. The resource utilization we have discussed in this subsection includes mainly bandwidth utilization, device-side capacity utilization, host-side CPU/memory utilization, and data reliability utilization that can avoid unnecessary error data accesses and improve critical data utilization. More information can be seen in Table 2.

1. Capacity

DIDACache (Shen et al., 2017) can dynamically adjust the OP to maximize the usable flash space for KV caching to increase the cost efficiency. SDF introduces a field-programmable gate array (FPGA) based controller and implements a host-side FTL to directly access the raw flash channels, avoiding reserving space for GC and parity coding in SSDs, which makes SDF achieve 99% user capacity and reduce per-gigabyte hardware cost by 50% on average.

2. Bandwidth

SDF can also achieve 95% of the flash's raw bandwidth by highly exploiting the channel-level parallelism. LightNVM can reach a high write bandwidth based on OCSSDs. However, it performs poorly under multi-thread workloads due to the use of global spinlocks in PBLK. Qin et al. (2021b) proposed QBLKe to improve the scalability and reduce software overhead. By adopting three techniques (see details in Section 4.2.1) to avoid the global spinlocks, QBLKe can improve the write bandwidth up to 78.9% compared with PBLK under 32-thread 4 KB random write test.

3. Data reliability

Three-dimensional (3D) NAND flash memory provides ultra-high capacity compared to the planar one; however, it suffers from severe thermal problems that lead to data loss and a sharp decrease in performance. ThermAlloc (Wang Y et al., 2019) distributes the centrally accessed data to different physical locations with a certain distance and postpones or reduces unnecessary GC operations to balance heat issues using OCSSDs, which makes ThermAlloc reduce the peak temperature by 30% and improve the data reliability. PATCH (Chen et al., 2019) focuses on the unreliability issue caused by low cell current in 3D flashes. It can identify the unreliable blocks and allocate/reallocate the write data to other reliable blocks, which can reduce the uncorrectable bit errors and avoid unnecessary data access. PVSensing (Wang Y et al., 2022) considers the process variation and allocates write requests to the corresponding physical blocks to reduce uncorrectable bit errors. Compared to PATCH, PVSensing can reach a more effective reduction of the uncorrectable bit errors and improve the utilization of reliable flash memory.

From the above-mentioned performance benefits, we can see that OCSSDs outperform SSDs in many specific scenarios. These opportunities come mainly from the carefully designed host-side FTL and the exposed multi-level parallelism. Meanwhile, developers need to consider their workloads and application behaviors to design the data access interfaces, the efficient GC mechanism, and the I/O scheduling strategy to better explore the performance advantages of OCSSDs.

4 Methodologies

Although OCSSDs bring many opportunities to achieve better performance, we should realize that achieving optimal performance rather than sub-optimal performance is challenging. There are some issues that have to be addressed at the software level for the opportunities to be fully turned into real performance improvements, especially designing effective methods according to the workloads and application behaviors under complex software/hardware environments. In this section, we survey the related works in respect of their research methods from five different perspectives. An overview of some typical optimization methods is shown in Table 3. We will discuss these methods in detail and try to answer the following questions:

1. What approaches do existing works adopt to exploit the performance benefits of OCSSDs?

2. What unique issues are addressed with these approaches?

3. What are the advantages and potential shortcomings of these approaches?

4.1 Interface design

Unlike well-packaged SSDs, with OCSSDs, developers can customize interfaces that are exposed to the host while ensuring effective access to the device. The low-level interfaces should consider the device geometry; the high-level interfaces need to ensure high efficiency, delivering semantic information efficiently, and good compatibility with existing systems, which can bring many performance advantages and good usability/scalability.

LightNVM proposes a classic three-layer architecture: a high-level I/O interface for user-space applications, an FTL in kernel space for main functionalities, and a non-volatile memory express (NVMe) device driver for hardware management. Each layer provides a different abstraction based on its characteristics and can interact well with each other through appropriate interfaces. LightNVM provides a PPA interface to expose its device geometry and parallelism, which is implemented to meet the NVMe standard (NVM Express, Inc., 2023). Meanwhile, host applications or file systems can access OCSSDs through a traditional block I/O interface provided by PBLK to reduce software changes. Moreover, LightNVM allows developers to customize their own application-specific interfaces by exposing geometric addresses to the user space.

Many other studies also customize interfaces for their practical needs. For example, as an object store, OCStore (Lu et al., 2019b) uses an object interface for host applications for better interaction. FlashKV combines the compaction procedure with GC and I/O scheduling, leaving only an input/output control (ioctl) interface to dispatch requests to FTL for simplification. OFTL (Lu et al., 2013) co-designs software and hardware in embedded systems and provides fine-grained byte-unit object interfaces to file systems, which can compact small updates into fewer pages to reduce WA. To provide different operational granularities for GC efficiency, SDF (Ouyang et al., 2014) customizes an asymmetric read/write/erase operation interface to external software.

All these designs try to ensure better interaction across the external software, the FTL, and the underlying hardware. However, application-friendly flexible interfaces and optimal performance are difficult to achieve simultaneously. To bridge this gap, Prism-SSD (Shen et al., 2022) proposes three-level abstraction of interfaces to expose the OCSSD hardware, namely the raw-flash-level abstraction, the flash-function-level abstraction, and the user-policy-level abstraction. Prism-SSD provides many application programming interfaces (APIs) and allows developers to choose the best trade off between usability and efficiency, and the multi-level abstraction may give us more inspiration to better design the OCSSD interfaces.

4.2 Flash translation later design

FTL provides OCSSDs rich functions such as GC, I/O scheduling, and data placement. It is a challenge for FTLs to provide high performance and low software overhead while ensuring high scalability and usability (Qin et al., 2021b). Without properly designing the FTL software architecture, it is hard to fully leverage the OCSSD's advantages and may put a heavy load on the host hardware.

4.2.1 Generic design

As the first generic subsystem for OCSSDs, LightNVM implements its FTL called PBLK in the kernel space and also leaves a simple controller on

Table 3 Some typical methods that use open-channel solid-state drives (OCSSDs)

Work	Interface	FTL position	Parallelism	GC	Scheduling	Scenario
OFTL	Object	Kernel	–	Coarse-grained block state maintenance	Compacted updates	Embedded system
SDF	Asymmetric ioctl	Kernel (per-channel FTL)	Channel-level, plane-level	Increasing the write unit size	Scheduling erasure operations during an idle period	Large write workloads
LOCS	Customized/ioctl	User/kernel	Channel-level	Delaying GC operations to process along with write requests	Four scheduling and dispatching coordinated policies	KV stores
FlashKV	Customized/ioctl	User/kernel	Channel-level	Combined with software	Priority-based scheduling	KV stores
DIDACache	Customized/ioctl	User/device	Channel-level	Dynamic space-based/locality-based eviction	Load balance and OPS management	KV caches
OCStore	Object/ioctl	Kernel/device	Channel-level	Per-channel GC	Transaction-aware scheduling	Object stores
ParaFS	Block/ioctl	Kernel/kernel	Channel-level	FS/FTL and foreground/background coordinated GC	Parallelism-aware scheduling	File system
AOCBLK	PPA/vector	Kernel/device	PU-level	Sending only GC copyback commands to devices	–	Heavy workloads
AMF	Block	Kernel/device	Channel-level	Reducing live data copy	Per-channel FIFO scheduling	File system
LightNVM	NVMe/PPA/block/vector	Kernel/device	All levels	The least valid pages first	PID controlled rate limiter	Linux subsystem
OC-Cache	Based on LightNVM	Based on LightNVM	Channel-level	Heat-aware cache replacement	Adaptive cache migration	Multi-tenants
QBLK/QBLKe	Based on PBLK	Based on PBLK	All levels	Per-channel GC	Score-based rate limiter	Multi-thread workloads
SSW	Working as a component of LightNVM	Working as a component of LightNVM	Channel-level	Strict sequential writing to reduce GC operations	Write request rescheduling and sequential–random write splitting	–
ThermAlloc	Working as a component of LightNVM	Working as a component of LightNVM	Chip-level	Postponed GC strategy	Thermal balancing strategy	3D flash memory
PATCH	Working as a component of LightNVM	Working as a component of LightNVM	Chip-level	–	Live migration strategy	3D flash memory
TTFLASH	–	Host/device	Channel-level, plane-level	Four strategies to remove GC blocking at all software levels	GC-tolerant read and GC-tolerant flush	Latency sensitive
MT-Cache	Based on PBLK	Based on PBLK	–	–	Hotness allocation policy	–
SOFA	SATA command	Host (global FTL)/device	Disk-level	Cost-aware GC	Global wear leveling	RAID

FTL: flash translation layer; GC: garbage collection; PPA: physical page address; NVMe: non-volatile memory express; PBLK: physical block device; SATA: serial advanced technology attachment; PU: parallel unit; FS: file system; OPS: over-provisioning space; FIFO: first input first output; PID: proportional, integral, and derivative; KV: key-value; RAID: redundant array of independent disk

the device side to manage the hardware well. PBLK provides rich functions such as GC, handling errors, and mapping logical address to physical address (L2P) for LightNVM, and deals with specific constraints on flash media and device controller. With the help of PBLK, LightNVM can provide high performance and small software overhead for the host applications.

However, general-purpose systems usually face the dilemma between performance and safety, just like Qin et al. (2021b) found that PBLK was limited under multi-thread workloads. The limitation comes from the global spinlocks used in the three components, i.e., the data buffer, translation map, and direct memory access (DMA) memory pool. The global spinlocks offer unnecessary protection for security and hurt the scalability of PBLK, which makes host software unable to fully exploit the OCSSD performance.

To address this limitation, they proposed QBLKe, an extended version of QBLK (Qin et al., 2019). QBLKe uses three main optimization techniques: the per-CPU ring buffer technique which allocates a ring buffer to each CPU to minimize the competition that comes from other buffers, the lock-free translation map which uses the atomic primitives to ensure the atomicity of operations, and the per-CPU DMA pool which assigns a DMA memory pool for each CPU to eliminate competition for the lock. Meanwhile, QBLKe allocates a GC thread for each channel to decrease GC page-migration overhead. All these optimization methods make QBLKe more scalable under multi-thread workloads and can achieve higher throughput and write bandwidth. By leveraging the advantages of QBLKe, they further developed a Linux file system called NBFS (Qin et al., 2021a) to achieve atomic writes with elimination of I/O orderings and provide a double performance of F2FS.

4.2.2 Specific design

Some previous works (e.g., Lu et al., 2013; Zhang JC et al., 2016) prefer to design dedicated FTLs with different placement strategies to achieve specific performance under domain-specific scenarios. Putting FTLs in the user space is beneficial for co-designing with applications. Leaving FTLs in the kernel space can reduce workloads on the device side and make them interact well with the file system while sensing the upper-level semantic information. Some other works (e.g., Lu et al., 2019b; Zhang XY et al., 2021) choose to leave a simple FTL still on the device side, which can use the ability of the device to do some simple operations closely related to hardware management and offload the host-side workloads to the device. The four types of classic domain-specific FTL design patterns are shown in Fig. 4.

1. Only in kernel

OFTL implements all the FTL functions in the kernel space (Fig. 4a), which can interact well with the upper system and directly access the raw flash memory. To significantly extend flash memory lifetime by reducing WA in FTL, OFTL buffers and compacts frequent partial page updates to reduce the number of updates and designs a lazy indexing technique that uses backward pointers to reduce index metadata persistence and journaling overhead. Although OFTL is an early work and does not consider data hotness and the optimization of GC, the software/hardware co-designed concept allows designing flexible approaches to solve specific bottlenecks for better performance.

2. FS-kernel

ParaFS (Zhang JC et al., 2016) uses a simple lightweight in-kernel FTL called S-FTL only for WL, ECC, and block-level mapping, leaving the application-related functions such as GC and I/O scheduling in the file system (Fig. 4b). In particular, the mapping is designed as static in a log-structured way (O'Neil et al., 1996) and has no in-place updates, which means that it does not need remapping operations and incurs nearly zero overhead. In addition, S-FTL knows the behaviors of the file system by exposing read/write/erase interfaces and ensures that the data grouping at the file system layer will not be shuffled in the flash channels. To sum up, ParaFS renders it easy to achieve efficient GC and reasonable I/O scheduling by integrating them with the file system; moreover, letting the S-FTL execute only simple commands directly related to the flash memory can reduce redundant functions between multiple layers and improve the overall performance.

3. Application-kernel

User-space applications often need to cross multiple software layers such as application layer, file system layer, and FTL to access devices, which will induce severe WA and GC overhead under

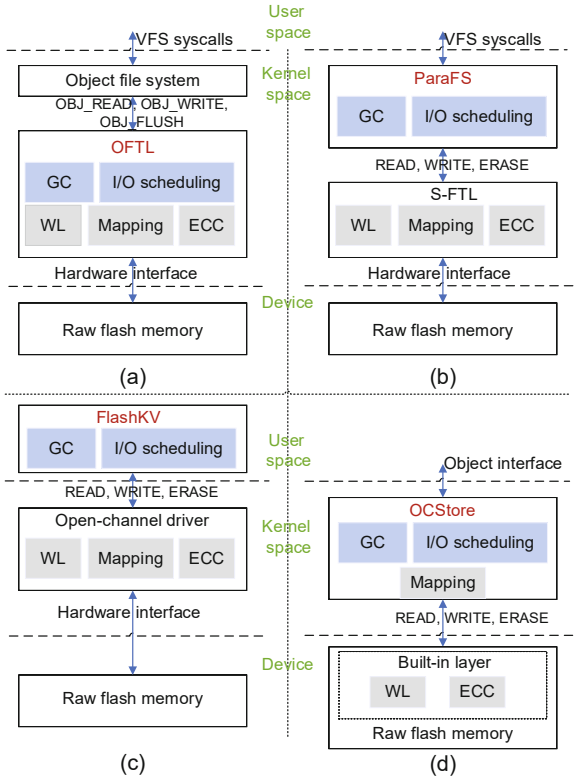

Fig. 4 Four types of domain-specific FTL designs: (a) OFTL; (b) ParaFS; (c) FlashKV; (d) OCStore. FTL: flash translation layer; GC: garbage collection; I/O: input/output; WL: wear leveling; ECC: error correcting code

write-intensive workloads if these layers take redundant functions (Fig. 5); functions with the same color mean reduplicated design. FileStore (Weil et al., 2006) implements mapping at all three layers and journaling in the application and file system layers, while BlueStore (Weil et al., 2006) implements GC and space allocation at both the host side and device side, which introduces extra software overhead.

The redundant functions are usually caused by weak semantic interfaces. Each layer prefers to design independent functions toward its own needs rather than co-design with other layers. Therefore, optimizing the software stack design, breaking through the multi-layer semantic isolation, and reducing the redundant or even conflicting functions overhead, are important methods to improve the performance.

A direct and effective way to eliminate the redundant functions of multiple layers is bypassing the intermediate coupling layer. For example, FlashKV accesses flash memory directly bypassing the embedded FTL and file system. It separates its FTL into

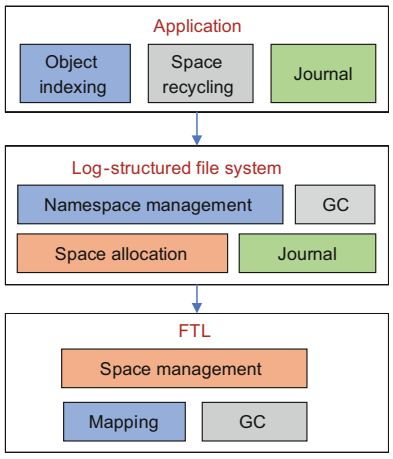

Fig. 5 Redundant functions across multiple layers. GC: garbage collection; FTL: flash translation layer (References to color refer to the online version of this figure)

two parts, a thick open-lib in the user space co-designed with KV stores and a thin open-channel driver in the kernel space (Fig. 4c); these interact with each other through an ioctl interface. The open-lib implements most of the functions, such as GC and I/O scheduling, to better meet the feature of applications, and the open-channel driver implements only simple functions that are deeply related to the hardware. This design allows FTL to know more about the application behaviors while leveraging the OC-SSD features better, which can reduce the overhead of multi-layer management.

Another typical example is DIDACache (Shen et al., 2017), which co-designs the KV cache with user-space FTL and eliminates unnecessary functions by bypassing multiple intermediate layers to bridge the semantic gap. This co-designed and bypassed approach can make the KV cache manager directly drive the underlying flash devices and bring many benefits, such as directly mapping the "keys" to physical flash pages to reduce the redundant mapping cost, eliminating multi-layer redundant GCs to reduce the processing overhead, and sensing the real-time workloads to adaptively tune the OP space size. These optimizations bring DIDACache a high throughput and a low latency while removing unnecessary erase operations.

The bypass method is an effective way to eliminate redundant functions; however, due to the complexity of the computer system architectures and the benefits of low-coupling design in implementation, the multi-layer software stack will not die and not

every design can bypass the middle software layers. It is still a challenge to bridge the semantic gap and reduce the redundant functions while keeping independent design and implementation. This involves not only the design of FTL but also the collaborative design of other software layers.

4. Kernel-device

The fourth FTL placement strategy is still leaving a simple FTL on the device side, which performs only hardware-closely-related functions, such as ECC and WL, to reduce additional interaction overhead but perform other functions on the host side, such as OCStore (Fig. 4d). OCStore co-designs FTL functions with the object storage system in the kernel space and removes redundant functions across multiple software layers by directly managing the flash memory, which makes OCStore better understand the software behaviors and reduce additional performance overhead.

In addition to the FTL placement rules, different mapping granularities and strategies will affect the FTL performance and management overhead. Page-level mapping adopted by LightNVM can reduce WA and provide fine-grained address management. However, as the capacity increases, the mapping table consumes the host memory excessively and is hard to be loaded entirely in memory.

To address this, AMF (Lee S et al., 2016) and MT-Cache (Oh and Ahn, 2021) adopt block-level mapping to reduce mapping tables. MT-Cache further employs a selective mapping table loading scheme that loads only part of the mapping table into the host memory to reduce memory footprint and designs a cache replacement policy to ensure a low cache miss ratio, enabling a final benefit that pays only 32% memory consumption for 80% I/O performance. FlashKV introduces a more coarse-grained mapping, namely, super-block mapping, to reduce the scale of mapping entries, which can be entirely cached in host memory. However, the coarse-grained mapping will produce WA and write latency under small writes, and choosing a fine- or coarse-grained mapping should be considered according to workloads and parallelism design.

4.3 Internal parallelism exploitation

The exposure of the internal parallelism of OCSSDs provides external software great opportunities to fully exploit the potential performance. Existing works (e.g., Wang P et al., 2014; Zhang JC et al., 2017) focus mainly on channel-level parallelism; they have achieved performance goals such as high throughput and efficient resource utilization, and some of them (e.g., Wang HT et al., 2018) also leverage the independent channels to solve specific bottleneck problems such as strong performance isolation. There are three main challenges to exploiting the internal parallelism: (1) How to ensure that the semantic data grouping and data order are not shuffled? (2) How to dispatch data to ensure a wide parallel stripe and high utilization of PUs? (3) How to design a good data placement for better using different levels of parallelisms? To address these challenges, existing works take a range of striking approaches.

1. Independent channel parallelism

SDF rationally exploits the channel- and plane-level parallelism by the shape of their large write workloads. As shown in Fig. 6a, it treats each channel as an independent device (44 channels from $/\mathrm{dev}/\mathrm{sda}_0$ to $/\mathrm{dev}/\mathrm{sda}_{43}$) and each channel has its own FTL engine ("SSD Ctrl" in the figure). SDF deploys multiple threads to different channels to achieve high concurrency.

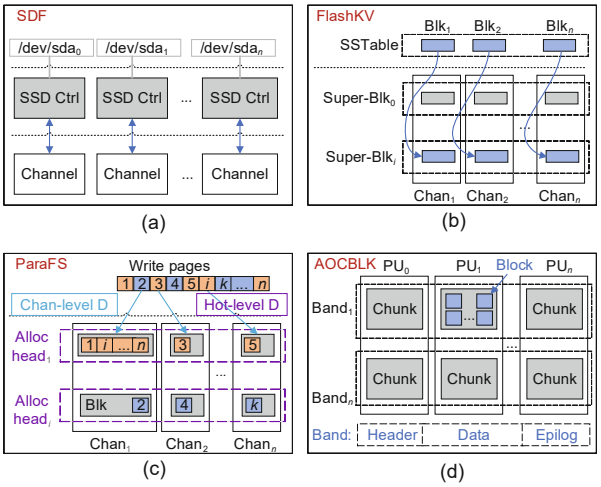

Fig. 6 Four instances of exploiting OCSSD's internal parallelism: (a) SDF; (b) FlashKV; (c) ParaFS; (d) AOCBLK. Chan: channel; Blk: block; D: dimension; PU: parallel unit; OCSSD: open-channel solid-state drive (References to color refer to the online version of this figure)

LOCS (Wang P et al., 2014) uses SDF as a fast storage device to improve the performance of user-space LSM-tree-based KV store. The advantage

of using SDF is that it enables the extended LevelDB to schedule I/O operations to the 44 channels. To exploit SDF's channel-level parallelism, LOCS uses multiple immutable MemTables instead of one to store more incoming data, which allows write requests to be issued concurrently. Additionally, each flash channel is assigned to a single I/O request queue, which can reduce queuing latency of multiple requests and effectively leverage channel concurrency.

Independent channel management facilitates the strong performance isolation we discussed in Section 3.2.4; it also contributes to SSD virtualization, which can provide isolated virtual SSDs for tenants on dedicated channels (Huang et al., 2017). BlockFlex (Reidys et al., 2022) even enables dynamic storage harvesting with independent channels in modern cloud platforms to maximize the storage utilization while ensuring performance isolation between virtual machines. However, this management is friendlier to isolated objects, and most of the time, we need to consider multi-channel co-parallelism according to the hotness/locality of workloads.

2. Coarse-grained striping

Coarse-grained striping not only reduces address-mapping management overhead but also facilitates channel-level parallelism. For example, FlashKV implements the KV store by proposing a parallel data layout that adopts super-block-level mapping to exploit the internal parallelism while keeping low metadata overhead. As shown in Fig. 6b, the super-block has the same size as SSTables, which renders it easy to manage the SSTables and has fewer mapping entries. Flash blocks within the same super-block can be accessed easily in parallel to fully leverage the channel-level parallelism because they are from different channels.

However, LOCS may cause serious channel contention due to its coarse-grained striping method, and FlashKV's large SSTables may induce high tail latency when doing KV writes. KVSSDs (Wu et al., 2021) integrate LSM trees and the FTL at the device side and introduce a fine-grained dynamic striping policy to exploit the internal parallelism of SSDs. However, KVSSDs are evaluated only in a simulated environment and are hard to integrate into real SSDs. We suggest that KVSSDs should integrate LSM trees with a host-side FTL, which may benefit more from the exposed internal parallelism of OCSSDs.

3. Channel-level parallelism with hotness grouping

F2FS fails to fully exploit the internal parallelism of flash devices due to the semantic isolation caused by the built-in FTL. ParaFS absorbs the good points of F2FS but uses OCSSDs as flash devices. ParaFS designs a 2D data allocation mechanism, namely, channel-level dimension and hotness-level dimension, to fully use flash channels while keeping effective hot/cold data grouping (Fig. 6c). In the channel-level dimension, the write data are divided into pages, which are striped to different channels, and can be persistent in flash memory in parallel. By doing so, the allocation process turns to the hotness-level dimension that groups pages with different hotness (in the figure, classifying the yellow pages and blue pages as different groups) and assigns these groups to different allocator heads with the same hotness. This parallel mechanism makes ParaFS fully leverage the channel-level parallelism while keeping an efficient GC.

4. Fine-grained parallelism

AOCBLK (Zhang XY et al., 2021) uses a fine-grained data placement to exploit PU-level parallelism. In the data layout, chunks in different PUs with the same index are managed as a band that is orthogonal to the PUs (Fig. 6d). Data are written sequentially to a band; each band has a header to record some unique numbers for consistency checks and an epilog at the end to store metadata. AOCBLK further adopts a subpage-based (e.g., 16 KB) data interleaving placement policy to better exploit the internal parallelism in PUs. Compared with traditional page-based (e.g., 96 KB) data placement, this fine-grained policy can better leverage the advantages of parallelism.

Although many works have tried to explore the internal parallelism of SSDs, Qiu et al. (2021) concluded that multi-level interleaving and cache mode pipelining were still not well exploited. They proposed an open-way NAND flash controller (oNFC) to support all the four parallelisms and a dual-level command scheduler that is integrated into the NFC to enable fine-grained plane-level interleaving. Evaluation results show a 93% theoretical bandwidth and 1.9–3.1 times speedup on average in page reading and programming latencies, respectively.

4.4 I/O scheduling optimization

Due to the black-box design, SSDs lack semantic information of upper systems. Weak upper-level scheduling policies may lead to hungry reads and busy erases, resulting in long-tail latencies and performance decline. Although OCSSDs expose their physical geometry, how to design rational I/O scheduling policies according to I/O features is still a challenge.

1. For overall throughput

To improve the I/O throughput, LOCS designs four dispatching policies according to I/O requests from LevelDB. When write requests are dominant, the round-robin dispatching policy is selected, which can assign the write requests evenly to all the channels to ensure the load balance of each channel (Fig. 7a). The advantage of this policy is its simplicity; however, when read/erase requests are intensive (both are required to be specified to specific channels), this policy will not work well, and the least weighted-queue-length write dispatching policy will be adopted (Fig. 7b). This policy maintains a weighted length table and assigns different weights

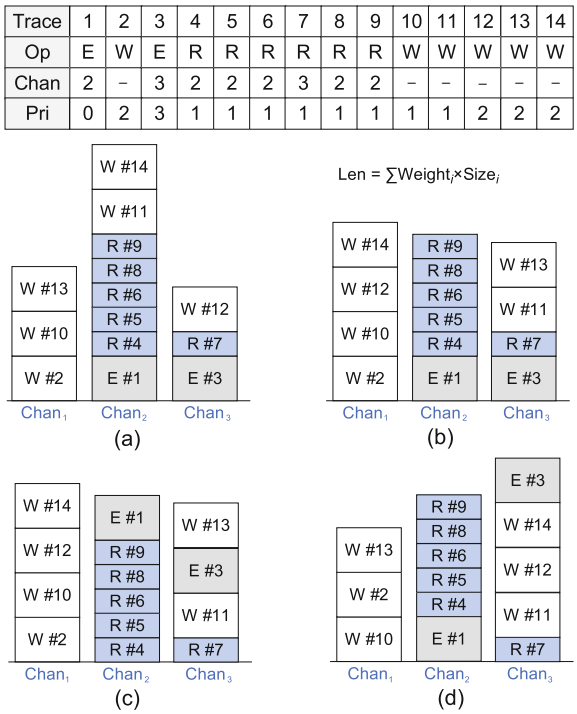

Fig. 7 I/O scheduling policies in LOCS and FlashKV: (a) round-robin dispatching; (b) least weighted-queue-length; (c) scheduling optimization for erase; (d) priority-based scheduling. I/O: input/output; Op: operation; Chan: channel; Pri: priority

to the read/write/erase requests to predict the processing latency in channels. Using the weighted queue length, the channels become more balanced but not friendly to the read requests. Based on these dispatching policies, LOCS further considers the dispatching optimization for write compaction and erase operation, which delays the processing of erase requests, and schedules them until there are enough write requests (Fig. 7c). This optimization is based on the idea that the erase process is not on the critical path and that write requests can balance the queue length. Evaluation results show that adopting the dynamic dispatching policy can better improve the overall throughput.

ParaFS adopts a parallelism-aware scheduling to schedule I/O requests in two phases under heavy-write workloads. In the dispatching phase, as shown in Fig. 8a, read and write requests are assigned different weights (W_{read} and W_{write} are set to "1" and "8," respectively), and a channel with the smallest calculated weight will be considered as the least busy one, which will be selected to dispatch the write request. In the scheduling phase, time is sliced evenly between read and write/erase requests to ensure predictable read latency. In the write/erase time slice, an algorithm is used to schedule write/erase requests. As shown in Fig. 8b, the algorithm considers the used space, free space (parameter f), and the percentage of processing the erase requests (parameter N_e). When e is larger than 1, the scheduler will send write requests in the corresponding channel; otherwise, it sends erase requests. This balanced scheduling helps ParaFS achieve more consistent performance.

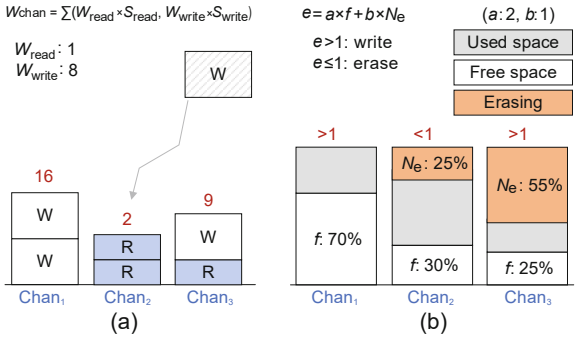

Fig. 8 The two scheduling phases in ParaFS: (a) dispatching phase; (b) scheduling phase. Chan: channel

2. For read throughput

Read requests may be blocked by write requests, which will cause severe read throughput

degradation. FlashKV designs an adaptive scheduling mechanism to dispatch read/write requests in compaction procedure by adopting two compaction threads. When in write-intensive workloads, it uses full write parallelism to accelerate the compaction, while in read-intensive workloads, the compaction threads will limit write requests to help improve the read throughput. To better optimize the read operations, FlashKV separates the read requests into clients' reads and compaction reads. Clients' reads are always small and discontinuous and can be managed in pages, while compaction reads are always large and localized and can be managed in batches. Moreover, as seen in Fig. 7d, FlashKV designs a priority-based scheduler (a smaller number means a higher priority) that can execute high-priority requests earlier according to the response time or the available capacity; e.g., read requests are executed before write requests and foreground requests before backgrounds, which can ensure better performance and user experience.

3. For latency

The transaction is latency-sensitive; however, the slowest I/O request determines the latency of the transactions. OCStore designs a transaction-aware scheduling policy with three strategies to achieve low transaction latency and stable performance. The first strategy is balancing the stripe width (the number of parallel channels) and stripe length (the number of pages in one channel) to reduce latency jitter. The number of channels is related to the parallelism, while the latency affects the overall stability of the performance in one channel, which means that the larger the stripe width, the higher the parallelism and the greater the performance jitter. The second is finding the lightest-load (using an estimated execution time) channel to dispatch the transaction requests to maintain a balanced latency. The third is assigning I/O requests in each channel the same time slices and setting a deadline for each request to make sure that every request is processed in time to avoid long tail latency. However, this strategy may not be friendly to the reads with high priority; although it can achieve stable performance, the read performance is limited.

Although the above-mentioned works can provide rational I/O scheduling strategies, they still suffer from frequent synchronous I/O overheads under partial-page writes and small hot/cold data grouping, which will incur severe WA and performance degradation. Therefore, Lu et al. (2019a) proposed a flash-friendly data layout and designed StageFS to provide balanced I/O scheduling with content- and structure-separated data updating. StageFS introduces a staging phase to absorb synchronized writes and a patching phase to lazily perform data allocation and grouping, especially in the staging phase. Small writes are appended using byte-addressable record units rather than page units to reduce WA. Evaluation results show that StageFS can improve performance by up to 211.4% and achieve lower GC overhead under frequent I/O synchronous workloads.

4.5 Garbage collection optimization

GC is an essential but expensive function for flash devices since they do not support in-place updating and must erase the original location before writing data. Due to the serious asymmetry of write and erase granularities, unreasonable small random writes and data assignments may cause inefficient GC operation, which may incur severe I/O delay and valid page migration overhead, resulting in serious WA, performance jitter, and long tail latency. Therefore, changing the write granularity, reasonably using data locality, reducing unnecessary erase operations, and so on are effective ways to optimize the GC mechanism.

1. Data locality

Writing data in a log-structured way can not only increase write throughput but also reduce the effects of random writes to achieve higher GC efficiency. To eliminate interruption of sequential writes by random writes, SSW proposes strictly sequential writing that can differentiate random requests from sequential requests by checking the metadata. SSW caches the sequential writes and random writes into different buffers and then dispatches them to different channels for concurrency while keeping them separately (as shown in Fig. 9a, 0–11 are identified as sequential writes and 13–16 as random writes). From the figure, we can see that SSW tends to maintain the data locality in the same flash block, which is beneficial for GC and can reduce valid page migration to improve the flash lifetime, but the channel-level parallelism is not well exploited and the data stripe is limited.

Different from SSW, AMF introduces a block I/O abstraction to dispatch the data to each channel

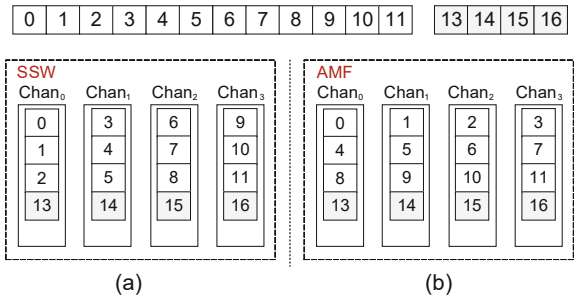

Fig. 9 Data placement strategies in SSW (a) and AMF (b). Chan: channel

for high parallelism (Fig. 9b). It can achieve a high stripe bandwidth but will incur inefficient GC operations. To fix this, AMF increases the number of inode-map segments to facilitate inode-map blocks to have more invalid data before being selected as victims; meanwhile, AMF separates the hot-data blocks from cold-data segments to improve the GC efficiency by identifying data features.

2. Hotness grouping

Separating data according to their access/update frequencies will leave data with the same hotness in the same group, which brings two advantages. First, data with the same hotness can be striped to different channels in parallel and accessed concurrently (Fig. 6c). Second, we can distinguish the data lifetime by identifying their hotness, meaning that data with the same hotness always have the same life span and can be assigned to the same flash block to reduce the migration of valid pages in GC processing. This hot/cold data grouping can significantly affect the GC performance.

Take Ext4 journaling file systems as an example; journal and file data usually have different hotness. In most cases, the journal may be deleted after the corresponding file data are written; recording them in the same block may increase extra GC overhead. However, SSDs find it hard to separate journals from file data. Thus, Son and Ahn (2021) proposed a host-side FTL to distinguish the two data types and reduce the GC overhead by writing journal pages to separate blocks. However, this scheme is too simple to identify the hotness of file data and improves only limited GC performance.

In recent years, multi-stream SSDs (Bhimani et al., 2017) have been developed to write data with the same or similar lifetime to the same erase unit according to the stream IDs, which can greatly improve the efficiency of GC and reduce the WA. However, due to the unawareness of host data features, multi-stream SSDs need to rely on the lifetime-based stream IDs identified and generated by the host side. Additionally, multi-stream SSDs are more like a compromise scheme between SSDs and OCSSDs; they are not as efficient as OCSSDs because the OCSSD FTL can directly identify the data lifetime and group data into channels for parallel writing.

3. Write granularity

ParaFS also designs a large write segment whose size and address are aligned to the flash block for both memory allocation and GC. This equivalent design can avoid valid page migration in a victim block to reduce WA and GC overhead. SDF adjusts its write granularity to reduce the GC overhead according to its large write workloads. It improves the write granularity to be several times larger than the erase size and makes the write units align with the erase blocks. The benefit is obvious: during GC, one block is impossible to contain both valid and invalid data pages, so no extra valid pages need to be migrated and the WA is eliminated. However, this strategy is not friendly to small writes.

4. Coordinated GC mechanism

F2FS uses a foreground/background coordinated GC mechanism to improve its overall GC efficiency. The foreground GC adopts the greedy policy which selects a section with the fewest valid blocks to minimize the time latency, and the background GC adopts the cost-benefit policy, which considers both the number of valid blocks in a section and the lifetime of the section. The foreground GC will be triggered when there is not enough free space, while the background GC is woken up periodically by the kernel. ParaFS adopts the similar mechanism at both the file system level and the FTL level to better optimize the GC efficiency. It employs several greedy foreground threads (one thread to each channel) for write-intensive workloads to do block recycling quickly and assigns a manager thread to wake up these GC threads as necessary. The cost-benefit background thread is triggered when the file system is idle. Both the foreground and background GCs in the file system migrate only the valid pages and mark the victim blocks as erasable; it is the GC in the FTL that directly does the erasing according to the trim commands from the file system to avoid additional copies. This coordinated GC in ParaFS improves the GC efficiency and also reduces performance jitter.

5. GC-free optimization

A well-designed fine-grained GC mechanism on OCSSDs can bring significant benefits for latency. TTFLASH (Yan et al., 2017) introduces four key strategies to remove GC blocking at all software levels to achieve low tail latency. First, TTFLASH adopts a plane-blocking GC (Fig. 10b) rather than channel-blocking GC (Fig. 10a) to ensure that the GC is exerted only on the affected planes using an intra-plane copyback mechanism, and that the channel can continue to provide I/O service for other non-GC planes. Second, by leveraging the idea of redundant array of independent NAND (RAIN), TTFLASH combines parity-based redundancy with GC operations, called GC-tolerant read (GTR), which can proactively regenerate the blocked read-page content by reading the parity from another plane to eliminate the waiting for GC completion. For example, in Fig. 10c, if page 2 cannot be fetched during the GC, its page content can be regenerated by XORing pages 0, 1, and the parity page, which is several orders of magnitude lower than waiting for GC completion. In addition, TTFLASH designs a rotating GC strategy to enforce that in each plane group, only one GC is performed at one plane at a time (planes with page 2/4/8 in Fig. 10c), which brings a zero GC blocking overhead in plane-level parallelism. Finally, TTFLASH introduces a GC-tolerant flush (GTF) strategy to use the cap-backed random access memory (RAM) to cache the writes quickly and later flush them to flash pages at a GC-tolerant proper time. GTR and rotating GC ensure that the page eviction is not blocked by GC, which makes GTF to flush N–1 pages directly to flash media in every N pages per stripe. These strategies bring a nearly GC-free benefit and significantly reduce the average and P99 latencies. However, TTFLASH needs the support of intra-SSD copy-page command and dedicates one channel for parity.

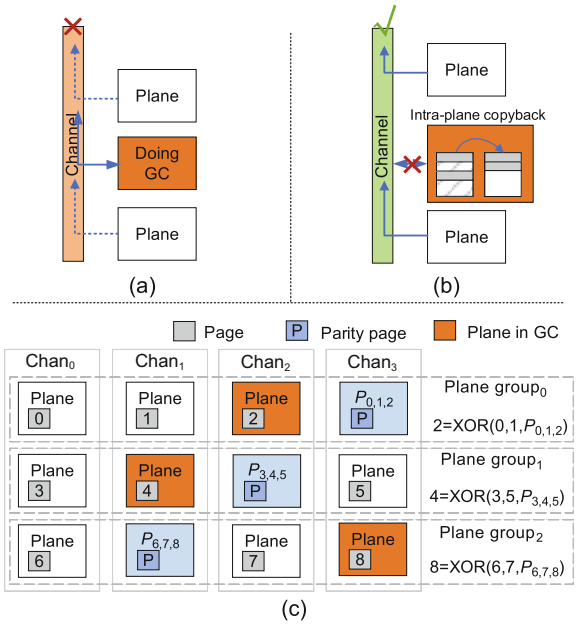

Fig. 10 Key strategies in TTFLASH: (a) channel-blocking GC; (b) plane-blocking GC; (c) GC-tolerant read/flush & rotating GC. GC: garbage collection; Chan: channel

6. GC copyback scheme

The GC scheme at the host side brings many benefits while reducing the device overloads. However, it may be frequently triggered under heavy write workloads and may severely take up host dynamic RAM (DRAM) bandwidth and space (Fig. 11a). To reduce this consumption, AOCBLK adopts a GC copyback scheme that deploys the GC buffer in the OCSSD device and exposes its address to the host side. By designing two copyback read/write commands, the host side needs only to send read/write commands, which takes nearly zero bandwidth on the device side; it is the OCSSD device that controls the valid data copying from flash memory to GC buffer or writing back from GC buffer to flash memory (Fig. 11b). Meanwhile, AOCBLK adopts a quota-based GC policy to coordinate the host and GC write speed. In this way, AOCBLK significantly reduces the impact of GC on host I/O performance. However, the in-device GC requires expensive device resources and may bring a great

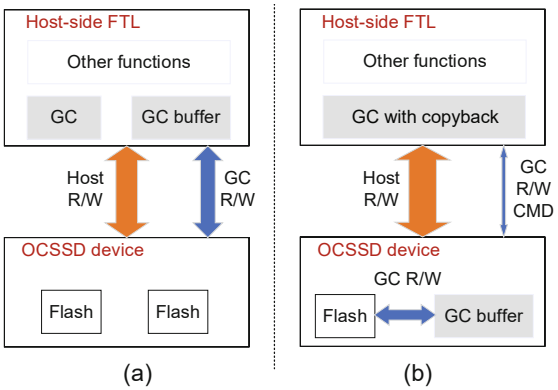

Fig. 11 The GC copyback scheme of AOCBLK: (a) GC without copyback; (b) GC with copyback. GC: garbage collection; FTL: flash translation layer; CMD: command; OCSSD: open-channel solid-state drive

burden to the device.

7. Postponed GC strategy

The GC in 3D NAND flash is different from that in the 2D case, which needs to pay more attention to thermal issues while keeping high efficiency. ThermAlloc (Wang Y et al., 2019) proposes a postponed GC strategy to achieve low heat production and high efficiency. It maps the logical block into three types of physical blocks (PBlk, RBlk, and BBlk) with enough distance, which brings the benefit that GC can be done in different physical areas in parallel, and that the heat can be scattered in different chips. It also designs an algorithm to postpone GC, which is triggered only when the RBlk runs out of space, and a temperature-aware strategy to decide where to place the write requests.

The works above describe five main methodologies to improve the performance of an individual OCSSD. The overall achieved main optimization goals can be seen in Table 4. In addition, it is meaningful and challenging to optimize the flash disk array in coordination using OCSSDs as the devices.

Existing works on traditional SSD-based RAID (redundant array of independent disks) systems have made a lot of optimizations for flash arrays, such as FusionRAID (Jiang et al., 2021), and achieved high consistency and low latency on commodity SSD arrays. However, due to the FTL isolation, it cannot fully take the advantages of flash disks and may cause unstable throughput and latency under heavy workloads. SOFA (Chiueh et al., 2014) proposes a log-structured flash array architecture that abstracts the FTL on top of the disk array management logic layer to reduce redundant software overhead; it also leaves a set of on-disk controllers to work with the host-side FTL (Fig. 12), so that all raw flash disks can be scheduled by the same FTL. Under the management of this global FTL, SOFA can reasonably schedule all the flash resources to prevent some disks from being worn out earlier than others and thus achieving global WL and load balance. The global FTL also allows SOFA to do global GC and small random writing aggregation while supporting I/O aggregation and distributed logging. All these optimizations significantly improve the performance and resource utilization of flash arrays.

IODA (Li et al., 2021) leverages the "PLM (predictable latency mode) windows" notion (NVM Express, Inc., 2023), which alternates the working time of flash devices between predictable and busy windows and further takes this concept into the design of flash arrays. As shown in Fig. 13, the working time is equally divided into the time window (TW). In the predictable TW, IODA guarantees that the device can do only reads or writes and no GCs will be triggered, which can ensure a strong predictable performance. In the busy TW, the device can perform GCs to ensure enough space for writes later. IODA alternates the busy/predictable TWs and guarantees that at most one device is in the busy status to execute the GC process and the others can still provide strong predictable performance. However, there is a strict constraint that the GC must be controllable, which needs OCSSDs to guarantee the deterministic GC and implement the predictable performance in

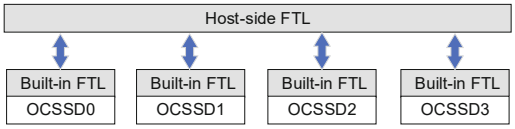

Fig. 12 FTL abstraction in SOFA. FTL: flash translation layer; OCSSD: open-channel solid-state drive

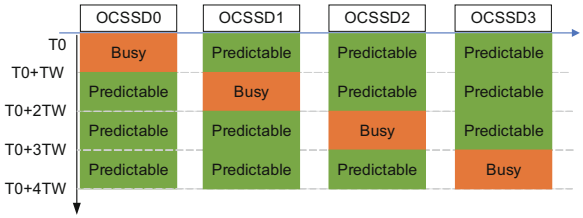

Fig. 13 Time window mechanism in IODA. OCSSD: open-channel solid-state drive

Table 4 Main goals achieved by various optimization methods

Optimization method	Throughput	Latency	Lifetime	Isolation	Resource utilization
Interface design	✓	✓	✓	✓	✓
FTL design	✓	✓	✓	✓	✓
Internal parallelism exploitation	✓	✓		✓	✓
I/O scheduling optimization	✓	✓	✓		
Garbage collection optimization	✓	✓	✓		

FTL: flash translation layer; I/O: input/output

the real flash array scenarios.

In practice, the above methods are often related to each other, since any individual optimization method cannot fully exert the performance advantages of OCSSDs, and researchers often adopt a combination of multiple methods to maximize the benefits. For example, LOCS adopts channel-level parallelism and I/O scheduling policies to fully leverage OCSSD's features, thus increasing the throughput by more than four times. Based on the exploitation of internal parallelism, ParaFS adopts a hot/cold data grouping layout, a parallelism-aware scheduling policy, and a foreground/background coordinated GC mechanism to achieve high throughput. We should fully leverage OCSSD's advantages to achieve optimization goals by rationally using these optimization methods coordinately according to system and workload features in domain-specific scenarios.

5 Challenges and future directions

5.1 Challenges

OCSSDs have great advantages over SSDs in terms of both performance and customizability due to their openness. However, the built-in FTL moving up to the host side and the exposure of the physical layout also bring some challenges, limiting the further research and application of OCSSDs.

1. Complexity and nonversatility of the FTL design

Although the host-side FTL can bring many advantages, such as providing rich functions, co-designing with host applications, and achieving higher performance, there are still three big challenges that should be faced. First, designing a host-side FTL is complicated; the developers must consider the design of the I/O interface, the behaviors of applications, and also the physical features of the raw flash memory to replace the already robust device-side FTL, which requires a high designing ability for them. Second, a host-side FTL requires high design, maintenance, and management prices, and may not be stable as the device-side FTL. Third, a host-side FTL is customized for host-side software; it is not designed for a generic purpose, which makes it hard to generalize to other software in practice. As far as we know, there is no universal host-side FTL for real business.

2. Lack of standards

There have been two versions of standards for OCSSDs, but sadly they both have failed. Until today, OCSSDs still have no universal standards in the storage industry (Picoli et al., 2020). On the other hand, the open-channel specification in the context of LightNVM simply defines the most universal part of OCSSDs; the other parts that may involve sensitive content and are inconvenient to be made public are implemented by manufacturers and vendors. These fragmented implementations introduce different methods, making it difficult to unify the standards. The lack of standards severely affects the development of OCSSDs, which is an urgent challenge to be addressed in the future.

3. Flexibility, security, and universality

Yang LH et al. (2019) believed that SSDs' internal physical layout should not be opened to external users, which would lead to unsecured access and hurt their availability and flexibility. OCSSDs are always co-designed with external applications to achieve higher performance, and thus it is difficult to directly apply them to other applications. Moreover, changes in the design of the upper software may cause co-changes of OCSSDs. All these make OCSSDs to have low flexibility and extensibility. On the other hand, for academic research, OCSSDs' openness is conducive to innovation and technology breakthroughs, but for engineering implementation, this openness may bring more design and maintenance risks, and a small mistake of the developers may cause significant security problems. Besides, due to the high complexity, low flexibility, and expensive nature of software development, OCSSDs are studied mainly in academia and enterprise-grade large-scale systems, mostly in domain-specific scenarios. Although LightNVM promotes the generic support of OCSSDs from the Linux system, and liblightnvm (González et al., 2016) provides a user-space library to facilitate developers' use of LightNVM, it still needs to make effort to make OCSSDs more universal in practice than in research.

5.2 Future development

In recent years, with the rise and development of NVMe, complex address transformations in OCSSDs could be avoided by zoned namespace (ZNS) (Han et al., 2021), which provides LBAs instead of PPAs, and the SSDs' internal parallelism is exploited

well. NVMe SSDs can achieve better performance such as lower latency and higher I/O throughput, and multi-stream SSDs can significantly improve GC efficiency and reduce WA. Although their performance is distant to that of OCSSDs, they earn a great application prospect for their standard and semi-open design, which can not only provide functional interfaces that need to be opened to external software but also hide access that degrades security and increases complexity. NVMe SSDs have been able to replace OCSSDs as underlying storage devices in many specific scenarios.

Despite facing many huge challenges, we should see that the open nature of OCSSDs makes them a better prospect and has attracted many scholars and developers to dedicate their works on them. OCSSDs allow upper-level systems and applications to customize their functions according to their own needs, and users in different scenarios can have more choices to leverage OCSSD's features for optimal performance and specific goals. On the other hand, the continuous improvement of general file systems such as LightNVM may enable the development of OCSSDs in more application scenarios. Developers should choose the most appropriate ones according to their specific needs.

OCSSDs should learn from NVMe SSDs and form their own standards and norms to better promote their research and application. Future research efforts, especially in domain-specific areas, need to take all factors, including standards, universality, products, and performance into consideration, which can further promote the development of OCSSDs.

6 Conclusions

By moving the FTL to the host side and co-designing with host-side software, OCSSDs enable SSDs' inherent characteristics and functionalities to be fully controlled, thereby achieving better performance such as higher throughput, lower latency, and better resource utilization. These advantages provide OCSSDs more opportunities in many domain-specific scenarios. Existing studies give us many insightful ideas such as rich FTL co-design at different software layers, full parallelism exploitation at different levels of parallelism, rational I/O scheduling policies to improve performance, and efficient GC mechanism to reduce performance overhead and long tail latency. These studies make OCSSDs far better than SSDs in many QoS- and latency-sensitive scenarios and provide them a positive reference and guidance for future research. However, some major challenges remain unresolved, and some research issues are still open for further investigation.

Contributors

Junchao CHEN designed the research. Junchao CHEN and Junyu WEI drafted the paper. Guangyan ZHANG helped organize the paper. Junchao CHEN and Guangyan ZHANG revised and finalized the paper.

Compliance with ethics guidelines

Guangyan ZHANG is a guest editor of this special feature, and he was not involved with the peer review process of this manuscript. Junchao CHEN, Guangyan ZHANG, and Junyu WEI declare that they have no conflict of interest.

References

Bhimani J, Yang JP, Yang ZY, et al., 2017. Enhancing SSDs with multi-stream: what? why? how? IEEE 36th Int Performance Computing and Communications Conf, p.1-2. https://doi.org/10.1109/PCCC.2017.8280493

Bjørling M, Gonzalez J, Bonnet P, 2017. LightNVM: the Linux open-channel SSD subsystem. 15th USENIX Conf on File and Storage Technologies, p.359-374.

Chen J, Wang Y, Zhou AC, et al., 2019. PATCH: process-variation-resilient space allocation for open-channel SSD with 3D flash. Design, Automation & Test in Europe Conf & Exhibition, p.216-221. https://doi.org/10.23919/DATE.2019.8715197

Chiueh TC, Tsao W, Sun HC, et al., 2014. Software orchestrated flash array. Proc Int Conf on Systems and Storage, p.1-11. https://doi.org/10.1145/2611354.2611360

Du YZ, Gu JH, Xiao ZZ, et al., 2020. SSW: a strictly sequential writing method for open-channel SSD. *J Syst Archit*, 109:101828. https://doi.org/10.1016/j.sysarc.2020.101828

González J, Bjørling M, 2017. Multi-tenant I/O isolation with open-channel SSDs. Non-Volatile Memories Workshop, p.1-2.

González J, Bjørling M, Lee S, et al., 2016. Application-driven flash translation layers on open-channel SSDs. Proc 7th Non-Volatile Memories Workshop, p.1-2.

Han K, Gwak H, Shin D, et al., 2021. ZNS+: advanced zoned namespace interface for supporting in-storage zone compaction. 15th USENIX Symp on Operating Systems Design and Implementation, p.147-162.

Hao MZ, Soundararajan G, Kenchammana-Hosekote DR, et al., 2016. The tail at store: a revelation from millions of hours of disk and SSD deployments. 14th USENIX Conf on File and Storage Technologies, p.263-276.

Hu Y, Jiang H, Feng D, et al., 2013. Exploring and exploiting the multilevel parallelism inside SSDs for improved

performance and endurance. *IEEE Trans Comput*, 62(6):1141-1155. https://doi.org/10.1109/TC.2012.60

Huang J, Badam A, Caulfield L, et al., 2017. FlashBlox: achieving both performance isolation and uniform lifetime for virtualized SSDs. 15th USENIX Conf on File and Storage Technologies, p.375-390.

Jiang TY, Zhang GY, Huang ZC, et al., 2021. FusionRAID: achieving consistent low latency for commodity SSD arrays. 19th USENIX Conf on File and Storage Technologies, p.355-370.

Kang W, Shin D, Yoo S, 2017. Reinforcement learning-assisted garbage collection to mitigate long-tail latency in SSD. *ACM Trans Embed Comput Syst*, 16(5s):134. https://doi.org/10.1145/3126537

Lee C, Sim D, Hwang JY, et al., 2015. F2FS: a new file system for flash storage. 13th USENIX Conf on File and Storage Technologies, p.273-286.

Lee S, Liu M, Jun SW, et al., 2016. Application-managed flash. 14th USENIX Conf on File and Storage Technologies, p.339-353.

Lee S, Han K, Shin D, 2019. Host-level workload-aware budget compensation I/O scheduling for open-channel SSDs. IEEE Non-Volatile Memory Systems and Applications Symp, p.1-2. https://doi.org/10.1109/NVMSA.2019.8863515

Li HC, Putra ML, Shi R, et al., 2021. IODA: a host/device co-design for strong predictability contract on modern flash storage. Proc ACM SIGOPS 28th Symp on Operating Systems Principles, p.263-279. https://doi.org/10.1145/3477132.3483573

Lu YY, Shu JW, Zheng WM, 2013. Extending the lifetime of flash-based storage through reducing write amplification from file systems. 11th USENIX Conf on File and Storage Technologies, p.257-270.

Lu YY, Shu JW, Zhang JC, 2019a. Mitigating synchronous I/O overhead in file systems on open-channel SSDs. *ACM Tran Stor*, 15(3):17. https://doi.org/10.1145/3319369

Lu YY, Zhang JC, Yang Z, et al., 2019b. OCStore: accelerating distributed object storage with open-channel SSDs. IEEE 39th Int Conf on Distributed Computing Systems, p.271-281. https://doi.org/10.1109/ICDCS.2019.00035

Marmol L, Sundararaman S, Talagala N, et al., 2015. NVMKV: a scalable, lightweight, FTL-aware key-value store. USENIX Annual Technical Conf, p.207-219.

Mathur A, Cao MM, Bhattacharya S, et al., 2007. The new Ext4 filesystem: current status and future plans. Proc Linux Symp, p.21-33.

NVM Express, Inc., 2023. NVM Express. https://nvmexpress.org/

Oh G, Ahn S, 2021. Implementation of memory efficient flash translation layer for open-channel SSDs. *Int J Adv Smart Converg*, 10(1):142-150. https://doi.org/10.7236/IJASC.2021.10.1.142

O'Neil P, Cheng E, Gawlick D, et al., 1996. The log-structured merge-tree (LSM-tree). *Acta Inform*, 33(4):351-385. https://doi.org/10.1007/s002360050048

Ouyang J, Lin SD, Jiang S, et al., 2014. SDF: software-defined flash for web-scale Internet storage systems. Proc 19th Int Conf on Architectural Support for Programming Languages and Operating Systems, p.471-484. https://doi.org/10.1145/2541940.2541959

Park SY, Seo E, Shin JY, et al., 2010. Exploiting internal parallelism of flash-based SSDs. *IEEE Comput Archit Lett*, 9(1):9-12. https://doi.org/10.1109/L-CA.2010.3

Picoli IL, Pasco CV, Jónsson BÞ, et al., 2017. uFLIP-OC: understanding flash I/O patterns on open-channel solid-state drives. Proc 8th Asia-Pacific Workshop on Systems, Article 20. https://doi.org/10.1145/3124680.3124741

Picoli IL, Hedam N, Bonnet P, et al., 2020. Open-channel SSD (what is it good for). 10th Conf on Innovative Data Systems Research, p.1-8.

Qin HW, Feng D, Tong W, et al., 2019. QBLK: towards fully exploiting the parallelism of open-channel SSDs. Design, Automation & Test in Europe Conf Exhibition, p.1064-1069. https://doi.org/10.23919/DATE.2019.8715049

Qin HW, Feng D, Tong W, et al., 2021a. Better atomic writes by exposing the flash out-of-band area to file systems. Proc 22nd ACM SIGPLAN/SIGBED Int Conf on Languages, Compilers, and Tools for Embedded Systems, p.12-23. https://doi.org/10.1145/3461648.3463843

Qin HW, Feng D, Tong W, et al., 2021b. QBLKe: host-side flash translation layer management for open-channel SSDs. *J Syst Archit*, 119:102233. https://doi.org/10.1016/j.sysarc.2021.102233

Qiu YH, Yin WB, Wang LL, 2021. A high-performance open-channel open-way NAND flash controller architecture. 31st Int Conf on Field-Programmable Logic and Applications, p.91-98. https://doi.org/10.1109/FPL53798.2021.00023

Reidys B, Sun JH, Badam A, et al., 2022. BlockFlex: enabling storage harvesting with software-defined flash in modern cloud platforms. 16th USENIX Symp on Operating Systems Design and Implementation, p.17-33.

Rodeh O, Bacik J, Mason C, 2013. BTRFS: the Linux B-tree filesystem. *ACM Trans Stor*, 9(3):9. https://doi.org/10.1145/2501620.2501623

Rosenblum M, Ousterhout JK, 1991. The design and implementation of a log-structured file system. Proc 13th ACM Symp on Operating Systems Principles, p.1-15. https://doi.org/10.1145/121132.121137

Shen ZY, Chen F, Jia YC, et al., 2017. DIDACache: a deep integration of device and application for flash based key-value caching. 15th USENIX Conf on File and Storage Technologies, p.391-405.

Shen ZY, Chen F, Yadgar G, et al., 2022. Prism-SSD: a flexible storage interface for SSDs. *IEEE Trans Comput Aided Des Integr Circ Syst*, 41(4):882-896. https://doi.org/10.1109/TCAD.2021.3072326

Son S, Ahn S, 2021. Optimizing garbage collection overhead of host-level flash translation layer for journaling filesystems. *Int J Int Broadcast Commun*, 13(2):27-35. https://doi.org/10.7236/IJIBC.2021.13.2.27

The Open-Channel SSD Community, 2023. LightNVM Open Channel Specification. https://openchannelssd.readthedocs.io/en/latest/specification

Wang HT, Li ZH, Zhang X, et al., 2018. OC-Cache: an open-channel SSD based cache for multi-tenant systems. IEEE 37th Int Performance Computing and Communications Conf, p.1-6. https://doi.org/10.1109/PCCC.2018.8711079

Wang P, Sun GY, Jiang S, et al., 2014. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. Proc 9th European Conf on Computer Systems, Article 16. https://doi.org/10.1145/2592798.2592804

Wang Y, Zhang MX, Yang X, et al., 2019. A thermal-aware physical space reallocation for open-channel SSD with 3-D flash memory. *IEEE Trans Comput Aided Des Integr Circ Syst*, 38(4):617-627. https://doi.org/10.1109/TCAD.2018.2821442

Wang Y, Huang JF, Chen J, et al., 2022. PVSensing: a process-variation-aware space allocation strategy for 3D NAND flash memory. *IEEE Trans Comput Aided Des Integr Circ Syst*, 41(5):1302-1315. https://doi.org/10.1109/TCAD.2021.3091957

Weil SA, Brandt SA, Miller EL, et al., 2006. Ceph: a scalable, high-performance distributed file system. Proc 7th Symp on Operating Systems Design and Implementation, p.307-320.

Wu SM, Lin KH, Chang LP, 2021. Integrating LSM trees with multichip flash translation layer for write-efficient KVSSDs. *IEEE Trans Comput Aided Des Integr Circ Syst*, 40(1):87-100. https://doi.org/10.1109/TCAD.2020.2987781

Yan SQ, Li HC, Hao MZ, et al., 2017. Tiny-tail flash: near-perfect elimination of garbage collection tail latencies in NAND SSDs. 15th USENIX Conf on File and Storage Technologies, p.15-28.

Yang JP, Plasson N, Gillis G, et al., 2014. Don't stack your log on my log. 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads, p.1-10.

Yang LH, Zhang T, Fan YL, et al., 2019. Exploiting internal parallelism of SSD for hash join. *Chin J Electron*, 28(5):889-898. https://doi.org/10.1049/cje.2019.06.009

Zhang JC, Shu JW, Lu YY, 2016. ParaFS: a log-structured file system to exploit the internal parallelism of flash devices. USENIX Annual Technical Conf, p.87-100.

Zhang JC, Lu YY, Shu JW, et al., 2017. FlashKV: accelerating KV performance with open-channel SSDs. *ACM Trans Embed Comput Syst*, 16(5s):1-19. https://doi.org/10.1145/3126545

Zhang XY, Zhu F, Li S, et al., 2021. Optimizing performance for open-channel SSDs in cloud storage system. IEEE Int Parallel and Distributed Processing Symp, p.902-911. https://doi.org/10.1109/IPDPS49936.2021.00099