# Storage Abstractions for SSDs: The Past, Present, and Future

XIANGQUN ZHANG, Syracuse University, Syracuse, United States
JANKI BHIMANI, Florida International University, Miami, United States
SHUYI PEI, Samsung Semiconductor Inc, San Jose, United States
EUNJI LEE, Soongsil University, Seoul, Korea (the Republic of)
SUNGJIN LEE, DGIST, Daegu, Korea (the Republic of)
YOON JAE SEONG, FADU Inc., Seoul, Korea (the Republic of)
EUI JIN KIM, FADU Inc., Seoul, Korea (the Republic of)
CHANGHO CHOI, Samsung Semiconductor Inc, San Jose, United States
EYEE HYUN NAM, FADU Inc., Seoul, Korea (the Republic of)
JONGMOO CHOI, Dankook University, Yongin, Korea (the Republic of)
BRYAN S. KIM, Syracuse University, Syracuse, United States

This article traces the evolution of SSD (solid-state drive) interfaces, examining the transition from the block storage paradigm inherited from hard disk drives to SSD-specific standards customized to flash memory. Early SSDs conformed to the block abstraction for compatibility with the existing software storage stack, but studies and deployments show that this limits the performance potential for SSDs. As a result, new SSD-specific interface standards emerged to not only capitalize on the low latency and abundant internal parallelism of SSDs, but also include new command sets that diverge from the longstanding block abstraction.

We first describe flash memory technology in the context of the block storage abstraction and the components within an SSD that provide the block storage illusion. We then describe the genealogy and relationships among academic research and industry standardization efforts for SSDs, along with some of their rise and fall in popularity. We classify these works into four evolving branches: (1) extending block abstraction with host-SSD hints/directives; (2) enhancing host-level control over SSDs; (3) offloading host-level management to SSDs; and (4) making SSDs byte-addressable. By dissecting these trajectories, the article also sheds light on the emerging challenges and opportunities, providing a roadmap for future research and development in SSD technologies.

CCS Concepts: • **Computer systems organization** → *Firmware*; • **Software and its engineering** → **Secondary storage**; *File systems management*; • **Information systems** → **Flash memory;**

This paper is supported in part by NSF grants 2008453, 2323100, 2338457, and 2402328.

Authors' Contact Information: Xiangqun Zhang, Syracuse University, Syracuse, New York, United States; e-mail: xzhang84@syr.edu; Janki Bhimani, Florida International University, Miami, Florida, United States; e-mail: jbhimani@fiu.edu; Shuyi Pei, Samsung Semiconductor Inc, San Jose, California, United States; e-mail: shuyi.pei@samsung.com; Eunji Lee, Soongsil University, Seoul, Korea (the Republic of); e-mail: ejlee@ssu.ac.kr; Sungjin Lee, DGIST, Daegu, Korea (the Republic of); e-mail: sungjin.lee@dgist.ac.kr; Yoon Jae Seong, FADU Inc., Seoul, Korea (the Republic of); e-mail: yjseong@fadutec.com; Eui Jin Kim, FADU Inc., Seoul, Korea (the Republic of); e-mail: euijin.kim@fadutec.com; Changho Choi, Samsung Semiconductor Inc, San Jose, California, United States; e-mail: changho.c@samsung.com; Eyee Hyun Nam, FADU Inc., Seoul, Korea (the Republic of); e-mail: ehnam@fadutec.com; Jongmoo Choi, Dankook University, Yongin, Gyeonggi, Korea (the Republic of); e-mail: choijm@dankook.ac.kr; Bryan S. Kim, Syracuse University, Syracuse, New York, United States; e-mail: bkim01@syr.edu.


ACM 1553-3077/2025/01-ART2
https://doi.org/10.1145/3708992

Additional Key Words and Phrases: SSD, garbage collection, flash translation layer, data placement scheme, zoned namespace, multi-stream, flexible data placement, computational storage, byte-addressable SSD

**ACM Reference Format:**
Xiangqun Zhang, Janki Bhimani, Shuyi Pei, Eunji Lee, Sungjin Lee, Yoon Jae Seong, Eui Jin Kim, Changho Choi, Eyee Hyun Nam, Jongmoo Choi, and Bryan S. Kim. 2025. Storage Abstractions for SSDs: The Past, Present, and Future. *ACM Trans. Storage* 21, 1, Article 2 (January 2025), 44 pages. https://doi.org/10.1145/3708992

## 1 Introduction

Modern computers use abstractions for interoperability between the host and its peripherals [25]. Storage, as the basis of the memory hierarchy, also requires an abstraction layer for host-storage communications. This abstraction layer allows the host to use storage devices without regard to its underlying implementation. Over the past decades, the fundamental design of the storage abstraction has remained relatively consistent: The device exposes an address range, and the host system issues read/write requests within that range. Subsequently, the storage device executes the requests and returns the results back to the host. This simple interface between the storage device and the host system is sufficient for data storage purposes. It aged well from floppy to hard drive and finally was passed down to the SSD.

However, new developments and findings in the field of storage device demand enhancements to the existing storage abstraction. SSDs are fundamentally different from magnetic disks such as floppies and hard disk drives. First, by using NAND flash, SSDs achieve high throughput by eliminating moving parts and exploiting internal parallelism. However, NAND flash does not support in-place updates, which means that garbage collection is required. Although the existing storage abstraction works on SSDs, it does not provide a method to reduce garbage collection overhead with host-SSD coordination. Second, the increasing speed of SSDs shifted the latency source in the storage stack. The storage device is not dominating the latency anymore; instead, the host also contributes about half of the total latency [170]. The internal bandwidth of the SSD is also larger than the external bandwidth between the host and the SSDs [136], which means that data transfer between the host and the SSDs poses a major limitation of utilizing SSDs to their full potential. Last but not least, the recent emergence of **Compute Express Link (CXL)** allows researchers to reimagine integrating SSDs into the memory hierarchy by directly using SSDs as part of the main memory. However, the main memory is byte-addressable while the traditional storage abstraction sees secondary storage in units of sectors [25]. Together, these three points warrant new enhancements to the existing storage abstraction.

Thankfully, both academia and industry are well aware of limitations in the traditional storage abstraction. Many new features, enhancements, and host-device co-designs have been created to address the shortcomings of the existing storage abstraction. In this article, we survey the SSD device abstraction and enhancements, exploring their backgrounds and designs that enable SSDs to better communicate with the host. We will also examine their relationships, applications, standardization efforts, and fluctuations in popularity over time. We categorize these abstraction enhancements into four categories:

(1) extending block abstraction with host-SSD hints/directives. This includes TRIM, Multi-stream, and **Flexible Data Placement (FDP)**;
(2) enhancing host-level control over SSDs. This includes **Open-channel (OC)** SSD and **Zoned Namespaces (ZNS)** SSD;

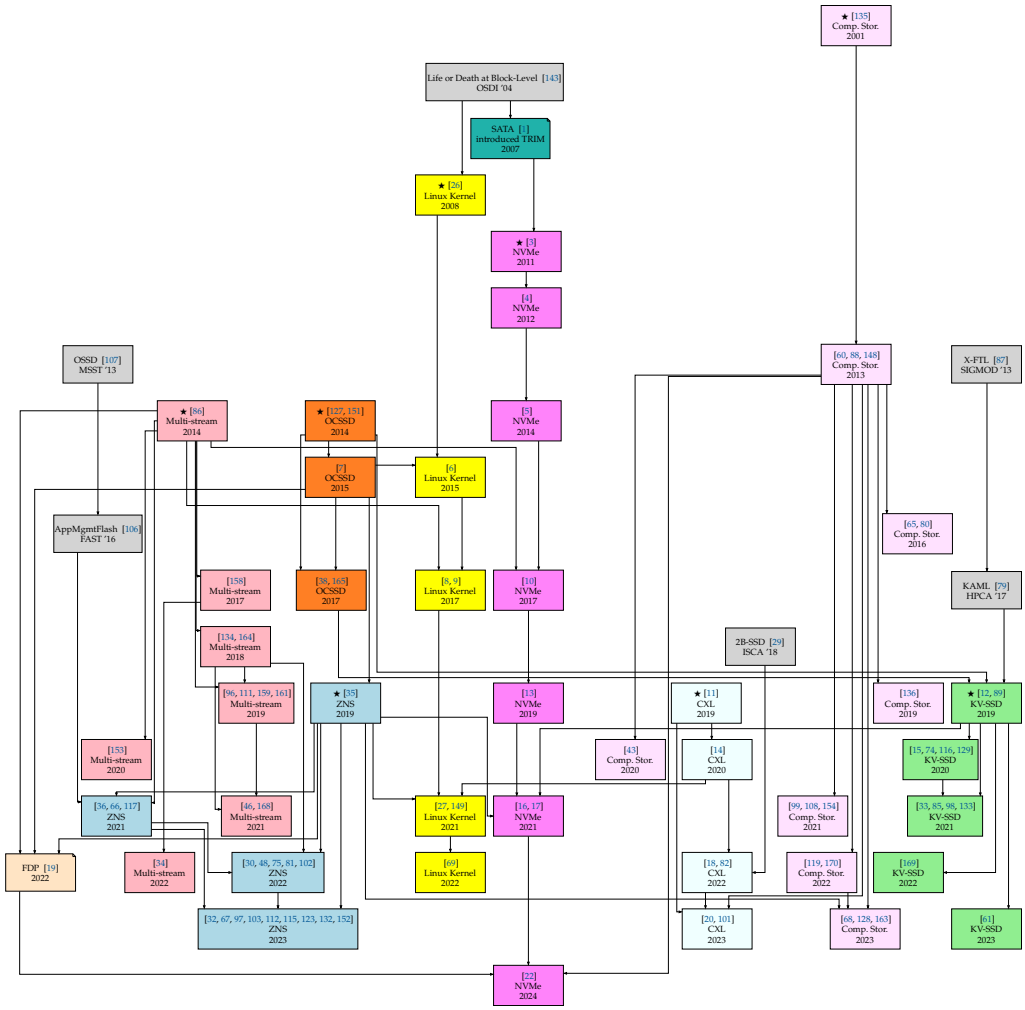


Fig. 1. The genealogy tree. Papers published in the same year are placed on the same row. Different colors indicate different categories (Gray indicates no specific category). ★ indicates the root of a given category. A detailed version showing every paper as an independent node can be found in the Appendix as Figure 12.

(3) offloading host-level management to SSDs. This includes **Key-value (KV)** SSD and computational storage;
(4) making SSDs byte-addressable, along with their use in CXL,

which we present along with their standardization history in Figure 1. Last, we will conclude our article with our view on the possible future directions for extensions to storage abstractions.

Our main contributions in this survey include:

(1) A comprehensive overview of enhancements to the traditional storage abstraction.
(2) Categorization of different enhancements based on their characteristics.
(3) Visualization of the relationship between different enhancements and their related work.
(4) Identification of directions and challenges for current and future enhancements to storage abstractions.

The rest of this survey is organized as follows: Section 2 provides a background on SSD and storage abstractions. Section 3 through Section 6 discuss storage abstraction enhancements, including those that providing extra hints/directives to SSD (Section 3), moving SSD responsibilities to the host (Section 4) and vice versa (Section 5), and making SSD byte-addressable (Section 6). Section 7 is our view on the future development of SSD storage abstractions, and Section 8 concludes this survey.

## 2 Abstraction Layers for Host-SSD Communication

Making SSDs work on the host computer is not an easy task: Multiple layers are involved, and they should seamlessly coordinate with each other to form a storage stack with high throughput and low latency. We abstract the storage stack into three layers: The operating system serves as the top layer, the communication protocol as the middle layer, and the SSD itself as the bottom layer. The operating system provides storage abstractions to the applications running on top of it, so the applications do not need to be concerned about the underlying storage details when running. The communication protocol is in charge of the communication between the operating system and the SSD, which ensures the interoperability between different hosts and SSDs. When receiving requests from the host, The SSD firmware has to perform operations fulfilling those requests. In this section, we will discuss different abstraction layers that coordinate the host and SSD along with their evolutions layer-by-layer.

### 2.1 Linux Block Layer (`bio`)

Storage devices come in different shapes and sizes, ranging from legacy tapes and floppy disks to contemporary hard disk drives and solid state drives, each characterized by its distinct underlying mechanisms. Despite this diversity, most mass storage devices expose their storage capacity to the operating system and support two primary operations: *read* and *write*. The operating system should be able to execute read and write requests within the specified address space regardless of the actual type and implementation of the device. Linux addresses this requirement through the implementation of the `bio` layer. The `bio` layer serves as an interface that bridges the gap between the operating system and various types of mass storage devices. By providing a consistent interface, irrespective of the device's type or implementation, the `bio` layer ensures interoperability [42]. This enables host applications to issue I/O requests independent of the storage device's implementation details. Despite the overarching purpose of the `bio` layer, it is crucial for the operating system to acknowledge the inherent diversity among storage devices in terms of their physical attributes and internal mechanisms. Consequently, the `bio` layer must exhibit adaptability to cater to the unique specifications of different devices.

To take advantage of all the features that a storage device provides, the `bio` layer has been extended for additional operations and fields tailored to numerous features of different storage devices. Listing 1 shows the evolution of the `bio` layer over the years, with the initial version of the `bio` layer released in 2001 on the left and a later version released in 2021 on the right. The initial `bio` layer was designed with two operations only: *read* and *write*. However, to pursue improved I/O performance for a variety of devices, the `bio` layer has undergone many enhancements. Notable additions include the discard operation (also known as `TRIM`) [143], introduced to the `bio` layer to provide an optional hint for improved I/O performance [26]. Other enhancements, including Multi-stream and Zoned Namespaces, are also added to the `bio` layer [9, 27]. The `bio` layer also underwent a major redesign for a multi-queue design named `blk-mq`, which scales with the number of host CPU cores and addresses the performance bottleneck at the `bio` layer due to the higher performance of the storage device that comes with the era of SSDs [37]. We present an overview of storage enhancement changes to the Linux `bio` layer using yellow boxes in Figure 1. In summary,

```c
// include/linux/bio.h, v2.5.1
struct bio {
  // ...
  /* bottom bits READ/WRITE, top bits priority */
  unsigned long bi_rw;
  // ...
}


// ...
/*
 * bio bi_rw flags
 *
 * bit 0 -- read (not set) or write (set)
 * bit 1 -- rw-ahead when set
 * bit 2 -- barrier
 */
#define BIO_RW          0
#define BIO_RW_AHEAD    1
#define BIO_RW_BARRIER  2
// (END OF THE BIO_* DEFINITION)
// ...
```

```c
// include/linux/blk_types.h, v5.15
struct bio {
  // ...
  /* bottom bits REQ_OP, top bits req_flags. */
  unsigned int bi_opf;
  // ...
  unsigned short bi_write_hint; // Multi-stream
  // ...
}
// ...
enum req_opf {
  REQ_OP_READ           = 0,
  REQ_OP_WRITE          = 1,
  REQ_OP_FLUSH          = 2,
  REQ_OP_DISCARD        = 3,
  REQ_OP_SECURE_ERASE   = 5,
  REQ_OP_WRITE_SAME     = 7,
  REQ_OP_WRITE_ZEROES   = 9,
  REQ_OP_ZONE_OPEN      = 10,
  REQ_OP_ZONE_CLOSE     = 11,
  REQ_OP_ZONE_FINISH    = 12,
  REQ_OP_ZONE_APPEND    = 13,
  REQ_OP_ZONE_RESET     = 15,
  REQ_OP_ZONE_RESET_ALL = 17,
  REQ_OP_DRV_IN         = 34,
  REQ_OP_DRV_OUT        = 35,
  REQ_OP_LAST,
};
// ...
```

Listing 1. The evolution of the `bio` layer from the initial version from Kernel v2.5.1 in 2001 (left) and Kernel v5.15 in 2021 (right) with discard, Multi-stream, and Zoned Namespaces support. More operations and abstraction enhancements, as defined by `BIO_*` and `REQ_OP_*`, have been supported over the years.

the `bio` layer functions as the fundamental interface for all mass storage devices for the Linux kernel since v2.5.1, offering both a universal and flexible framework to adapt to the general and distinct characteristics of different storage devices, enabling enhancements to storage abstractions.

### 2.2 Communication Protocols

The host and the SSD must "speak" the same "language" to make them interoperable. Protocols, such as SATA and NVMe, are created to enable the host and the SSD to understand requests and responses from each other. **Serial AT Attachment (SATA)** is a bus interface developed to connect mass storage devices to a host system. Introduced in 2001, it serves as a high-speed serial link replacement **for parallel ATA (PATA)** attachment of mass storage devices [155]. SATA quickly became the dominant bus interface for mass storage devices [2], particularly hard disk drives, due to its improved speed compared to PATA. During the past two decades, SATA has undergone three major versions: SATA 1.0, 2.0, and 3.0 [121]. The primary enhancement in each major version focuses on the data transfer speed. While minor versions like SATA 3.1 exist, they do not focus on improving transfer speed [121]. As the SATA workgroup only planned for three major versions up to SATA 3 with a maximum transfer speed of 6 Gb/s [155], an alternative method of connecting SSDs to the host computer is imperative to exceed the speed limitations imposed by SATA.

**NVM Express**, commonly known as **NVMe**, serves as a communication interface bridging the connection between a host system and a non-volatile memory subsystem. By utilizing PCIe as its underlying bus interface, NVMe can achieve about 1 GB/s throughput even using only one single PCIe 3.0 lane with the ability to scale up to a maximum data transfer speed of nearly 16 GB/s when utilizing all 16 lanes when it was first introduced in 2011 [3]. Consequently, NVMe with PCIe has emerged as the predominant interface for SSDs. NVMe has undergone iterations over the years like SATA, incorporating clarifications and introducing new features [3, 13, 17] with the expansion

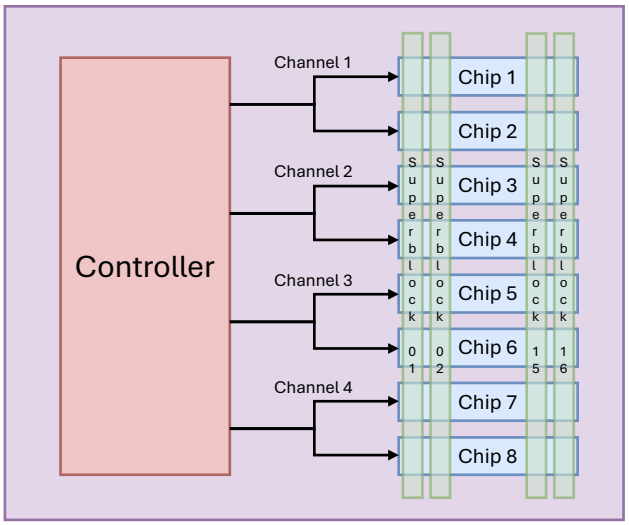


Fig. 2. The overall structure of a traditional SSD.

to include support for physical layers beyond PCIe [125, 126]. Given NVMe's established status as the *de facto* standard for SSDs, any proposed enhancements to SSD storage abstractions must be ratified and accepted into the NVMe standard for universal recognition. Figure 1 illustrates the ratification of new storage abstraction enhancements into SATA (in sea-green boxes) and NVMe (in orchid-colored boxes), which we will also discuss in future sections.

### 2.3 Solid State Drive (SSD)

Upon receiving a SATA/NVMe request from the host, the SSD handles the request according to the type of operation. SSD leverages NAND flash chips as its underlying storage medium, and a read/write request can be striped and handled by several flash chips concurrently, resulting in high throughput by exploiting internal parallelism [30]. The widespread adoption of SSDs led to the evolution of their upper layers, including the creation of `bio` `blk-mq` and NVMe.

However, NAND flash chips do not support in-place updates. The smallest unit of write is a page, whereas the smallest reclaim unit is a multi-page erase block. In practice, SSDs exploit internal parallelism by grouping erase blocks from each chip into a superblock, as shown in Figure 2. When handling a write request, the SSD sequentially places incoming data pages chip-by-chip in a superblock. A new superblock will be chosen if and only if the previous superblock is full. Updating existing data requires placing new data pages elsewhere on the SSD and invalidating corresponding old data pages. To reclaim space from invalid data, the SSD has to perform **garbage collection (GC)**. A superblock will be chosen as the victim: Any valid data in the victim block will be relocated. This process causes **write amplification (WA)**, which can be measured by the **write amplification factor (WAF)**. The SSD also needs to erase the victim block after relocating all valid pages, which is orders of magnitude slower than reading or writing a single page [54]. In summary, internal data movement and block erase together contribute to the long SSD tail latency [157].

Therefore, reducing GC overhead has been a central focus of SSD research, but the lack of predictability hinders optimization. Traditional storage abstraction does not provide direct control over GC triggering, awareness of GC events, and data placement to reduce GC overhead. Consequently, enhancements to storage abstractions are imperative for SSDs to collaborate

effectively with hosts and achieve their full potential. Some examples on this front include Multi-stream, Zoned Namespaces, and Flexible Data Placement. By exchanging more information between the SSD and the host, garbage collection overhead can be significantly reduced [19, 36, 86].

In summary, SSD dominates the current storage market, thanks to its performance by leveraging multiple NAND flash chips. However, NAND flash is also a double-edged sword, which requires data relocation and block erase during the garbage collection process, since NAND flash does not support in-place updates. To overcome the performance loss due to garbage collection, the host and the SSD should communicate and coordinate to reduce the overhead caused by garbage collection.

### 2.4 Storage Stack Summary

We identified the three layers of the storage stack in this section: the operating system layer (most importantly `bio`), the communication protocol via a selected physical layer, and the SSD itself. Any of these three layers can become a bottleneck for the whole storage stack. Therefore, it is crucial to identify and remove performance bottlenecks from all three layers.

Since the communication protocol, being the middle layer, is built upon the underlying industry-standard physical bus (e.g., NVMe over PCIe), prior works primarily focused on the host layer and the SSD layer when introducing new performance improvements. By reducing the data returned to the host after a request, the time for data transfer can be reduced, because the storage stack on the host can be heavy, which includes the application, the filesystem, the `bio` layer, and the device driver. However, the SSD releases more bandwidth and internal CPU resources to handle host requests by reducing the frequency of garbage collection. The communication protocol depends on the other two layers and would be modified if the host and the SSD require additional fields to interchange information. We will discuss the various enhancements to storage abstractions yielding better performance by achieving the goals above in the following sections.

#### Enhancements to Storage Abstractions

We categorize storage abstraction enhancements into four categories:

(1) Extending block abstraction with host-SSD hints/directives.
(2) Enhancing host-level control over SSDs.
(3) Offloading host-level management to SSDs.
(4) Making SSDs byte-addressable.

These enhancements improve performance and/or provide new possibilities for SSDs from different perspectives, which includes garbage collection reduction, data movement reduction, and/or host-SSD co-design.

## 3 Host-SSD Hints/Directives

In this section, we focus on the hints and directives provided to the SSD from the host. We define *hint* as an optional data entry field that the SSD can *optionally* utilize or ignore. The SSD can utilize the given hints for hopefully better performance, but it can also safely ignore such hints if the SSD does not know how to utilize such hints. For example, an SSD that supports `TRIM` hint can sometimes ignore the hint if the given range is too small [147, 150]. However, we define *directive* as a command that SSD *should* follow. The SSD is expected to perform the operation by utilizing the information sent by the host, as every existing feature built upon directives is designed to follow the orders given by the directives [19, 86, 120].[1] To summarize, hints are more optional than

[1]Developers may sometimes call the directive values (e.g., Multi-stream stream IDs) to be passed as "write hints" [105, 110]. We will also call these values "write hints" or "lifetime hints" in this article to avoid any confusion.

directives: A device can safely ignore hints but is expected to follow directives to the maximum extent.

### 3.1 Discard/TRIM

First proposed in the paper by Sivathanu et al. [143], the design of the TRIM operation is simple but effective: The host system tells the SSD using the dataset management command [17, 121] about which logical address now contains invalid data due to file deletion. This essentially reduces the number of valid pages relocated in the SSD in the garbage collection process. Otherwise, the SSD will not know that a logical address range contains deleted data and will relocate the corresponding physical pages during garbage collection, causing extra GC overhead [1]. Since the TRIM command indicates the invalidity of the data, it is also used to securely erase data by physically removing the data from the SSD immediately after the data is invalidated [55].

Unlike most other categories in this survey, there are very limited design choices and extensions for TRIM; one example, though, is the frequency to send TRIM requests to limit the number of I/O requests [32]. Only a few prior works focused on TRIM policies and implications [73, 83, 93, 104]. Nevertheless, the effectiveness and (relative) simplicity of TRIM [86] encouraged most modern operating systems and storage protocols to adopt TRIM as a part of them. This includes Microsoft Windows since Windows 7 [122], Linux since kernel version 2.6.28-rc1 [26], macOS since Lion [144], SATA since 2007 [1], and NVMe since version 1.0 [3].

Despite the simplicity of TRIM, it took years to improve TRIM policies in the Linux kernel. The most intuitive time to issue TRIM requests is right after when the filesystem knows a location is freed up due to deletion and overwrite. This is called online TRIM (also known as synchronous TRIM [50, 147]). However, early SATA protocol does not support queued TRIM requests. To issue a TRIM request, the operating system has to ensure that the current I/O queue is empty. This can cause significant performance degradation, since TRIM is blocked by and can also block other I/O requests [45, 50, 150]. To minimize the performance impact caused by TRIM, the operating system needs to perform TRIM when there are no outstanding I/O requests in the I/O queue. One approach is to keep track of all discardable segments on the host side and send a batch of TRIM requests when there are no outstanding I/O requests (e.g., once a week, when the system is idle). This approach is called *batched* TRIM [52, 92, 114]. Due to its simplicity, it is widely supported by different filesystems such as ext3/4 [51] and F2FS [92].

However, batched TRIM is not real-time, as the user is responsible for choosing an adequate frequency (e.g., once per week). If the SSD is near full, then TRIM helps on SSD performance, since it frees more internal space for data relocation after garbage collection, which reduces the possibility of triggering GC every time after a write [62, 73]. If the user is unaware that the next batched TRIM is scheduled in the far future when the SSD is near full due to excessive use, then the efficiency of the SSD can be impacted due to excessive GCs. With the introduction of queueable TRIM in SATA 3.1 [139] and later in NVMe [17], TRIM requests can also be sent into queues without blocking or being blocked by other I/O requests. This means online TRIM is finally becoming practical, and the operating system can issue online TRIM requests to SSDs without blocking other I/O requests.

Although online TRIM provides real-time information to the SSD about the freed space, this could severely impact SSD performance, since it takes a long time to handle TRIM requests [62, 84]. To mitigate this issue, some Linux filesystems now support *asynchronous TRIM* so TRIM requests will only be issued to SSDs after there are enough ranges of discardable spaces [145]. Unlike batched TRIM, asynchronous TRIM is more flexible, because it is designed to send TRIM requests more frequently [146]. It also prevents excessive, fragmented TRIM requests that can sometimes be ignored by SSDs [145] or cause SSD performance degradation [147]. With its balance between

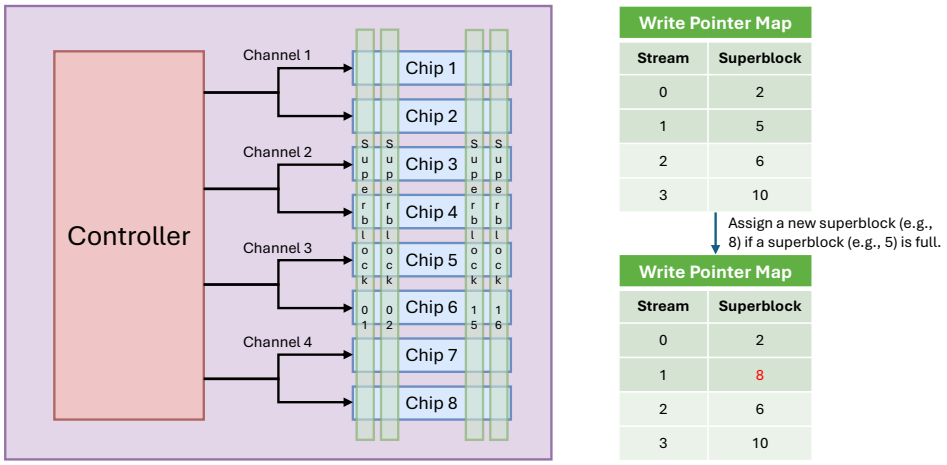


Fig. 3. The overall structure of a Multi-stream SSD.

TRIM frequency and efficiency, asynchronous TRIM has recently become the default TRIM choice for filesystems like Btrfs [147], marking the most recent improvement for TRIM.

## 3.2 Multi-stream

*3.2.1 From Manual to Automatic.* Multi-stream [86] is a directive that allows hosts to inform SSD of the physical placement preference of different data. It has been observed that data with similar characteristics may be invalidated at the same time; by grouping data with similar characteristics into the same superblock, data in the same superblock tend to be bimodal, i.e., being all valid or all invalid. If all picked garbage collection victims contain mostly or entirely invalid data, then the write amplification can be kept low. Data attached with different stream IDs based on host-calculated *write hints* [9, 110] will be written to different superblocks as shown in Figure 3. To summarize, the host calculates write hints based on the data characteristics and uses these write hints as stream ID directives when writing the data to the SSD [13]. When the superblock associated with the stream ID is full, another free superblock will be assigned to this stream ID to ensure different superblocks will not mix data with different stream IDs. Multi-stream thus provides the bridge for the host to communicate with the SSD regarding the data characteristics known by the host. A series of works has spun off from the original Multi-stream paper, providing different enhancements, and Multi-stream was ratified into NVMe 1.3 [10] and supported by the Linux kernel since v4.13-rc1 [9]. Figure 4 shows the relationship between different Multi-stream related papers. Table 1 summarizes different Multi-stream related papers along with some of their most important characteristics.

One critical shortcoming of the original Multi-stream paper is that it requires manual assignment of the stream IDs. If an application would like to exploit the benefit provided by Multi-stream, then it has to be rewritten so the stream ID can be assigned to each write request. For example, an update to RocksDB was necessary to fully exploit the benefits provided by Multi-stream [110]. The requirement of manually assigning stream IDs limited the adoption of Multi-stream. To mitigate this problem, Yang et al. proposed AutoStream [158] for automatic assignment of stream IDs. It is the first paper to enable automatic assignment of stream IDs, paving the way for subsequent papers in this field. AutoStream is implemented on the device driver level, since some applications may bypass the block I/O layer, and the SSD may have limited computational resources for stream ID calculation. AutoStream provided two different algorithms, **multi-queue (MQ)** and **sequentiality, frequency, and recency (SFR)**.

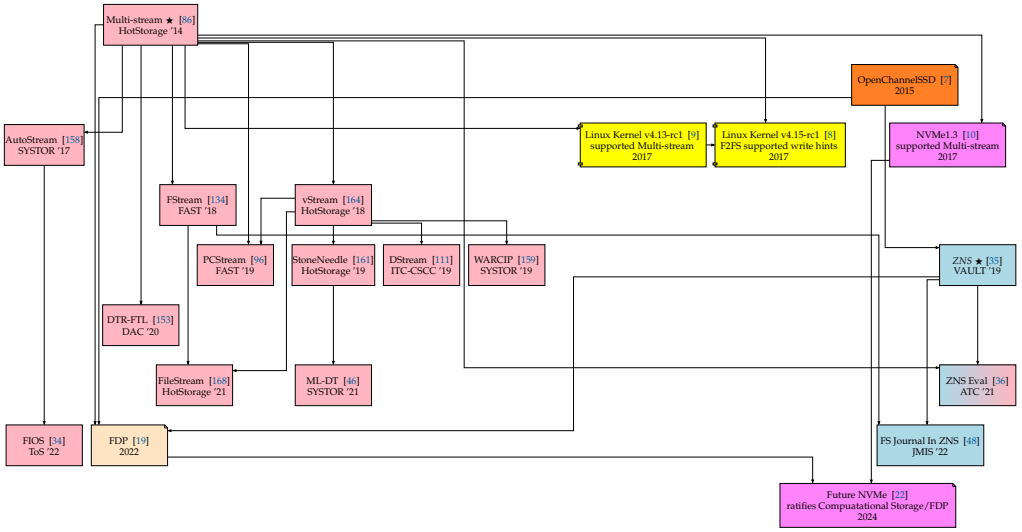


Fig. 4. Multi-stream related works.

Table 1. A Summary of Multi-stream and Its Related Works

| Name | Location | On/Offline | Algorithm | Description |
|---|---|---|---|---|
| Multi-stream [86] | Host | N/A | Manual | Provides an interface to tag data with a stream ID. Requires manual assignment of stream IDs. |
| AutoStream [158] | Host | Online | MQ, SFR | Assigns stream ID using access frequency, recency, and sequentiality automatically. |
| FStream [134] | Host | Online | N/A | Provides different streams for different filesystem metadata and different files. |
| vStream [164] | SSD | Online | K-means | Groups many manually assigned virtual streams to a few physical streams. |
| PCStream [96] | Host | Online | K-means | Assigns stream ID based on program context (PC) automatically, extended from their HotStorage work [95]. |
| DStream [111] | SSD | Online | K-means | Groups hot and cold data streams into a dynamic number of physical streams. |
| StoneNeedle [161] | Host | Online | LSTM, PCC, K-means | Assigns stream ID based on data hotness prediction based on LSTM and PCC. |
| WARCIP [159] | Both | Online | K-means | Adjusts the stream granularity and learn from the feedback of the SSD dynamically. |
| DTR-FTL [153] | SSD | Online | DTR | Assigns superblock using expected data lifetime and block erase count. |
| FileStream [168] | Host | Online | K-means++ | Assigns stream ID using the ratio between update frequency and file size. |
| ML-DT [46] | SSD | Offline | TCN, SVM, LSTM, RF | Assigns superblock based on pre-trained machine learning models. |
| FIOS [34] | Host | Online | K-means PCC | Assigns stream ID using access frequency, sequentiality, and block correlations. |

For both methods, the address space is split into chunks of a given size to reduce the overhead caused by AutoStream. The MQ method has multiple queues corresponding to different hotness in logarithmic scale. All chunks initially have a hotness of 1 and will eventually be promoted to queues for higher hotness if they have enough accesses in a certain period. After each promotion, all head chunks in each queue will be checked to see if they should be demoted. If a certain number of accesses were made to other chunks but not the queue head chunk, then the head chunk will be demoted to the lower queue.

The SFR method is based on sequentiality, frequency, and recency. If a write request starts from the end address of the previous write request, then it will use the same stream ID from the previous write request. For other cases, the recency weight for the chunk will be calculated using $2^{\text{time since prev access}/\text{decay period}}$, where the decay period is a user-controlled variable. The new access count $c_{new}$ of the chunk will be recalculated as $((c_{old} + 1)/\text{recency weight})$. The stream ID will be calculated using the logarithm of the new access count. FIOS [34] later extended this method using PCC to calculate the correlations of different chunks.

*3.2.2 Finding the Layer in Charge.* AutoStream rekindles the research interest of Multi-stream in academia after three years without any published paper on Multi-stream. It also marks the shift from manual stream ID assignment to automatic, which reduces the development overhead of application developers. Several approaches were created in different layers to automate the stream ID assignment process; possible layers to implement stream assignment include the filesystem [134], device driver [158], runtime [96], and SSD itself [153].

FStream [134] provided automatic stream ID assignment by separating different filesystem metadata. For example, ext4 has journal, inode, and other miscellaneous information. FStream separates those metadata into different stream IDs with the ability to assign distinct streams to files with specific names or extensions. FileStream [168] inherited the design choice of working on the file level. Unlike most of the work in this category, which attaches a stream ID to each single write request, FileStream calculates the stream ID based on the file. It attaches related information to the file inode, which is stored in the VFS layer. Files with the same parent path and file extension are considered the same type of files. The FileStream mapper aims to reduce the mixture of different file types and the lifetime difference of different files, and the remapper will group files with similar characteristics using the K-means++ algorithm into a limited number of stream IDs based on the mapper results.

vStream [164] is the only work that requires manual stream ID assignment in this category after the original Multi-stream paper. However, it proposed a new concept called virtual streams so writes from different sources can have their own streams. Multi-stream SSDs only allow for a limited number of concurrent streams, which is not sufficient for a large number of tenants. Different Multi-stream SSDs may provide different numbers of streams, which requires developer attention if the stream IDs are assigned by the application. The problem intensifies when different applications use the same stream ID for different purposes, which renders streams useless. By providing a large number of virtual streams (i.e., $2^{16} - 1$ in vStream), the problem can be mitigated, since there are enough (virtual) streams for different applications and different purposes. A remapper in the SSD will map the virtual streams into a limited number of physical streams using the K-means algorithm to group virtual streams with similar characteristics to the same physical streams, marking the first of several works in this category using K-means to cluster several entities (e.g., files, streams) into a limited number of streams.

Both WARCIP [159] and DStream [111] later proposed having a variable number of clusters when using K-means. WARCIP separates the address space into chunks, similar to AutoStream. It clusters chunks with similar lifetimes into the same cluster using K-means, where each cluster

corresponds to a stream. However, the number of clusters can change depending on demand. If a cluster becomes too busy, i.e., too many requests have been put into a single cluster, then the cluster will be split into two. In contrast, WARCIP may merge two clusters if a cluster does not receive enough write requests in a period. The SSD will also provide feedback to the host if the host falsely clusters long-lived data into clusters intended for short-lived data. Together, these features ensure that each cluster minimizes the write interval of different write requests in each stream for a better WAF. DStream, however, shows that the SSD internal metadata writes may increase if there are too many streams. It counts the number of updates for each logical page, which will be used as the standard of hotness; however, when using K-means to group pages into clusters, it may combine the two closest clusters into one if incoming data has a farther distance than the distance between the two clusters and vice versa.

PCStream [95, 96] used **program context (PC)** instead of block address or file information when assigning stream IDs. The program context is defined as the call stack when a write request is issued. By knowing the PC, one can identify which series of function calls ultimately caused the write request, which tells the origin of the data. For applications written in Java, the PC lies in JVM, which the authors have modified to support PCStream for Java applications. Since data from the same origin can be considered to have the same characteristics, it makes sense to place data from the same origin to the same stream. PCStream then uses K-means to cluster multiple different PCs with similar characteristics into the same stream, since the number of stream IDs is limited.

DTR-FTL [153] is a scheme implemented in the FTL layer, which means that the host does not directly send stream IDs to the SSD. The two components, lifetime-rating addressing and time-aware garbage collector, work inside the SSD and decide which erase unit a write request should be assigned to. The lifetime-rating addressing strategy assigns hot data to superblocks with higher **Program/Erase (P/E)** cycles, since blocks with higher P/E cycles have a shorter data retention time, and cold data will stay on the SSD for a longer time. However, the time-aware garbage collector interpolates the expected valid pages left in the superblock. Let $T_{mean}$ indicate the average data invalidation time of a given superblock. After $T_{mean}$, half of the pages in the superblock became invalid, leaving the other half valid. The algorithm chooses the block with the smallest number of expected valid pages calculated by interpolation. During the GC process, the relocated valid pages will be placed into a superblock with a retention time that matches the expected lifetime left for those pages.

*3.2.3 Machine Learning Model Approaches.* With the increasing popularity of machine learning models, some approaches leverage machine learning for stream ID assignment. StoneNeedle [161] started applying complex machine learning algorithms to assign stream IDs. It extracts useful workload features to calculate hotness, which is later processed using PCC to determine the correlation between hotness and features. **Long short-term memory (LSTM)** is then used to learn the characteristics for stream ID prediction. A similar approach is later adopted by ML-DT [46]. One major difference is that the training process is offline, which means that it cannot adapt to different workloads on the fly. This is because the training process for machine learning models is too heavy to be placed in SSD, since SSD has limited computational power. However, ML-DT utilizes multiple machine learning algorithms, including **temporal convolutional network (TCN)**, LSTM, **support vector machine (SVM)**, and **random forest (RF)**. The authors concluded that TCN is the best model with the best accuracy and the lowest resource requirement in SSD. It is also worth noting that this work is heavily influenced by Multi-stream and its related work (which can be seen from the Evaluation section of the paper), but it does not directly use the Multi-stream interface between the host and the SSD. Rather, the trained model will be placed inside the SSD and used to assign different superblocks internally.

```
// RocksDB
// db/flush_job.cc
Status FlushJob::WriteLevel0Table() {
  // ...
  auto write_hint = cfd_->CalculateSSTWriteHint(0);
  // ...
}

// db/compaction/compaction_job.cc
void CompactionJob::Prepare() {
  // ...
  write_hint_ = cfd->CalculateSSTWriteHint
                      (c->output_level());
  // ...
}
Status CompactionJob::OpenCompactionOutputFile
(SubcompactionState* sub_compact,
CompactionOutputs& outputs) {
  // ...
  writable_file->SetWriteLifeTimeHint(write_hint_);
  // ...
}

// env/io_posix.cc
void PosixWritableFile::SetWriteLifeTimeHint
(Env::WriteLifeTimeHint hint) {
  // ...
  if (fcntl(fd_, F_SET_RW_HINT, &hint) == 0) {
    write_hint_ = hint;
  }
  // ...
}
```

```
// ZenFS for ZNS Eval
// fs/io_zenfs.cc
IOStatus ZoneFile::SetWriteLifeTimeHint
(Env::WriteLifeTimeHint lifetime) {
  lifetime_ = lifetime;
  return IOStatus::OK();
}
void ZonedWritableFile::SetWriteLifeTimeHint
(Env::WriteLifeTimeHint hint) {
  zoneFile_->SetWriteLifeTimeHint(hint);
}

IOStatus ZoneFile::Append
(void* data, int data_size, int valid_size) {
  // ...
  active_zone_ = zbd_->AllocateZone(lifetime_);
  // ...
}
```

Listing 2. Code from RocksDB (left) and ZenFS (right). The RocksDB code for calculating `write_hint_` (not shown) and using this value as desired stream ID (shown here) was added for Multi-stream support [110]. ZenFS uses the same algorithm and I/O path for calculating and passing `write_hint_` to the device [70].

*3.2.4 The Fall of Multi-stream.* Despite the promising results of prior works above, the Linux kernel removed Multi-stream support in the `bio` layer and the default NVMe driver in 2022 due to the low adoption rate [69]. To quote the commit message, "No vendor ever really shipped working support for this, and they are not interested in supporting it... No known applications use these functions." One possible reason is the emergence of Zoned Namespaces SSD, which we will discuss in Section 4.2. However, the removal of Multi-stream and write hints from the `bio` layer received backlash from some vendors, including Samsung [105], Micron [130], and Western Digial [140]. They mentioned that the write hints in the `bio` layer was used by UFS [53], which is a common standard for host/flash communication used in smartphones [72]. The write amplification can be significantly reduced when using F2FS with write hints. Despite the support from the vendors to keep write hints and Multi-stream in the Linux kernel for UFS, the kernel maintainers still decided to remove the relevant code, stating the lack of Multi-stream interface implementation on UFS in the Linux kernel to support the vendors' claims [28]. The support for Multi-stream and write hints was ultimately removed in the Linux kernel v5.18-rc1 [69].

Although Multi-stream eventually faded out from the history, there are several legacies left by Multi-stream. The authors of FileStream mentioned that they would like to apply their scheme to Zoned Namespaces SSD. The ZNS Evaluation paper [36] used the same algorithm and I/O path for assigning stream IDs to ZNS zones in RocksDB [70, 110]. Listing 2 shows the relevant code. Using the method `SetWriteLifeTimeHint`, RocksDB sets the write hint for a given file depending on its type (e.g., SSTable of different levels, WAL), which is calculated using the `CalculateSSTWriteHint` method. When writing a file, the write hint will be passed, using the `fcntl` operation `F_SET_RW_HINT` (added to the Linux kernel with Multi-stream [9]), as the stream

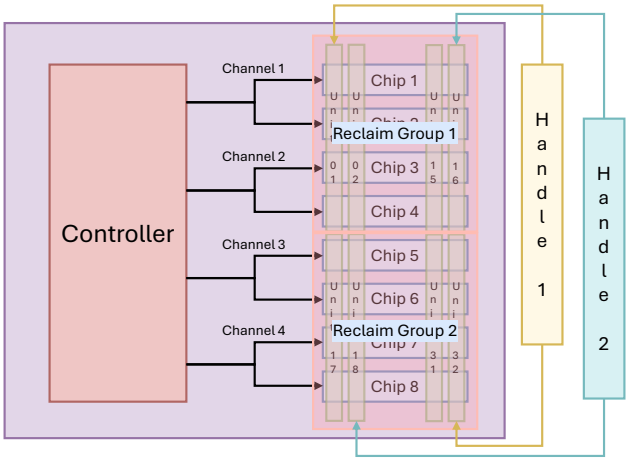


Fig. 5. The overall structure of an FDP-enabled SSD.

ID when the underlying SSD supports Multi-stream. The `F_SET_RW_HINT` operation is one of the few parts that survived from the removal of Multi-stream code, since it is used in ZenFS and ultimately ZNS SSD; the same write hint field is used to determine the zone to write to, which requires the `F_SET_RW_HINT` operation. This shows that there are some possibilities to reapply schemes in the Multi-stream category to ZNS SSDs, and it is possible to see some of the schemes built for Multi-stream SSDs appear in ZNS.

### 3.3 Flexible Data Placement (FDP)

Flexible Data Placement can be considered as an enhanced version of Multi-stream in some sense. The underlying concept of FDP is very similar to Multi-stream: By passing directives of desired data placement location when writing data to the SSD, the SSD can place data with similar invalidation timeframes into the same erase unit, which reduces write amplification caused by valid data relocation during garbage collection. However, FDP provides more flexibility than Multi-stream. We summarize these flexibilities into two main points: the ability to change garbage collection units and the ability to provide feedback to the host.

Traditional SSDs perform garbage collection on a superblock level, which is a collection of blocks from all chips. Traditional SSDs do not provide any details about this information, and the garbage collection unit is also nonconfigurable by the host. FDP provides the ability for hosts to configure the garbage collection unit based on the capability of the SSD. An FDP-enabled SSD provides the host with a list of possible FDP configurations, which tells the possible granularities of the garbage collection unit supported by the SSD as shown in Figure 5. The granularity is defined by the **reclaim unit (RU)**, which can be as small as a single erase block on a die or as big as a superblock. The reclaim units are then organized into **reclaim groups (RGs)**; each can contain as little as one reclaim unit or as many as all reclaim units [137]. The SSD provides a list of **reclaim unit handles (RUHs)**; each points to a reclaim unit in every single reclaim group. After choosing a configuration, the host can choose to place data in different garbage collection units by providing desired RUH and RG, ensuring that data with different RUHs and RGs will not be mixed, similar to Multi-stream. When an RU pointed by an RUH is full, the SSD automatically chooses another RU in the same RG to that RUH. Furthermore, the host can also allow or disallow data isolation after garbage collection within a reclaim group, i.e., if data from the same reclaim group but from different reclaim units can be mixed together after garbage collection

Table 2. A Summary of Prior Works that Can Be Potentially Improved with FDP

| Name | Category | Description |
|---|---|---|
| WARCIP [159] | WAF Reduction | Provides a feedback mechanism for SSDs to inform the effectiveness of data separation. |
| PLAN [167] | WAF Reduction | Uses differently sized GC units for sequential and random workloads for lower GC overhead. |
| OPS-Iso [94] | Perf. Isolation | Limits GC activity caused by a tenant to itself to prevent interference between tenants. |
| VSSD [47] | Perf. Isolation | Provides virtual SSDs to different users (tenants) and uses a fair scheduler for fairness between different tenants. |
| FlashBlox [71] | Perf. Isolation | Provides virtual SSDs with customizable SSD-internal resources, including channels and dies, depending on the tenant demand. |
| CostPI [113] | Perf. Isolation | Provides virtual SSDs with further customizable SSD-internal resources such as data cache and mapping table cache. |
| DC-Store [100] | Perf. Isolation | Provides virtual SSDs to different containers using NVMe set on a real SSD with statically assigned hardware resources. |

The genealogy tree is not provided, as papers in each category show a linear, chronological relation.

[120]. These features more flexible than Multi-stream and ZNS when choosing data placement locations.

Additionally, traditional SSDs do not provide any garbage collection-related statistics or feedbacks to the host. This is the same for Multi-stream SSDs: Although the host can choose to place data by attaching a stream ID, the host does not know the effectiveness of such information; in other words, the host cannot identify if providing the given stream ID helps in reducing write amplification. FDP addresses this issue by providing statistics and events related to garbage collection. The host can now query the exact number of bytes written by the host or the SSD, which is enough for the host to calculate the exact write amplification factor. FDP also supports event logging; some notable examples include [120]:

— If the reclaim unit has changed for a write frontier (e.g., due to garbage collection);
— If the reclaim unit is underutilized (i.e., not written to full in a time period);
— If the reclaim unit was not written to full when the host changes a write frontier to another reclaim unit.

These features allow the host to learn about the internal states of the SSD with respect to garbage collection, and the host can then adapt accordingly to achieve lower write amplification.

Although FDP has been recently ratified and accepted by NVMe, it still takes time for FDP to be applied and researched, since the new NVMe version with FDP support is yet to be released. However, we identify two paper categories closely related to FDP. The first category is related to those FDP features on reducing WAF, and the second category is SSD performance isolation. A summary of the papers can be found in Table 2.

*3.3.1 WAF-reducing FDP Features.* One feature provided by FDP is the ability to provide feedback to the host regarding data placement and its effectiveness, so the host can dynamically change how data should be placed physically. WARCIP [159], which we discussed in the Multi-stream section (Section 3.2), provides similar feedback mechanism from the SSD to the host; the host dynamically merges and splits streams according to the utilization of each stream provided by the SSD feedback mechanism.

The second paper is PLAN [167], which utilizes some concepts in FDP, including dynamic garbage collection unit granularities and shows that SSDs with superblocks exploiting all chips do not show the best performance when performing random writes. The performance can be improved by dynamically assigning erase units with different sizes by adjusting the number of chips used based on write characteristics. PLAN shows the feasibility and effectiveness potential of FDP if RUs (i.e., erase units) with different sizes are used. However, PLAN is designed entirely within the SSD firmware, which means that it can only infer the necessary information from the request heuristics. With FDP, the host can directly control the organization of reclaim units from the information that the host has, which is more comprehensive than what the SSD sees. This leads to a lower write amplification factor and a better SSD performance due to more informed decisions.

*3.3.2 Performance Isolation.* It is expected for SSDs to have multiple tenants. In a single computer, several programs may run concurrently and issue I/O requests; in a large data center, an SSD may be shared by multiple tenants, such as containers, virtual machines, and users. It is important to ensure that the I/O requests of one tenant do not disturb other tenants; otherwise, the performance of other tenants may be affected [100]. Since FDP provides an interface for choosing RUs that can potentially sit on top of different channels and chips for data placement, using FDP to achieve performance isolation is feasible. The host can take over the task of identifying different tenants and provide better performance isolation results with the FDP interface. Below are five papers we believe are most related to FDP with the potential to be improved using FDP.

The first paper that may be potentially improved with FDP is OPS-Iso [94], which shows that whole-SSD GC causes disturbance between different tenants, because one tenant could trigger GC and affect the I/O performance of others. As a mitigation, the paper separates the GC activities of different tenants so the GC triggered by a tenant will not affect others, since the GC activity is limited to the chips used by that tenant. Another paper from 2015, VSSD [47], provides different **virtual SSDs (vSSDs)** to different users and uses a fair scheduler to provide service to them. A later work, FlashBlox [71], provides vSSDs in different granularities (i.e., channel-isolated, chip-isolated, etc.), which is very similar to FDP. CostPI [113] from ICPP 2019 provides more isolation for other parts of SSD internals, including mapping table cache and data cache, in addition to the chip isolation used in its prior works. Last, DC-Store [100] provides performance and resource isolation for containers. It implements NVMe sets on a real SSD and statically assigns internal SSD resources to segregate different tenants physically. These prior works can be implemented with FDP support to provide better performance isolation with tenant identification on the host side and garbage collection segregation in the SSD. To conclude, we hope that more research can be done after FDP is integrated into the NVMe standard.

## 4 Enhancing Host Control over SSD

In this section, we will mainly discuss the design in which the host system has more control over the SSD. Traditional SSDs are considered "blackboxes," and the SSD exposes limited information to the host. The host only knows some key characteristics of the block device, most importantly, the device capacity. Other characteristics (e.g., TRIM support) are considered optional. Internal activities, such as garbage collection and wear leveling, are not exposed to the host at all.

However, the SSD types in this section expose more information to the host. The host takes over some of the SSD's responsibilities. Unlike the SSD types in the previous section, the SSD types in this section are not compatible with the traditional SSDs, which means that a code change is required at some level. There are several candidates, including the application layer, the filesystem layer, or the driver layer. The choices are paper-specific, and we will discuss the choices in detail in the rest of the section.

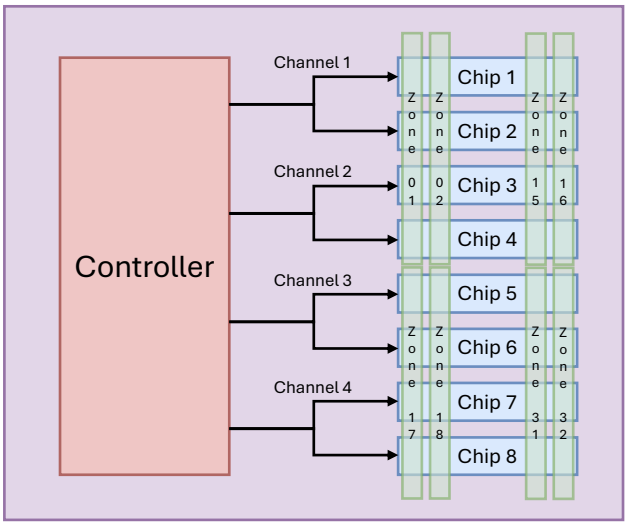


Fig. 6. The overall structure of a Zoned-namespaces (ZNS) SSD.

### 4.1 Open-channel (OC) SSD

To overcome the opacity of traditional SSDs, Open-channel SSD is created to provide a transparent SSD from the host point of view. An Open-channel SSD moves most SSD responsibilities to the host, including metadata management, write buffering, and wear leveling [38]. Users can directly place data on the desired channels [127, 151]. However, the flexibility of the Open-channel SSD is a double-edged sword. The application has to be aware of the existence of the Open-channel SSD, and the developers have to manage most SSD responsibilities, further limiting its application. The Open-channel SSD protocol was never ratified by the NVMe standard or widely adopted by industry [64], and it was eventually abandoned by the maintainers and substituted by Zoned Namespaces [7].

### 4.2 Zoned Namespaces (ZNS) SSD

*4.2.1 Overview.* Similar to Open-channel SSDs, ZNS SSDs also transfer some SSD responsibilities to the host. It was inspired by **shingled magnetic recording (SMR)** HDDs, where the storage device is organized into a series of zones [35]. Figure 6 shows the overall structure of a ZNS SSD. When handling a write request, the host system has to choose a zone to write to, and the incoming data has to be sequentially appended to the zone, which can span several channels and dies, but not necessarily all channels and dies, as opposed to traditional SSDs [30, 167]. Furthermore, unlike Multi-stream SSD, where the placement information (given as the form as stream ID) is considered optional, the host has to explicitly choose a zone to write to. The host must also manage the garbage collection of ZNS SSDs by choosing a victim zone to perform GC. In summary, ZNS SSDs expose their superblocks as zones to the host in some sense, and the host has to choose a zone to write to when issuing a write request and a zone to erase when the number of available zones falls under a certain range. In the rest of the section, we will first focus on the efforts to support ZNS on the host side, then discuss the research works shown in the genealogy tree (Figure 7). A summary of the research work related to ZNS can be found in Table 3.

*4.2.2 Bringing Host Support to ZNS SSDs.* When compared to Open-channel SSDs, ZNS SSDs require fewer responsibilities to be moved from the host [117]. This makes the development of

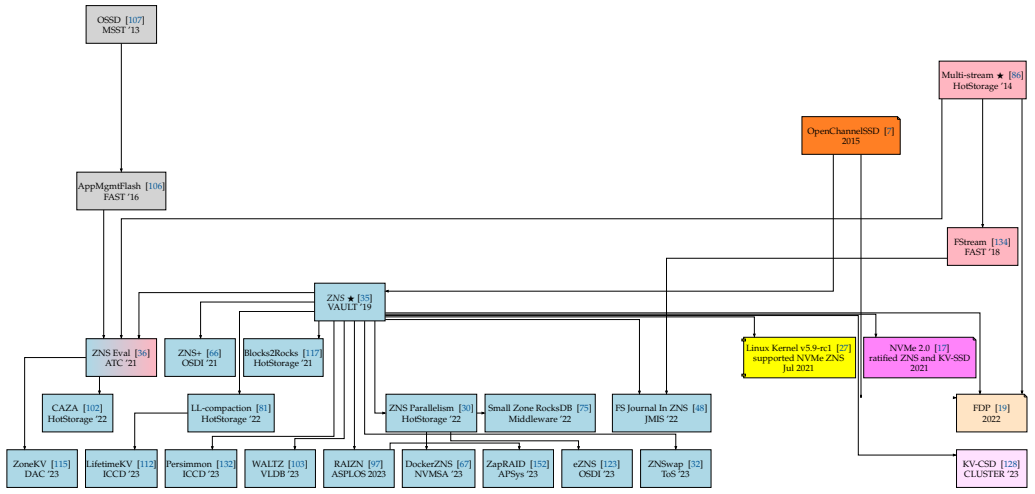


Fig. 7. Zoned-namespaces (ZNS) related works.

ZNS-aware applications easier. The existing code for Multi-stream SSD can be used directly by applying stream ID as the zone number. The existing code in the kernel can also be recycled similarly. Listing 2 shows an example of recycling the Multi-stream stream ID assignment algorithm and the datapath by ZNS. The modification in RocksDB was intended for Multi-stream, but was directly used by ZenFS, the storage backend extension for RocksDB ZNS SSD support [36]. ZenFS is backed by a simple filesystem named ZoneFS, which exposes each zone as a file [59]; each file stores its corresponding zone information in its inode [56]. The desired zone number will be selected using the RocksDB based on the type of data (e.g., SSTable level) by rehashing the stream ID calculation logic from Multi-stream. RocksDB then passes the zone number to ZenFS as its middle layer, which ultimately uses ZoneFS as the backing filesystem. By writing to different files, each representing a zone, RocksDB can effectively separate data with different characteristics into different GC units.

However, ZoneFS has its limitations: The files representing different zones can only be written sequentially [59], requiring developers to write additional code to make their applications ZNS-compatible. Some traditional filesystems, including F2FS and Btrfs, now support ZNS SSDs as a backend so applications can run with ZNS SSDs without modification. Both filesystems can adapt to ZNS SSDs relatively easily due to their underlying design concepts. F2FS is a log-structured filesystem, which means that all write requests are placed sequentially; in other words, F2FS aligns intrinsically with the design concept of ZNS SSDs. However, it still requires conventional zones (i.e., with random write support) or other conventional devices to place filesystem metadata due to its design [58].

However, Btrfs is designed based on **Copy-on-Write (CoW)** to place the newer version of data in a new location instead of performing the in-place update, which also aligns with the design concept of ZNS SSDs. Since the only Btrfs metadata that requires a fixed location is the filesystem superblocks (not to be confused with the superblocks in SSDs we described in previous sections), two zones are reserved for filesystem superblocks. The latest filesystem superblock will be appended to the previous version. When one zone is full, the filesystem superblock will be written to the other zone, and the previous zone will be reset and used when the new zone is full. The latest filesystem superblock can always be easily retrieved by checking the last write pointer location, since it is always appended to the previous filesystem superblock [57].

Table 3. A Summary of ZNS and Its Related Works

| Name | Changes | Target App. | Description |
|---|---|---|---|
| ZNS [35] | Proposal | N/A | Proposes a new SSD type and storage abstraction named Zoned Namespaces SSD. |
| ZNS+ [66] | Filesystem, Protocol, SSD | General | Proposes a new extension to the ZNS interface with support of offloading data copy to SSD and threaded logging. |
| Blocks2Rocks [117] | Filesystem, Protocol, SSD | Filesystem, KV DB | Proposes a new extension to ZNS SSD protocol to store variable-sized "rocks" without fragmentation or read-modify-write. |
| ZNS Eval [36] | Filesystem, App., SSD | KV DB | Evaluates ZNS SSD against traditional block-interface SSD (including Multi-stream SSD) under key-value database workloads. |
| CAZA [102] | Filesystem | KV DB | Places SSTables having overlapped key ranges to the same zone instead of putting SSTables of the same level to the same zone. |
| ZNS Parallelism [30] | Scheduler | General | Identifies conflicting zones that degrade performance when writing concurrently and reschedules requests in I/O scheduler. |
| LL-compaction [81] | LevelDB | KV DB | Splits SSTables when creating so short-lived key ranges will not mix with long-lived key ranges. |
| Small Zone RocksDB [75] | Filesystem | KV DB | Exploits internal parallelism on small-zone ZNS SSDs by striping SSTable to different zones when writing. |
| FS Journal in ZNS [48] | Kernel | Filesystem | Separates journal and data for legacy filesystems, e.g., ext4, on ZNS SSDs similar to FStream [134]. |
| DockerZNS [67] | Kernel | Docker | Provides a solution to use ZNS SSDs for Docker images with QoS support and performance isolation. |
| Persimmon [132] | Filesystem | General | Makes F2FS more ZNS-native by keeping metadata and checkpoints append-only. |
| RAIZN [97] | Kernel | General | Provides a software RAID with ZNS SSDs and expose as a large ZNS volume. |
| ZapRAID [152] | Driver | Universal | Provides a software RAID with ZNS SSDs and expose as a volume with random read/write support. |
| eZNS [123] | Driver | General | Assigns variable non-conflicting zones into logical v-zones to maximize internal parallelism utilization and throughput. |
| WALTZ [103] | Filesystem | KV DB | Reduces KV DB latency by writing to other zones instead of asking for zone utilization when a zone is full. |
| LifetimeKV [112] | Filesystem, App. | KV DB | Minimizes SSTable lifetime variety in a level by minimizing inter-level key overlap and compacting long-lived SSTables. |
| ZNSwap [32] | Kernel | Linux Swap | Provides an efficient way to use ZNS SSDs for Linux Swap space. Extended from their FAST 2022 work [31]. |
| ZoneKV [115] | RocksDB | KV DB | Skips the 4-level write hint limitation left by Multi-stream, which limited SSTables of $L_4$ and beyond being written to different zones. |

In summary, ZNS SSDs can be supported in two ways: application being ZNS-aware or filesystem with ZNS support. The first approach provides more flexibility for applications to control data placement, but this could be time-consuming for developers and eventually hinder the adoption of ZNS SSDs. To mitigate the issue, filesystems can be designed to be ZNS compatible so applications can run with ZNS SSDs smoothly without being aware of the underlying SSD type.

*4.2.3 Improving Performance on KV Database.* Early research on ZNS focused on combining key-value databases with ZNS SSDs. The policy of writing key-value database files is the central topic for these works. ZoneKV [115] tackles a problem inherited from Multi-stream. The write hint brought in by Linux kernel and RocksDB has only four levels of temperature, namely, `SHORT`, `MEDIUM`, `LONG`, and `EXTREME` [9, 110]. ZoneKV sets the lifetime to 1 for $L_0$ and $L_1$, 2 for $L_2$, and 3

for $L_3$ SSTables, which is the same for RocksDB [110]. However, RocksDB sets the lifetime of all SSTables of $L_4$ and beyond to 4 due to the limitation, while ZoneKV sets the lifetime of $L_i$ SSTables to $i$, bypassing the limit of 4.

CAZA [102] from HotStorage 2022 improves the algorithm brought in by Multi-stream and used by the ZNS evaluation paper [36]: Instead of allocating zones by using compaction level only, RocksDB can leverage the SSTable information to make better zone choices. By the definition of SSTable compaction, when a compaction happens, several SSTables with overlapping key ranges will be invalidated together and compacted into a single, new SSTable. This indicates that SSTables with overlapping range should be written to the same zone. If an SSTable has no key overlap, then a new zone is allocated for the SSTable. However, as of February 2024, the improvement brought by CAZA to the `CalculateSSTWriteHint` function is not reflected in the RocksDB codebase [23]. LL-compaction [81], also from HotStorage 2022, provides another zone selection algorithm for KV databases. It focuses on keeping zones dedicated for each compaction level, but the SSTables are split into fine-grained key ranges so key ranges that will be compacted soon will not be mixed with other SSTables in the same zone. However, it does not work on some databases like RocksDB, because they employ a priority-driven SSTable selection algorithm with extra factors, including age and number of deleted items, when choosing SSTables to compact. LifetimeKV [112] also notices the possibility of having short-lived SSTables after compaction similar to LL-compaction. It proposes two mitigations: First, a newly generated SSTable should not have an overlapping key range with upper-level SSTables, so when an upper-level SSTable is chosen for compaction, the new SSTable aforementioned will not be selected for compaction, increasing its potential lifetime to match with other SSTables in the same level; second, an SSTable in a level will be prioritized to be chosen for compaction if it has stayed too long in the level, which reduces the possibility of having multiple SSTables with long lifetime variety at the same level.

WALTZ [103] tackles the KV database performance problem by looking at the problem in another way. Instead of mainly improving the method of placing SSTables like the aforementioned works, WALTZ identifies the usage of the zone report command as a source of performance degradation when a zone is full. When writing to a zone fails (e.g., the zone is almost full), a zone report request for checking the current write pointer location will be sent to check the remaining free space in the zone. This causes extra latency due to the communication overhead between the host and the SSD. Instead of checking SSD for write pointer information, WALTZ utilizes other zones to write the data so the long latency caused by the zone report command can be eliminated.

*4.2.4 Exploiting ZNS Internal Parallelism.* Don't Forget ZNS Parallelism [30] from HotStorage 2022 brings up another topic of ZNS SSD: internal parallelism. Modern SSDs rely on internal parallelism for their blazing speed, but this information is not shared with the host system for traditional SSDs. Interestingly, ZNS SSDs also do not share this information with the host system even though ZNS SSDs are more transparent than traditional SSDs. If a host writes to more than one zone concurrently, then there is a chance that the host is unknowingly writing to the same chip, causing performance degradation. However, some ZNS SSDs expose large zones (e.g., 2.18 GB), while others expose small zones (e.g., 96 MB and utilizing a single flash chip). The SSD may assign a higher degree of parallelism to larger zones, while small zones may not be assigned with a high degree of parallelism. It is possible to write to multiple small zones simultaneously, but again, without knowing the zone-to-chip mapping, the host could unknowingly write to two zones mapped to the same chips. The paper proposed an algorithm to identify zones that are mapped to the same chips: By checking if writing to all combinations of two zones causes performance degradation, the algorithm is able to identify these ***conflict groups* (CGs)** of zones. The system I/O scheduler is then modified to schedule I/O requests so data writing to the same CG will not be

written at the same time for the maximum performance. A later work, eZNS [123], extends the idea by automatically assigning zones that do not conflict with each other into logical zones (v-zones), which contain a variable number of zones. The number of zones in a v-zone can shrink or expand, depending on the workload requirement to maximize the performance of each v-zone.

Small Zone RocksDB [75] also focuses on exploiting parallelism on small zone ZNS SSDs. However, instead of implementing its solution in the I/O scheduler, it is a work focusing only on RocksDB and ZenFS. For ZNS SSDs with small zones, the size of an SSTable is much larger than the zone size. Instead of writing SSTables sequentially zone by zone, the paper proposes striping them sequentially to multiple different zones to exploit parallelism; however, each zone will still be used to save SSTables of a single compaction level.

*4.2.5 Improving ZNS Internals.* There are other works that focus more than bringing ZNS support to key-value databases. ZNS+ [66] identified some potential problems for the vanilla ZNS SSD and host-SSD protocol. When garbage collection is performed, any valid data in the victim zone should be relocated, similar to the garbage collection process in a traditional SSD. However, in a traditional SSD, garbage collection is triggered internally in the SSD. The relocation traffic is not observable by the host but does not require host attention either. ZNS SSD, however, requires the host to manage the garbage collection process. The host has to relocate the valid data from the victim zone, which incurs extra overhead by reading the valid data in the victim zone to the host and then writing them back to the SSD. This process causes data to move not only once but twice between the host and SSD.

To remove the external data movement between the host and the SSD during the garbage collection process, ZNS+ proposed an extension to the ZNS protocol so the data relocation process will be internal to the SSD. The simple NVMe copyback function is extended, named `zone_compaction`, to accept noncontiguous address ranges for valid data relocation. This is for F2FS with threaded logging, where there can be direct overwrites to dirty segments, which is supported by adding `TL_opened` to the ZNS+ protocol. The SSD should also show its internal mapping information to the host: The host then knows which chunk (the smallest unit of copyback) of data is stored on which physical chip. When performing copyback, the host should relocate data from one chip to the same chip to prevent inter-chip traffic. However, to the best of our knowledge, this proposed extension to ZNS is not a part of the NVMe ZNS protocol as of February 2024.

Another work, Blocks2Rocks [117], also proposed changes to the ZNS interface. With the trend of increasing SSD page size (e.g., commonly around 16 KB as of the writing of this survey), updating a small unit of data, e.g., 4 KB, causes overhead due to read-modify-write. The SSD has to invalidate the old physical page and then write the page to a new location. Blocks2Rocks proposed an extension to support small "rocks," which can be as small as 16 bytes, on ZNS SSD. The NVMe command set specification stated that a block size of 512 bytes is not supported [21], which shows the necessity for protocol modifications. The author proposed saving small rocks on in-SSD NVRAM, which requires about the size of one page per active ZNS zone. This space is used to buffer the rocks written to the SSD, which will be transferred to the flash chip for consistency when the buffer is full. However, similar to ZNS+, this work was not ratified into the NVMe protocol.

*4.2.6 Bring ZNS to New Use Cases.* Although the design of ZNS and KV databases is well orchestrated for each other, and the earliest evaluation focuses on KV databases for ZNS-related papers, there are efforts to utilize ZNS SSDs for other kinds of workload. ZNSwap [31, 32] uses ZNS SSDs for Linux swap space. When a memory page is swapped out, it will be written to the SSD. The paper provides different policies on choosing a zone to write to for swapped-out pages. After choosing a zone, the location of the page will be written to the page table entry, which will also be updated if the page is relocated during future GC processes. DockerZNS [67] leverages ZNS

SSDs for Docker images, which may require different QoS and segregation, and there may be noisy neighbors. It provides a number of zones for each Docker image and stripes the image to several zones based on its QoS requirement, which is done by knowing the performance a single small zone can provide. The paper also first identifies the conflict groups of zones [30] so the zones used for the same image will not use the same chips, ensuring maximum utilization of internal parallelism.

There are also works on improving current filesystems for ZNS. FS Journal in ZNS [48] works on separating filesystem metadata and user data, similar to FStream but on ZNS SSDs. Persimmon [132] changes F2FS so it is more ZNS-native by changing the metadata to append-only and improving the checkpoint logic. As we discussed earlier in the section, the original F2FS design expects metadata and checkpoints to be in a known address range, which means that updating metadata and checkpointing lead to in-place overwrites, causing incompatibility with ZNS design goals. Persimmon assigns dedicated zones for frequently updated metadata for easy cleaning, which eliminates in-place updates of filesystem metadata. It also writes checkpoints at the end of each zone for easier garbage collection, since checkpoints will not spill to other zones.

Beyond filesystems, RAID is also a fundamental service for applications. RAIZN [97] provides RAID support for ZNS SSDs and exposes an SSD interface compatible with ZNS-aware applications and filesystems. ZapRAID [152], however, exposes a block-level volume with random read/write support. In summary, these papers together strive for more fundamental improvements for better application performance running on top of them, and we hope future works can show the benefit of using ZNS SSDs for general use cases in different environments, including servers and even smartphones.

## 5 Offloading Host Responsibility to SSD

In this section, we will discuss papers that move host responsibilities to the SSD. Traditionally, an SSD accepts I/O request and provides storage as its only feature. However, moving some jobs from the host to the SSD has some benefits. The first benefit is to reduce the latency caused by the heavy I/O stack in the operating system. Since the operating system I/O stack includes many layers, including driver, `bio`, and filesystem, it makes sense to reduce the number and sizes of I/O requests. If the data can be processed within the SSD, then the data size to be moved from SSD to host can be reduced, which results in a shorter time in the data transfer process [136, 171]. The second benefit is to prevent stacking logs on another layer of logs. Some applications, especially key-value databases, utilize a log-like manner when writing data, which is similar to how SSDs write data internally. By integrating the key-value database into the SSD, the performance can be improved by removing one log layer.

### 5.1 Computational Storage on SSDs

*5.1.1 Prelude.* The concept of computational storage predates the era of SSDs. Active Disks [135] was published in 2001, marking one of the first instances of a computational storage system. The paper identifies that HDDs have processors and RAM just like a normal computer, which means that they can also process data like standalone computers. By processing data inside multiple HDDs before sending them to the host, the performance of Active Disks scales better than using a single host processing data from all HDDs. However, because of the limited computational power of HDDs at the time, the performance of a computational storage system with only one HDD is limited; the system only works better when multiple HDDs are working together. Thankfully, modern SSDs have better computational powers than HDDs, as one can expect based on the Moore's Law. SSDs like Samsung PM1725 have dual-core processors with a frequency of 750 MHz [80]. A single SSD can also easily surpass the throughput of the HDD array presented in Active Disks by

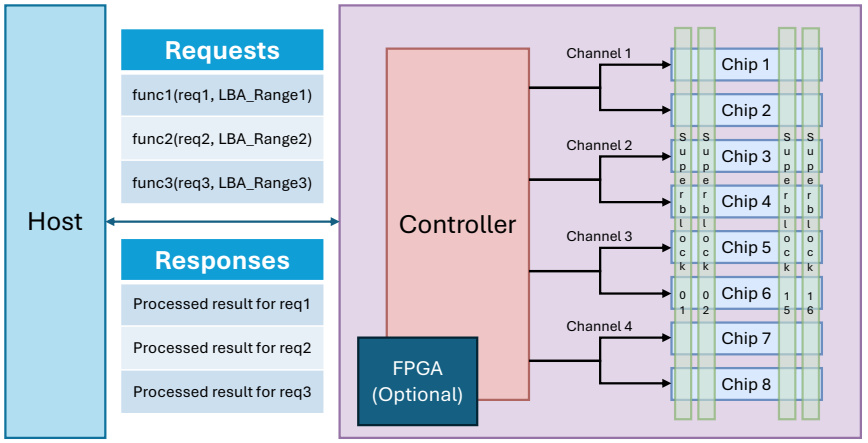


Fig. 8. The overall structure of a Computational Storage-enabled SSD.

exploiting its internal parallelism, which means computational storage can be achieved by using a single SSD.

The structure of a typical computational storage-enabled SSD is shown in Figure 8. The SSD allows a series of functions (also known as tasklets) to be run on SSDs, and the data to be returned to the host will be processed first before sending back to the host. Not only are fewer data transferred to the host, but the host can also immediately use the processed data without the need to process them on the host again. Some works may also use extra FPGA instead of SSD processors for computing. However, reprogramming the SSD firmware or the FPGA is usually challenging. Most commercially available SSDs are blackboxes without the ability to reprogram their firmware, whereas attaching FPGA to SSDs also requires nontrivial efforts. Therefore, many works aim to make coding for computational storage easier and more general without the need to reprogram the firmware or use FPGA. The relationship between different prior works can be found in Figure 9, and the summary of the related papers can be found in Table 4.

*5.1.2 SmartSSD-based Approaches.* SmartSSD [88] is the first work to allow an SSD to be easily programmed for different computational tasks. Users can write C code on the host, then cross-compile it to ARM-architecture binary, and finally embed the program (named *tasklet*) to the SSD firmware. The tasklet is then executed to perform the given task, after which the host polls for results. This requires modification of the SATA protocol with additional vendor-specific commands. The target application scenario for the original SmartSSD is the Hadoop MapReduce type of workload, but it also enabled the ability for arbitrary tasks to be run on SSD for near data processing. QuerySmartSSD [60] leverages the ability to create tasklets and perform query processing, showing better performance and lower energy consumption compared to a traditional SSD. YourSQL [80] is also based on SmartSSD and provides early filtering of SQL query results, which drastically reduces the number of I/Os. However, SmartSSD has a limitation: The tasklet loading process is not dynamic, meaning that a tasklet can only be loaded to the SSD internals offline. Biscuit [65] further enhanced the ability of the original SmartSSD with dynamic task loading and unloading, which means that users can dynamically load their tasks on SSD instead of coupling the task code into the SSD firmware. It also supports the C++ 11 standard and libraries with a few exceptions, providing more flexibility for developers to write and deploy their tasklets.

*5.1.3 FPGA-assisted Platforms.* INSIDER [136] builds a general computational storage platform using FPGA to accelerate in-storage computing tasks. It provides a POSIX-like file I/O APIs to the

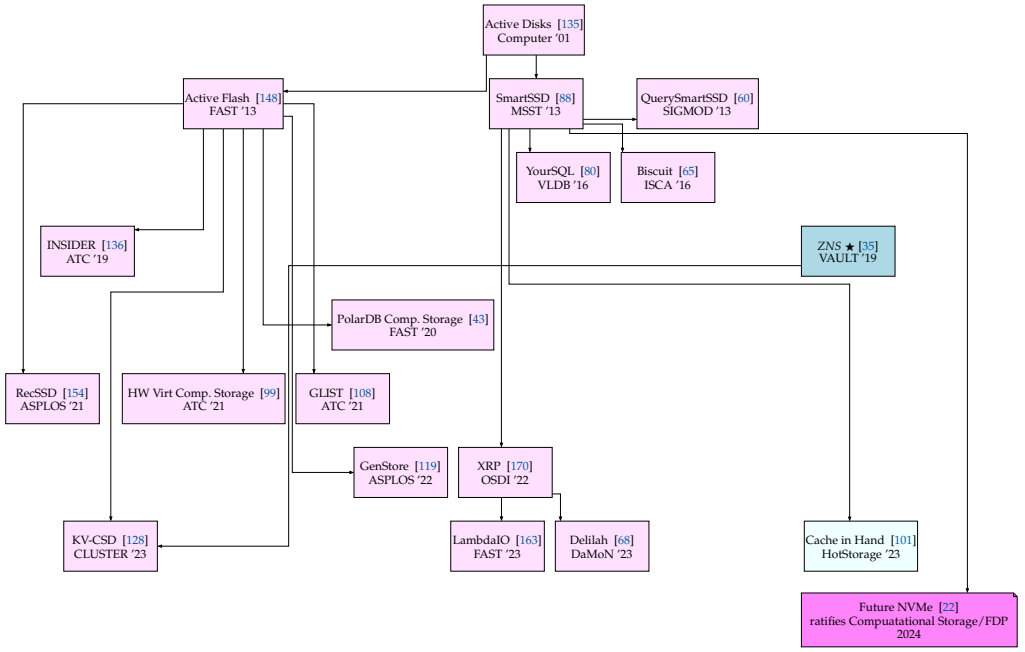


Fig. 9. Computational Storage related works.

user. Users can register tasks and bind them to real files. When reading/writing such real files, the user will be able to read the corresponding virtual files with the results processed by the bounded tasks. Users can also choose to program in C++ or Verilog, depending on the users' needs.

So far, all works we discussed above assume the host is a single physical machine. However, modern cloud infrastructures may use virtual machines to provide the service to different clients. The Hardware-based Virtualization Mechanism for Computational Storage Devices [99] provides the ability to share a single computational storage system using the standard **single-root I/O virtualization (SR-IOV)** layer. It also features an architecture that decouples SSD and FPGA, which allows scaling the number of SSDs in the same system.

*5.1.4 Special Use Cases.* Some works focus on a special use case rather than developing a general platform. PolarDB Meets Computational Storage [43] is a paper focusing on bringing distributed databases with SQL support to computational storage. To achieve this task, the storage engine, the distributed filesystem PolarFS, and the computational storage driver have to be modified. The storage engine passes additional information, including data offset, table schema, and scan conditions of the SQL query, to the PolarFS. Based on those information, PolarFS brings data from different drives, and the computational storage driver will optimize the scan conditions and split them into smaller subtasks before they are passed to the drives for better resource utilization. The underlying block structure is also changed to simplify the FPGA implementation of data scan.

RecSSD [154] focuses on integrating recommender systems on computational storage. However, its operators are embedded in SSD FTL, which means that it lacks the ability to dynamically load/unload different recommender models. Another work, **GLIST** [108] (short for **G**raph **L**earning **I**n-**ST**orage), target on graph learning, which is also used for recommender systems but can also be used in other use cases. To overcome this issue, GLIST provides a set of APIs to directly operate on the graph components (e.g., edge, vertex) and is able to directly analyze graphs using pre-trained

Table 4. A Summary of Computational Storage and Its Related Works

| Name | Type | Target App. | Description |
|---|---|---|---|
| Active Disks [135] | Application | Mixed | The first publication in the category. Based on an array of HDDs, it shows the potential of computational storage. |
| Active Flash [148] | Application | Scientific | Models energy consumption when using computational SSDs. A prototype is also available for common scientific functions. |
| SmartSSD [88] | Platform | MapReduce | The first general-purpose computational SSD platform. Though designed for MapReduce, it allows general tasklets on SSDs. |
| QuerySmartSSD [60] | Application | SQL DB | Uses SmartSSD for query processing. Shows SmartSSD can be used for applications other than MapReduce. |
| YourSQL [80] | Application | SQL DB | Uses SmartSSD for early filtering. This significantly reduces the number of I/Os in certain workloads like SQL `JOIN`. |
| Biscuit [65] | Platform | General | Extends SmartSSD with the ability of loading tasks online and C++ 11 support. |
| INSIDER [136] | Platform | General | Provides a computational storage platform with a set of APIs and the ability to program in C++ and Verilog. |
| PolarDB Comp. Storage [43] | Application | PolarDB | Leverages FPGA for scan conditions on PolarDB with multiple drives. |
| HW Virt Comp. Storage [99] | Platform | General | Provides the ability to share the computational storage system by several virtual machines. |
| RecSSD [154] | Platform | Recommender | Provides the ability to embed recommender-related operators in SSD FTL; reduces unused data between host and SSD. |
| GLIST [108] | Platform | Graph | Provides a set of APIs to operate graphs and graph components and analyze graphs using pre-trained models. |
| GenStore [119] | Platform | Gene Analysis | Provides an in-storage processing system for genome sequence analysis to reduce data movement between host and SSD. |
| XRP [170] | Platform | General | Integrates eBPF functions into NVMe drivers for data processing. Processes tasks closer to data but not in storage. |
| LambdaIO [163] | Platform | General | Provides a platform for using extended eBPF for storage named sBPF in computational SSDs with verifications. |
| Delilah [68] | Platform | General | Similar to LambdaIO, but with ordinary eBPF and without eBPF program verification. |
| KV-CSD [128] | Application | KV DB | Uses Linux-based SoC as the computational power and ZNS SSD to provide a KV-SSD-like interface to the host. |

models, which can be dynamically loaded by the host. This addresses the limitation of RecSSD and allows the final result to be sent to the host without the need for large data movements outside the SSD.

GenStore [119] brings computational SSD into genome sequence analysis. It also filters out unwanted data to reduce the data size to be transferred from SSDs to the host. GenStore has two modes: accelerator mode and regular mode. the accelerator mode allows the SSD to perform in-storage processing while the regular mode allows the SSD to be used as a regular SSD.

KV-CSD [128] has a unique combination of two worlds: computational storage on ZNS SSDs. By using a Linux-based SoC and implementing the key-value store in the SoC, KV-CSD has a similar architecture to KV-SSDs, where a host-level application uses the device as a key-value database.

*5.1.5 Energy Concerns.* Many papers in the field of computational storage discuss their energy efficiency compared to traditional computer systems [60, 80, 88]. ActiveFlash [39, 148] focuses on modeling the energy consumption of computational SSDs. The authors also created a prototype using OpenSSD with common scientific data processing functions in SSD, including max, mean, standard deviation, and linear progression.

*5.1.6 Toward a Standardized Computational SSD Design.* We have observed a dozen prior works that focused on computational storage with SSDs. However, there was no industry standardization effort that allows users to create tasklets without regard to the underlying storage device. Thankfully, the NVMe standard is now looking into the possibility of using eBPF for computational storage [68]. This can be traced back to XRP [170, 171], which shows the ability to use eBPF in NVMe drivers. The paper shows that given the increasing speed of storage devices, the upper layers above the SSD in the host now account for almost 50% of the total overhead, of which the filesystem accounts for 80% of the overhead. The closest layer to the SSD, being the NVMe driver, only accounts for about 2% of the total overhead. This means moving the data processing location to the NVMe driver can significantly reduce latency by about 50%. The BPF function, which is saved in a pointer in the `bio` request, will be invoked when an NVMe request completes. Although this work should not be categorized as computational storage, this work shows close relationships with two later works, LambdaIO [163] and Delilah [68], both of which use BPF functions for in-storage processing.

LambdaIO [163] argues that using eBPF has limitations to general in-storage computing architecture, because eBPF requires a static verifier. eBPF does not support pointer access and dynamic-length loops, which are common but error-prone features. The authors present $\lambda$-IO and sBPF, where *s* stands for storage. The $\lambda$-IO part provides a set of APIs by extending common file APIs, allowing users to provide functions to perform in SSD. sBPF addresses the limitation of no pointer accesses and dynamic-length loops in the standard eBPF. The combination of these two provides a platform for developers to develop in-storage computing functions using familiar frameworks. Delilah [68] is a similar work but has more limitations, e.g., no verification of the provided eBPF programs. Last, recent NVMe technical proposal TP4091 about computational storage was created for a unified computational storage stack [68], and Samsung created a new generation of SmartSSD based on TP4091 [124]. We sincerely hope that the interface for computational storage on SSDs can be fully standardized in the near future.

## 5.2 Key-value (KV) SSD

Key-value SSDs provide a key-value interface instead of the traditional block interface. The host directly communicates with the SSD with a key-value interface similar to how applications use key-value databases. The SSD internally functions as a key-value database, and the FTL is in charge of the mapping management from keys to values, as shown in Figure 10 [89]. In some sense, KV-SSD can be considered as a special kind of computational storage, since the SSD is designed to handle one task only [166]. Figure 11 shows the relationship between KV-SSD-related works, and Table 5 shows the summary of KV-SSD-related works.

*5.2.1 Prelude.* Before the standardized KV-SSD [89], there are some other papers focusing on creating SSDs with better database support on the hardware level. One of the earliest attempts is the X-FTL [87], which exposes a standard SSD interface but with extended SATA and filesystem operations. A transaction ID can be provided to the read/write request for consistency, and two new operations, `commit()` and `abort()`, are provided with transaction ID as the argument. The FTL is also extended with an extra mapping table to record which pages are for which transaction, which works like a list of SQLite rollback journals, and the page mapping from the extra mapping

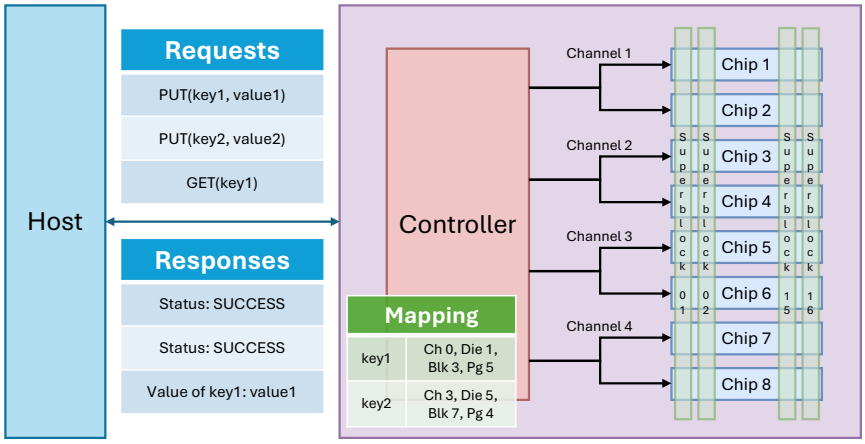


Fig. 10. The overall structure of a Key-value (KV) SSD.

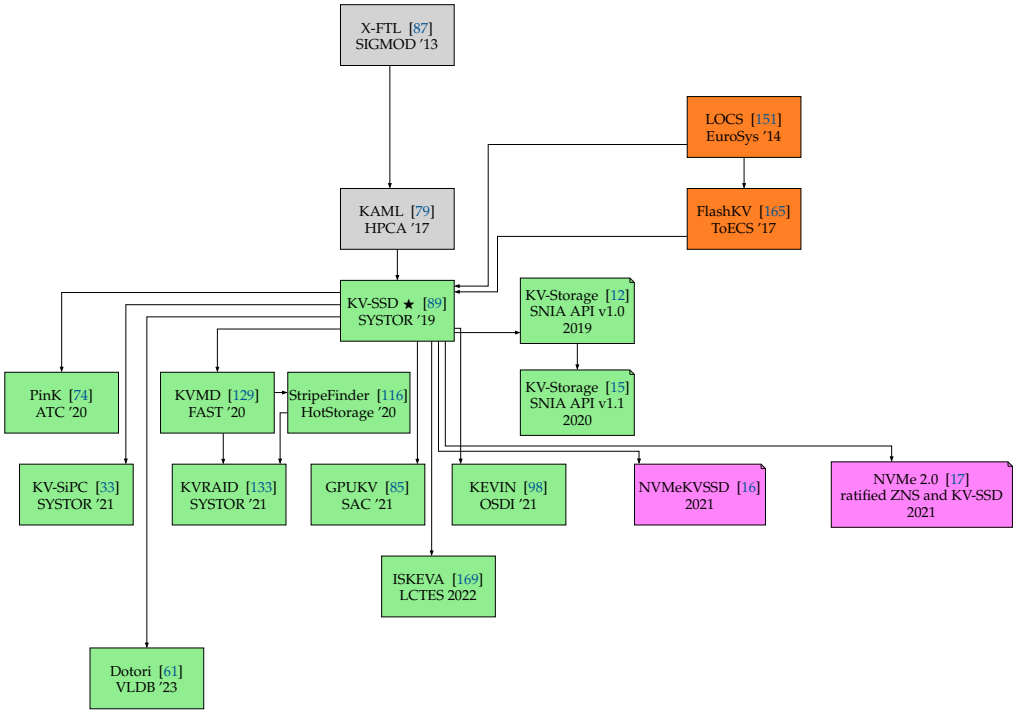


Fig. 11. Key-value SSD related works.

table will reflect on the normal mapping table when the transaction is committed or discarded in case the transaction is aborted. However, X-FTL still requires a modified version of SQLite running on top of the host to communicate with the underlying modified filesystem and device.

Although X-FTL works on improving SSDs for better SQLite support, later works focus on key-value databases. This is because key-value databases have some responsibilities that overlap with SSD FTL: Namespace management manages the mapping from a key to a value, which is similar to

Table 5. A Summary of KV-SSD and Its Related Works

| Name | Description |
|---|---|
| X-FTL [87] | Extends the SATA protocol and filesystem for better SQLite transactional support on SSD. |
| KAML [79] | Provides a new, key-value-based interface for the SSD; applications can directly use the SSD as a key-value database. |
| LOCS [151] | Uses Open-channel SSD for LevelDB. The paper provides policies to improve performance when performing I/O requests. |
| FlashKV [165] | Allows a leaner I/O stack for LevelDB by using an Open-channel SSD with a custom user library and driver. |
| KV-SSD [89] | Provides a leaner I/O stack than KAML by posing the key-value APIs directly from the device. The first vendor's attempt to create a key-value SSD. |
| PinK [74] | Solves the long tail latency caused by bloom filters because of its probabilistic nature. Instead of using bloom filters, PinK pins top several levels in the DRAM for faster access speed. |
| KVMD [129] | Provides the ability to use several KV-SSDs for better performance and reliability. Similar to RAID in traditional block-interface SSDs. |
| StripeFinder [116] | Improves the overhead caused by the erasure coding approach in KVMD. However, the paper claims that it is better to use replication instead of erasure coding for workloads with many small objects. |
| KV-SiPC [33] | Extends the use case of KV-SSD to OpenMP applications by translating OpenMP API calls into KV-SSD operation calls. It also dynamically changes the number of parallel compute threads and parallel data access threads based on CPU and SSD utilization. |
| KEVIN [98] | Uses KV-SSD as the foundation for a general filesystem. Translates file I/O-related system calls into KV-SSD operations and extends KV-SSD interface for better transaction support. |
| GPUKV [85] | Allows direct peer-to-peer access between GPU and KV-SSD when the workload is performed on GPU instead of CPU. Reduces I/O overhead due to the data movement from KV-SSD to GPU via user space. |
| KVRAID [133] | Stores small key-value objects by packing them together to reduce the overhead caused by an excessive number of keys associated with small key-value objects. |
| ISKEVA [169] | Extracts video features in FTL and saves the information to KV-SSD, with the extra support of filtering data within the SSD to reduce the data to be transferred to host when performing a query. |
| Dotori [61] | Provides extra KV-SSD features, including transactions, versioning, snapshots, and range queries; the paper also proposed OAK-tree, a new way of organizing B+-trees tailored for KV-SSDs. |

the FTL mapping table, which maps a logical address to a physical address; Meanwhile, key-value databases use a log-like writing mechanism, which is similar to how SSDs perform data writes and updates. It is a natural move to remove the extra layer of overhead for better performance [160].

KAML [79] later extended similar mechanisms to key-value databases. KAML SSD is the shorthand for key-addressable, multi-log SSD, which well summarizes the characteristics of KAML. A library, `libkaml`, provides a list of APIs similar to what traditional key-value databases usually provide, including the support for `read`, `update`, `update`, `commit`, and `abort`. Different namespaces are also supported, which can be considered as tables or files based on demand. `libkaml` then communicates with the device driver and the device to find out the value data page(s) a key is associated with. A key is 64-bit in size, but the data size for the value can vary. This approach removes the requirement of running a database on the host, as applications can directly use the SSD as a key-value database.

Two papers, LOCS [151] and FlashKV [165], couple key-value databases with Open-channel SSDs for better performance. Both are based on LevelDB and provide custom user libraries to support

the key-value database running on top of the host. The libraries cooperate with the Open-channel driver in the kernel to save the actual data on SSD. The main difference between the two is that LOCS utilizes file-level parallelism, whereas FlashKV leverages channel-level parallelism, which shows better read performance when only a single SSTable is accessed [165]. KAML, LOCS, and FlashKV together build the foundation before the first vendor attempt of creating a KV-SSD [89], which provides an even leaner I/O stack than KAML; and unlike LOCS and FlashKV, users do not need to write their own management policies and FTL.

*5.2.2 Internal Performance Improvements.* The creation of KV-SSD allows the SSD to be used directly as a key-value database. However, there is more space for improvement. PinK [74] identifies the probabilistic nature of bloom filters and improves performance by pinning the top levels of LSM-trees. Since bloom filters are probabilistic, it improves the average latency, but it does not improve the tail latency. Reconstructing the bloom filter also results in high CPU overhead, which is less of a problem on a computer with faster CPUs, but it is a more serious problem on KV-SSDs. Instead of using the bloom filter, PinK pins the highest several levels of LSM-trees in the KV-SSD DRAM, which is feasible because the inquiry overhead is bounded by $O(h-1)$, where $h$ is the height of the LSM tree, which is bounded; the size of the top levels of LSM trees, which keep the hottest keys, are also relatively small, as argued by the authors. The tail latency can be improved with these two optimizations tailored for KV-SSDs.

*5.2.3 Other KV-SSD Usage Scenarios.* Although KV-SSDs are for key-value stores intuitively, KEVIN [98] takes a step further by leveraging KV-SSD for a more general use case: It builds a filesystem to be used by any type of application, hence named key-value indexed solid-state drive. By extending the KV-SSD interface with transaction support, the KV-SSD can be used to achieve consistency at the hardware level. Common system calls regarding file/folder creation, e.g., `mkdir()`, `creat()`, `unlink()`, and `readdir()`, are translated into KV-SSD operations including `GET()`, `SET()`, `DELETE()`, and `ITERATE()`, so existing applications do not need to change their code.

While some papers try to generalize the use cases of KV-SSDs, others focus on bringing KV-SSDs to other specific use cases. One use case is programs based on OpenMP with high concurrency. It is not uncommon to have high concurrency for key-value databases. For example, Facebook shows that they have billions of `GET()` requests in a period of 14 days, which translates to at least 800 queries per second [44]. Therefore, a key-value system should be able to handle a vast number of requests concurrently. KV-SiPC [33] provides an approach to address OpenMP workloads with several program threads. Existing applications do not need to modify their code to migrate from traditional block-interface SSDs to KV-SSDs for their applications. KV-SiPC changes the OpenMP internals so it uses the key-value APIs for upper-level applications, and it also adapts the number of parallel compute threads and parallel data access threads based on CPU and SSD utilization.

The existence of KV-SSD reduces the level of the I/O stack from the host to the SSD. However, most papers assume that the workload is running on the CPU. If the workload runs on another device (e.g., GPU), then there is another I/O stack to transfer data to the target device. To further reduce the number of levels in the I/O stack, GPUKV [85] is proposed to allow direct communication between the GPU and the storage. Instead of bringing data from the KV-SSD to the host OS and then from the host OS to the GPU, GPUKV creates the data path from KV-SSD to GPU directly using the PCIe peer-to-peer feature without going through the user space. This approach removes the heavy user space I/O stack for GPU workloads.

Another work, ISKEVA [169], uses KV-SSD as an engine for video metadata. Videos may have metadata associated with the video file itself, and extra features may be added to the file (e.g., if an object exists in a video). A feature extractor is integrated into the SSD FTL, and the extracted

features are saved in the KV-SSD. The KV-SSD with ISKEVA supports extra query flags for data filtering so only filtered results will be returned to the host, eliminating the requirement of the host to perform feature extraction and result filtering.

*5.2.4 Feature Improvements.* Although the creation of KV-SSD led to the first KV-SSD standard [12, 74, 89], interface improvements are brought up for more features. KVMD [129] allows the creation of an array of KV-SSD devices, similar to RAID in traditional block-interface SSDs. However, since data is stored in key-value mappings in KV-SSDs, it is replicated/erasure-coded based on the key of the value. The value of a key can be mapped to several devices for better reliability and performance. However, the approach of using erasure coding for reliability in KVMD causes significant data replication. This is because the erasure coding must be made on the key-value namespace, and the value sizes are usually only several times greater than the key sizes (e.g., smaller than 6:1 and sometimes about 1:1, as reported by Facebook [44]), causing a lot of extra overhead for the data stored. StripeFinder [116] aims to enhance spatial efficiency when using erasure coding. By sharing as much metadata across different keys, it achieves a lower spatial overhead with a value-to-key ratio around 12:1 compared to KVMD. However, the author concludes that even with StripeFinder, the overhead of using erasure coding remains significant for realistic workloads, and it is better to just use replication in this case. KVRAID [133] further addresses the issue. Since small and large data objects have similar IOPS, but the metadata overhead associated with smaller data objects is greater than larger data objects, KVRAID packs several small logical data objects from the host into a large physical data object to mitigate the metadata overhead. The data will be written to the device when a sufficient number of objects are accumulated in SSD DRAM; however, to bound the I/O latency, there is also a timeout mechanism where the currently accumulated data will be written to the device despite the number of objects accumulated. This approach efficiently stores small objects (i.e., 128 to 4,096 bytes) on KV-SSD arrays.

Dotori [61] provides more features to the KV-SSD, including transactions, versioning, snapshots, and range queries. It also provides better indexing support for KV-SSD by using their proposed OAK-tree, a B+-tree tailored for KV-SSDs. The authors also call for the standardization of these features by implementing them in SSD (instead of on-host like Dotori) and extending the KV-SSD interface. Features like transactions are already used by some prior works (e.g., KEVIN [98]), showing the usefulness of these features for extended KV-SSD use cases.

## 6 CXL and Byte-addressable SSD

With the increasing adoptions of data-intensive applications, including large language models and data analytics, the memory capacity has become a significant bottleneck for such workloads due to high DRAM price and the limited number of DIMM slots for DRAM [24]. This problem, known as the *memory wall*, causes extra memory management overhead for developers when developing their models and applications [162]. Recently, a new emerging technology named **Compute Express Link (CXL)** provides a new unified architecture to overcome the limitations caused by the memory wall. By directly accessing PCIe-attached SSDs with load/store instructions, the CPU can directly use SSDs as a part of the main memory [11]. A significant challenge for CXL-enabled SSD is the ability to be byte-addressable [82]. The granularity for memory access is bytes, whereas the granularity for SSD access is based on the traditional block abstraction. This granularity mismatch causes traffic amplification due to the SSD read/write granularity in the read-modify-write cycle [162].

Thankfully, Samsung created a new type of SSD named 2B-SSD with two different allowed access granularities: Byte and Block [29]. With the ability to allow direct byte-level access using SSD internal DRAM, the traffic amplification caused by the granularity mismatch of NAND flash and

Table 6. Comparison between Several Related SSD Interface Enhancements for Data Placement[64, 131]

| | Traditional | Multi-stream | FDP | ZNS | OC |
|---|---|---|---|---|---|
| Backward Compatibility | ✓ | ✓ | ✓ | × | × |
| Random Writes Allowed | ✓ | ✓ | ✓ | × (✓ in part) | ✓ |
| Data Placement | SSD | SSD/Host directive | SSD/Host directive | Host to zone | Host to channel |
| Garbage Collection | SSD | SSD | SSD | Host | Host |
| Wear Leveling | SSD | SSD | SSD | SSD | Host |
| Error Correction | SSD | SSD | SSD | SSD | SSD |
| Host Change Needed | N/A | Minor | Some | Major | Significant |

DRAM can be reduced [162]. Although the development of CXL is still at an early stage, both academia and industry show interest in combining CXL and byte-addressable SSDs [82, 101, 162]. In summary, The dual of byte-addressable SSD and CXL marks a new page for SSD and storage abstraction. The block interface in the storage abstraction for the past decades has been overthrown, and the use of SSD has been extended to overcome the capacity limit of main memory. We hope future research can lead to even more exciting use cases of byte-addressable SSDs.

## 7 The Future

So far, we have discussed several enhancements to storage abstractions, including TRIM, Multi-stream, FDP, Open-channel SSD, ZNS, computational storage, KV-SSD, and byte-addressable SSD. TRIM is widely adopted by modern SSDs. Multi-stream and Open-channel SSD are now obsolete in practice. ZNS, computational storage, and KV-SSD are established research areas. FDP and byte-addressable SSD with CXL are on the rise. In this section, we would like to provide our thoughts on the following questions:

— What can we learn from the (partially) failed attempts?
— What can we do to further the research in well-established research areas?
— What should we do to ensure the success of existing and new storage abstraction enhancements?

### 7.1 Learning from (Partially) Failed Attempts

We have discussed two partially failed enhancements in the survey, namely, Multi-stream and Open-channel SSD. These two enhancements were not able to gain widespread acceptance in academia, the open-source community, or industry. Interestingly, both are at the two different extremes of complexity, as shown in Table 6. Multi-stream only requires a single lifetime hint based on the workload characteristics from the host, whereas Open-channel SSD requires an extensive modification on the host layer to determine the location of data placement. We believe the lesson here is that if an enhancement to the storage abstraction requires user intervention to enable, then it should provide a balanced flexibility for users to control.

However, TRIM is an even simpler enhancement compared to Multi-stream, but the adoption of TRIM is widespread. One possible reason is that the responsible layer for issuing TRIM requests is more clear. TRIM should be issued when an LBA is considered invalid. It can be assumed that most users use their storage device with a filesystem, and the filesystem knows if an LBA is invalid, since this is part of the filesystem's responsibility [143]. Application developers do not need to implement TRIM support on their side. SSDs with TRIM enabled show significantly improved performance, albeit the concept of TRIM is simple [86]. Multi-stream, however, can be implemented in layers including the application layer [86], the file system layer [134], and the driver layer [158]. There is no unified consensus on which layer is responsible for implementing write hints. Nevertheless, this problem is partially solved in smartphones, where the device vendor has tighter control of

the whole system than computers [90]. The underlying storage is also tightly coupled with the device and the system: Almost every hardware component is unremovable, so the smartphone vendors can directly provide drivers tailored for the device, e.g., storage driver with write hints. Although the adoption rate of Multi-stream is almost zero on computers, it has more popularity on smartphones, as vendors have reported [105, 130, 140].

It is also worth noting that Multi-stream and Open-channel, despite their low adoption rates, also show their impact on FDP and ZNS. We see many similarities between Multi-stream and FDP, and FDP can be somehow considered as the successor of Multi-stream [137]; the initial ZNS evaluation paper rehashes policies and code from Multi-stream [36]. ZNS is also the direct successor of Open-channel SSD, as announced on the Open-channel SSD official website [7]. FDP also provides the flexibility of data placement similar to Open-channel SSD without the extra management overhead [137]. In summary, although Multi-stream and Open-channel are obsolete, they show their influence on enhancements to come.

**Lessons Learned:** An enhancement should balance simplicity and configurability. The layer for implementing the enhancement should be clearly defined, and the enhancement should be transparent from the application developer's point of view. A new enhancement is more likely to succeed in a tightly controlled environment, e.g., smartphones and enterprise systems, where the controller of the system (i.e., smartphone vendor and data center administrator) can apply the enhancement between the application layer and the storage device with enough justification to design related code and apply the enhancements. Last, the sunset of an enhancement can be the sunrise of a better enhancement in the future.

### 7.2 Expanding Research Scopes for Well-established Enhancements

We consider ZNS, KV-SSD, and computational storage to be well-established research areas. ZNS and KV-SSD have been developed and researched for about half a decade, whereas computational storage has been researched for decades, since the HDD era. Further expanding research scopes for these areas can be challenging but rewarding.

We identify two possible methods to expand the research scope: the first is to apply existing methods from other research areas. For example, applying methods used in Multi-stream to ZNS could lead to new discoveries [36]. Another possible way is to discover new use cases for an existing technology. An example would be applying ZNS to cases other than key-value databases. ZNS has a close connection to key-value databases since its birth [36], and our survey shows the most common topic for ZNS-related paper is also key-value databases [75, 81, 102, 103, 112, 115]. However, researchers are applying ZNS SSDs to more scenarios, including Linux swap space [32] and general filesystem [132]. A possible future direction can be integrating ZNS SSDs or KV-SSDs with CXL, since CXL is on the rise. Last, an enhancement must be standardized to be adopted by users. If an enhancement is well-studied but yet to be standardized (e.g., computational storage), then a retrospective review can be helpful for a better design of the standard.

### 7.3 Ensuring Future Success of Established and New Enhancements

An established enhancement must continuously adapt to the ever-changing demands to keep competitiveness. Any enhancement may have its limitations, and it is essential to address them in time; otherwise, people may shift to new or other enhancements. One effective approach is to learn from the experiences and challenges of similar enhancements. For example, we have discussed the limitations and similarities of **Zoned Namespace (ZNS)** compared to Multi-stream technology. ZNS zones are designed to be append-only and lack random write support. Although some ZNS SSDs provide conventional zones that support random writes [58, 59], this provision has proven insufficient. To address this, the new NVMe technical proposal, TP4076, has been ratified to

allow random-writable regions within otherwise sequential-write-only zones [41]. Additionally, TP4093 has been ratified to enable the passing of lifetime hints to ZNS zone writes [142], efficiently integrating Multi-stream functionality within ZNS zones.

Established enhancements can also be extended to complement other protocols or storage technologies to attract a broader user base. Multi-stream technology, for instance, has gained traction in the UFS domain, with smartphone storage vendors expressing interest [105, 130, 140] although it was not able to gain enough attention on the NVMe end. Meanwhile, zoned storage has garnered attention in the UFS domain and was officially incorporated into the UFS standard in November 2023 [63, 77]. A paper named ZMS from Samsung discusses the I/O stack of applying ZNS to mobile storage with the UFS interface [72]. It is important to note that the concept of ZNS SSDs was originally developed from **Shingled Magnetic Recording (SMR)** HDDs. In summary, learning from existing technologies can lead to the development of new extensions to storage abstractions, which could subsequently be adopted by other sectors to achieve a broader impact.

Conversely, a new enhancement must demonstrate sufficient demand to be successful, with demand primarily driven by SSD users rather than vendors. Retrospectively, both Multi-stream and Open-channel SSDs were vendor-controlled: Multi-stream by Samsung [86] and Open-channel SSD by CNEX Labs [38]. Despite their demonstrated benefits, the real demand for these enhancements was uncertain at the time of their creation, and the number of developers willing to integrate these enhancements into their applications was unknown. In contrast, the **Flexible Data Placement (FDP)** and **Compute Express Link (CXL)** technologies were developed based on clear user demand. FDP was created by Google and Meta [137], while the CXL consortium includes major industry players such as Google, Meta, Microsoft, Dell, HP, Nvidia, and Alibaba [49]. These enhancements have secured their markets since their inception, leading to a higher likelihood of future success.

Exploring the potential for storage abstraction extensions to enter consumer markets can also drive increased demand. Zoned storage, for instance, has been successfully integrated into the UFS standard, which is predominantly used in smartphones. FDP and CXL can adopt similar strategies to target consumer markets. For FDP, gaining filesystem support is a prerequisite to ensure seamless integration without requiring user or application-side configuration. An FDP-enabled SSD with filesystem support should exhibit similar or better **write amplification factor (WAF)** and SSD longevity compared to FStream [134], a worthy pursuit given the declining trend of maximum SSD program-erase cycles [91, 118]. CXL may also find applications in personal computers or even smartphones, especially as interest grows in on-device machine learning model training [76, 141, 156]. The limited resources of personal computers or smartphones present a significant opportunity for CXL to overcome the memory wall on these devices.

With backing from both creators and users, FDP and CXL hold significant potential from their inception. However, numerous questions remain before their widespread adoption can be realized. Addressing these questions can provide valuable insights for other researchers in the field. For instance, conducting feasibility studies can identify the best use cases for these new enhancements [82, 162]. Another approach is to analyze similar past enhancements and apply those insights to the current context. Comparing related enhancements can also help determine if the new enhancement offers greater benefits. By thoroughly studying feasibility, benefits, and demand, we hope that FDP and CXL will achieve widespread industry adoption in the near future.

### 7.4 Future Vision for SSD Abstraction Enhancements

While the article has extensively covered the past and present SSD interface enhancements, envisioning the future of SSD technology is equally crucial. We would like to end this article with the following potential advancements that could shape the next generation of SSDs. We hope these

directions can kickstart research interests and provide better storage interfaces and abstractions in the future.

*7.4.1 Learning from Universal Flash Storage.* UFS has been a critical interface for mobile and embedded storage solutions. Mobile and embedded devices are more energy-aware than traditional computers, which means UFS has to be energy-efficient. While focusing on performance improvements, new UFS standards also aim for lower power consumption. For example, UFS 4.0 has a power efficiency that is reportedly 46% better than UFS 3.1 while doubling the read performance [138]. Although our article mostly focuses on the storage interface for traditional computers (especially NVMe), we are seeing more interactions between UFS and NVMe where one learns from the other (e.g., integrating Multi-stream and ZNS support on UFS). As UFS is mostly used for mobile devices, it has to be power-smart; storage protocols for traditional computers may also learn from UFS in the future in light of more concerns about energy efficiency.

*7.4.2 Integrating Other Enhancements with Compute Express Link.* CXL breaks the memory wall by utilizing SSDs as main memory. Although CXL is now a popular topic, there are still many questions to answer. SSDs have internal activities such as garbage collection and wear leveling, which occupy internal bandwidth and CPU resources [78, 167]. Without the communication between the CXL components and the SSD, CXL performance may be severely impacted by these SSD-internal tasks. SSDs are inherently slower than DRAMs already, and researchers should do anything they can to make SSDs faster when being used as main memory. A possible direction is to have special storage abstraction extensions so CXL components may coordinate with SSDs to prevent these internal tasks when possible. A similar design has been done for SSD RAID systems [109], and we believe such a design will be strongly beneficial for better CXL performance. CXL also supports different interleave granularities and interleave ways when placing data within an address range [20]. Researching the impact of different interleave granularities/ways is also needed, since they can result in tradeoffs similar to what one can see from the design of cache systems.

Another possible research direction is to directly integrate SmartSSDs with CXL. CXL Consortium defined three different types of CXL devices; the CXL Type 2 (T2) devices are considered to be accelerator-like devices with their own memory that hosts can access [40]. This is coincidentally similar to SmartSSDs, which have the ability to perform calculations on the data they store. The host can then offload some time-insensitive tasks to the CXL-enabled SmartSSD when needed.

Flexible Data Placement and Zoned Namespace Storage may also help on CXL. We have discussed the advantages brought by ZNSwap, which uses ZNS SSDs for memory swap space [31, 32]. By placing data carefully, FDP and ZNS devices should perform better than traditional SSDs. Performance isolation between different tenants may also be achieved with FDP. Considering a CXL memory pool shared by multiple tenants, separating data from different tenants will be beneficial for better garbage collection efficiency and performance isolation between different tenants. This could be a huge task, because CXL systems can be complex, and there may also be different design choices (e.g., segregation by program context [96], process, or (virtual) machine). Still, this approach should be able to improve performance and reduce disturbances from noisy neighbors nonetheless.

## 8 Conclusion

In this survey, we discussed the shortcomings of the traditional storage abstraction and explored several enhancements that address them. We categorized them into four different categories, each with a different philosophy. For each enhancement, we discussed its history and relationships with other enhancements from various perspectives, including source code interpretation and design concept comparison, along with the ecosystem and research efforts made by both industry and

academia. Finally, we identified the future for existing and emerging enhancements by reflecting on partially failed attempts and proposing possible new research directions. We hope this article lays a cornerstone for exploring the current landscape and inspires future research on enhancements to the SSD storage abstraction.

## References


[1] 2007. Notification of Deleted Data Proposal for ATA8-ACS2. Retrieved from https://t13.org/system/files/Documents/2007/e07154r0-Notification%20for%20Deleted%20Data%20Proposal%20for%20ATA-ACS2_2.doc

[2] 2009. Serial ATA: Meeting Storage Needs Today and Tomorrow. Retrieved from https://web.archive.org/web/20120417133358/http://www.serialata.org/documents/SATA-Rev-30-Presentation.pdf

[3] 2011. NVMe 1.0. Retrieved from https://nvmexpress.org/wp-content/uploads/NVM-Express-1_0e.pdf

[4] 2012. NVMe 1.1. Retrieved from https://nvmexpress.org/wp-content/uploads/NVM-Express-1_1b-1.pdf

[5] 2014. NVMe 1.2. Retrieved from https://nvmexpress.org/wp-content/uploads/NVM-Express-1_2a.pdf

[6] 2015. lightnvm: Support for Open-Channel SSDs. Retrieved from https://github.com/torvalds/linux/commit/cd9e9808d18fe7107c306f6e71c8be7230ee42b4

[7] 2015. OpenChannelSSD. Retrieved from http://lightnvm.io

[8] 2017. f2fs: apply write hints to select the type of segments for buffered write. Retrieved from https://github.com/torvalds/linux/commit/a02cd4229e298aadbe8f5cf286edee8058d87116

[9] 2017. Merge branch "for-4.13/block" of git://git.kernel.dk/linux-block. Retrieved from https://github.com/torvalds/linux/commit/c6b1e36c8fa04a6680c44fe0321d0370400e90b6

[10] 2017. NVMe 1.3. Retrieved from https://nvmexpress.org/wp-content/uploads/NVM-Express-1_3a-20171024_ratified.pdf

[11] 2019. Compute Express Link. Retrieved from https://docs.wixstatic.com/ugd/0c1418_d9878707bbb7427786b70c3c91d5fbd1.pdf

[12] 2019. Key Value Storage API Specification Version 1.0. Retrieved from https://www.snia.org/sites/default/files/technical-work/kvsapi/release/SNIA-Key-Value-Storage-API-v1.0.pdf

[13] 2019. NVMe 1.4. Retrieved from https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf

[14] 2020. Compute Express Link. Retrieved from https://www.computeexpresslink.org/_files/ugd/0c1418_14c5283e7f3e40f9b2955c7d0f60bebe.pdf

[15] 2020. Key Value Storage API Specification Version 1.1. Retrieved from https://www.snia.org/sites/default/files/technical-work/kvsapi/release/SNIA-Key-Value-Storage-API-v1.1.pdf

[16] 2021. NVM Express Key-Value Command Set Specification 1.0. Retrieved from https://nvmexpress.org/wp-content/uploads/NVM-Express-Key-Value-Command-Set-Specification-1.0-2021.06.02-Ratified-1.pdf

[17] 2021. NVM Express® Base Specification 2.0. Retrieved from https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-2_0-2021.06.02-Ratified-5.pdf

[18] 2022. Compute Express Link. Retrieved from https://www.computeexpresslink.org/_files/ugd/0c1418_a8713008916044ae9604405d10a7773b.pdf

[19] 2022. Hyperscale Innovation: Flexible Data Placement Mode (FDP). Retrieved from https://nvmexpress.org/wp-content/uploads/Hyperscale-Innovation-Flexible-Data-Placement-Mode-FDP.pdf

[20] 2023. Compute Express Link. Retrieved from https://www.computeexpresslink.org/_files/ugd/0c1418_6ede12bda4d34ffeb879c3700dde38f9.pdf

[21] 2023. NVM-Express-NVM-Command-Set-Specification-1.0d-2023.12.28-Ratified.pdf. Retrieved from https://nvmexpress.org/wp-content/uploads/NVM-Express-NVM-Command-Set-Specification-1.0d-2023.12.28-Ratified.pdf

[22] 2024. Retrieved from https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-2_0-2021.06.02-Ratified-5.pdf

[23] 2024. rocksdb/db/column_family.cc at daf06f1361140bb4ca9304b27b0aa36ef5842f56 - facebook/rocksdb. Retrieved from https://github.com/facebook/rocksdb/blob/daf06f1361140bb4ca9304b27b0aa36ef5842f56/db/column_family.cc

[24] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. 2019. FlatFlash: Exploiting the byte-accessibility of SSDs within a unified memory-storage hierarchy. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. Association for Computing Machinery, 971–985. DOI: https://doi.org/10.1145/3297858.3304061

[25] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2023. *Operating Systems: Three Easy Pieces* (1.10 ed.). Arpaci-Dusseau Books.

[26] Jens Axboe. 2008. Add “discard” request handling · torvalds/linux@fb2dce8. Retrieved from https://github.com/torvalds/linux/commit/fb2dce862d9f9a68e6b9374579056ec9eca02a63

[27] Jens Axboe. 2021. Merge branch “nvme-5.9” of git://git.infradead.org/nvme into for-5.9/.... Retrieved from https://github.com/torvalds/linux/commit/80ee071b18660c56d3f7a7ab67793b78a96baae5

[28] Jens Axboe. 2022. Re: [EXT] Re: [PATCH 2/2] block: remove the per-bio/request write hint. - Jens Axboe. Retrieved from https://lore.kernel.org/all/800fa121-5da2-e4c0-d756-991f007f0ad4@kernel.dk/

[29] Duck-Ho Bae, Insoon Jo, Youra Adel Choi, Joo-Young Hwang, Sangyeun Cho, Dong-Gi Lee, and Jaeheon Jeong. 2018. 2B-SSD: The case for dual, byte-and block-addressable solid-state drives. In *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*. IEEE, 425–438. Retrieved from https://ieeexplore.ieee.org/abstract/document/8416845

[30] Hanyeoreum Bae, Jiseon Kim, Miryeong Kwon, and Myoungsoo Jung. 2022. What you can't forget: Exploiting parallelism for zoned namespaces. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'22)*. Association for Computing Machinery, 79–85. DOI: https://doi.org/10.1145/3538643.3539744

[31] Shai Bergman, Niklas Cassel, Matias Bjørling, and Mark Silberstein. 2022. ZNSwap: un-block your swap. In *Proceedings of the USENIX Annual Technical Conference (ATC'22)*. USENIX Association, 1–18. Retrieved from https://www.usenix.org/conference/atc22/presentation/bergman

[32] Shai Bergman, Niklas Cassel, Matias Bjørling, and Mark Silberstein. 2023. ZNSwap: Un-block your swap. *ACM Trans. Stor.* 19, 2, Article 12 (03 2023), 25 pages. DOI: https://doi.org/10.1145/3582434

[33] Janki Bhimani, Jingpei Yang, Ningfang Mi, Changho Choi, Manoj Saha, and Adnan Maruf. 2021. Fine-grained control of concurrency within KV-SSDs. In *Proceedings of the 14th ACM International Conference on Systems and Storage (SYSTOR'21)*. Association for Computing Machinery, Article 4, 12 pages. DOI: https://doi.org/10.1145/3456727.3463777

[34] Janki Bhimani, Zhengyu Yang, Jingpei Yang, Adnan Maruf, Ningfang Mi, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. 2022. Automatic stream identification to improve flash endurance in data centers. *ACM Trans. Stor.* 18, 2, Article 17 (04 2022), 29 pages. DOI: https://doi.org/10.1145/3470007

[35] Matias Bjørling. 2019. From open-channel SSDs to zoned namespaces *(VAULT'19)*. USENIX Association. Retrieved from https://www.usenix.org/conference/vault19/presentation/bjorling

[36] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. 2021. ZNS: Avoiding the block interface tax for flash-based SSDs. In *Proceedings of the USENIX Annual Technical Conference (ATC'21)*. USENIX Association, 689–703. Retrieved from https://www.usenix.org/conference/atc21/presentation/bjorling

[37] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. 2013. Linux block IO: Introducing multi-queue SSD access on multi-core systems. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR'13)*. Association for Computing Machinery, Article 22, 10 pages. DOI: https://doi.org/10.1145/2485732.2485740

[38] Matias Bjørling, Javier González, and Philippe Bonnet. 2017. LightNVM: The Linux open-channel SSD subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. USENIX Association, 359–374. Retrieved from https://www.usenix.org/conference/fast17/technical-sessions/presentation/bjorling

[39] Simona Boboila, Youngjae Kim, Sudharshan S. Vazhkudai, Peter Desnoyers, and Galen M. Shipman. 2012. Active Flash: Out-of-core data analytics on flash storage. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*. 1–12. DOI: https://doi.org/10.1109/MSST.2012.6232366

[40] Kurtis Bowman. 2023. Compute Express Link (CXL) Device Ecosystem and Usage Models. Retrieved from https://computeexpresslink.org/wp-content/uploads/2023/12/CXL_FMS-Panel-2023_FINAL.pdf

[41] Judy Brock, Bill Martin, Javier Gonzalez, Klaus B. Jensen, Fred Knight, Yoni Shternhell, Matias Bjørling, and Paul Suhler. 2022. TP4076a Zoned Random Write Area 2022.01.19 Ratified. Retrieved from https://nvmexpress.org/wp-content/uploads/NVM-Express-2.0-Ratified-TPs_20230111.zip

[42] Neil Brown. 2017. A block layer introduction part 1: The bio layer [LWN.net]. Retrieved from https://lwn.net/Articles/736534/

[43] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan et al. 2020. 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 (FAST'20)*. USENIX Association, 29–41. Retrieved from https://www.usenix.org/conference/fast20/presentation/cao-wei

[44] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*. USENIX Association, 209–223. Retrieved from https://www.usenix.org/conference/fast20/presentation/cao-zhichao

[45] Marc Carino. 2013. [PATCH v3 0/3] Introduce new SATA queued commands - Marc C. Retrieved from https://lore.kernel.org/all/1376023752-3105-1-git-send-email-marc.ceeeee@gmail.com/

[46] Chandranil Chakraborttii and Heiner Litz. 2021. Reducing write amplification in flash by death-time prediction of logical block addresses. In *Proceedings of the 14th ACM International Conference on Systems and Storage (SYSTOR'21)*. Association for Computing Machinery, Article 11, 12 pages. DOI: https://doi.org/10.1145/3456727.3463784

[47] Da-Wei Chang, Hsin-Hung Chen, and Wei-Jian Su. 2015. VSSD: Performance isolation in a solid-state drive. *ACM Trans. Des. Autom. Electron. Syst.* 20, 4, Article 51 (09 2015), 33 pages. DOI: https://doi.org/10.1145/2755560

[48] Young-in Choi and Sungyong Ahn. 2022. Separating the file system journal to reduce write amplification of garbage collection on ZNS SSDs. *J. Multim. Inf. Syst.* 9, 4 (2022), 261–268. DOI: https://doi.org/10.33851/JMIS.2022.9.4.261

[49] CXL Consortium. 2023. Our Members - Compute Express Link. Retrieved from https://computeexpresslink.org/our-members/

[50] Jonathan Corbet. 2010. The best way to throw blocks away [LWN.net]. Retrieved from https://lwn.net/Articles/417809/

[51] Lukas Czerner. 2010. [PATCH 0/3 v. 8] Ext3/Ext4 Batched discard support - Lukas Czerner. Retrieved from https://lore.kernel.org/linux-ext4/1285342559-16424-1-git-send-email-lczerner@redhat.com/

[52] Lukas Czerner. 2010. [PATCH 1/3] Add ioctl FITRIM. - Lukas Czerner. Retrieved from https://lore.kernel.org/linux-ext4/1281094276-11377-2-git-send-email-lczerner@redhat.com/

[53] Emily Desjardins. 2011. JEDEC Announces Publication of Universal Flash Storage (UFS) Standard | JEDEC. Retrieved from https://www.jedec.org/news/pressreleases/jedec-announces-publication-universal-flash-storage-ufs-standard

[54] Peter Desnoyers. 2014. Analytic models of SSD write performance. *ACM Trans. Stor.* 10, 2, Article 8 (03 2014), 25 pages. DOI: https://doi.org/10.1145/2577384

[55] Sarah M. Diesburg and An-I Andy Wang. 2010. A survey of confidential data storage and deletion methods. *ACM Comput. Surv.* 43, 1, Article 2 (12 2010), 37 pages. DOI: https://doi.org/10.1145/1824795.1824797

[56] Western Digital. 2023. linux/fs/zonefs/zonefs.h at f2661062f16b2de5d7b6a5c42a9a5c96326b8454 · torvalds/linux. Retrieved from https://github.com/torvalds/linux/blob/f2661062f16b2de5d7b6a5c42a9a5c96326b8454/fs/zonefs/zonefs.h#L126

[57] Western Digital. 2024. btrfs | Zoned Storage. Retrieved from https://zonedstorage.io/docs/filesystems/btrfs

[58] Western Digital. 2024. f2fs | Zoned Storage. Retrieved from https://zonedstorage.io/docs/filesystems/f2fs

[59] Western Digital. 2024. ZoneFS | Zoned Storage. Retrieved from https://zonedstorage.io/docs/filesystems/zonefs

[60] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query processing on smart SSDs: Opportunities and challenges. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. Association for Computing Machinery, 1221–1230. DOI: https://doi.org/10.1145/2463676.2465295

[61] Carl Duffy, Jaehoon Shim, Sang-Hoon Kim, and Jin-Soo Kim. 2023. Dotori: A key-value SSD based KV store. *Proc. VLDB Endow.* 16, 6 (02 2023), 1560–1572. DOI: https://doi.org/10.14778/3583140.3583167

[62] Jake Edge. 2019. Issues around discard [LWN.net]. Retrieved from https://lwn.net/Articles/787272/

[63] Jake Edge. 2023. Zoned storage and filesystems [LWN.net]. Retrieved from https://lwn.net/Articles/932748/

[64] Javier González. 2023. FDP and ZNS for NAND Data Placement: Landscape, Trade-Offs, and Direction. In *Proceedings of the Flash Memory Summit*. Retrieved from https://www.flashmemorysummit.com/English/Conference/Program_at_a_Glance_Tue.html

[65] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho et al. 2016. Biscuit: A framework for near-data processing of big data workloads. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*. IEEE Press, 153–165. DOI: https://doi.org/10.1109/ISCA.2016.23

[66] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. 2021. ZNS+: Advanced zoned namespace interface for supporting in-storage zone compaction. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*. USENIX Association, 147–162. Retrieved from https://www.usenix.org/conference/osdi21/presentation/han

[67] Yejin Han, Myunghoon Oh, Jaedong Lee, Seehwan Yoo, Bryan S. Kim, and Jongmoo Choi. 2023. Achieving performance isolation in Docker environments with ZNS SSDs. In *Proceedings of the IEEE 12th Non-volatile Memory Systems and Applications Symposium (NVMSA'23)*. 25–31. DOI: https://doi.org/10.1109/NVMSA58981.2023.00016

[68] Niclas Hedam, Morten Tychsen Clausen, Philippe Bonnet, Sangjin Lee, and Ken Friis Larsen. 2023. Delilah: eBPF-offload on computational storage. In *Proceedings of the 19th International Workshop on Data Management on New Hardware (DaMoN'23)*. 70–76. Retrieved from https://dl.acm.org/doi/abs/10.1145/3592980.3595319

[69] Christoph Hellwig. 2022. Merge tag "for-5.18/write-streams-2022-03-18" of git://git.kernel.dk/.... Retrieved from https://github.com/torvalds/linux/commit/561593a048d7d6915889706f4b503a65435c033a

[70] Hans Holmberg. 2021. Initial commmit for ZenFS - westerndigitalcorporation/zenfs@7f8e885. Retrieved from https://github.com/westerndigitalcorporation/zenfs/commit/7f8e885d670205cfdc91a8b2b34ca5b492f42d43

[71] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. 2017. FlashBlox: Achieving both performance isolation and uniform lifetime for virtualized SSDs. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. USENIX Association, 375–390. Retrieved from https://www.usenix.org/conference/fast17/technical-sessions/presentation/huang

[72] Joo-Young Hwang, Seokhwan Kim, Daejun Park, Yong-Gil Song, Junyoung Han, Seunghyun Choi, Sangyeun Cho, and Youjip Won. 2024. ZMS: Zone abstraction for mobile flash storage. In *Proceedings of the USENIX Annual Technical Conference (ATC'24)*. USENIX Association, 173–189. Retrieved from https://www.usenix.org/conference/atc24/presentation/hwang

[73] Choulseung Hyun, Jongmoo Choi, Donghee Lee, and Sam H. Noh. 2011. To TRIM or not to TRIM: Judicious trimming for solid state drives. In *Poster presentation in the 23rd ACM Symposium on Operating Systems Principles (SIGOPS'11)*. Retrieved from https://sigops.org/s/conferences/sosp/2011/posters/summaries/sosp11-final16.pdf

[74] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. 2020. PinK: High-speed in-storage key-value store with bounded tails. In *Proceedings of the USENIX Annual Technical Conference (ATC'20)*. USENIX Association, 173–187. Retrieved from https://www.usenix.org/conference/atc20/presentation/im

[75] Minwoo Im, Kyungsu Kang, and Heonyoung Yeom. 2022. Accelerating RocksDB for small-zone ZNS SSDs by parallel I/O mechanism. In *Proceedings of the 23rd International Middleware Conference Industrial Track (Middleware'22)*. Association for Computing Machinery, 15–21. DOI: https://doi.org/10.1145/3564695.3564774

[76] Apple Inc. 2024. Introducing Apple's On-device and Server Foundation Models - Apple Machine Learning Research. Retrieved from https://machinelearning.apple.com/research/introducing-apple-foundation-models

[77] JEDEC. 2023. Zoned Storage for UFS | JEDEC. Retrieved from https://www.jedec.org/standards-documents/docs/jesd220-5

[78] Ziyang Jiao, Janki Bhimani, and Bryan S. Kim. 2022. Wear leveling in SSDs considered harmful. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'22)*. Association for Computing Machinery, 72–78. DOI: https://doi.org/10.1145/3538643.3539750

[79] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. 2017. KAML: A flexible, high-performance key-value SSD. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*. 373–384. DOI: https://doi.org/10.1109/HPCA.2017.15

[80] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. 2016. YourSQL: A high-performance database system leveraging in-storage computing. *Proc. VLDB Endow.* 9, 12 (08 2016), 924–935. DOI: https://doi.org/10.14778/2994509.2994512

[81] Jeeyoon Jung and Dongkun Shin. 2022. Lifetime-leveling LSM-tree compaction for ZNS SSD. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'22)*. Association for Computing Machinery, 100–105. DOI: https://doi.org/10.1145/3538643.3539741

[82] Myoungsoo Jung. 2022. Hello bytes, bye blocks: PCIe storage meets compute express link for memory expansion (CXL-SSD). In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'22)*. Association for Computing Machinery, 45–51. DOI: https://doi.org/10.1145/3538643.3539745

[83] Myoungsoo Jung and Mahmut Kandemir. 2013. Revisiting widely held SSD expectations and rethinking system-level implications. *SIGMETRICS Perform. Eval. Rev.* 41, 1 (06 2013), 203–216. DOI: https://doi.org/10.1145/2494232.2465548

[84] Myoungsoo Jung and Mahmut Kandemir. 2013. Revisiting widely held SSD expectations and rethinking system-level implications. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'13)*. Association for Computing Machinery, 203–216. DOI: https://doi.org/10.1145/2465529.2465548

[85] Min-Gyo Jung, Chang-Gyu Lee, Donggyu Park, Sungyong Park, Jungki Noh, Woosuk Chung, Kyoung Park, and Youngjae Kim. 2021. GPUKV: An integrated framework with KVSSD and GPU through P2P communication support. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing (SAC'21)*. Association for Computing Machinery, 1156–1164. DOI: https://doi.org/10.1145/3412841.3441990

[86] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. 2014. The multi-streamed solid-state drive. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'14)*. USENIX Association. Retrieved from https://www.usenix.org/conference/hotstorage14/workshop-program/presentation/kang

[87] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. 2013. X-FTL: Transactional FTL for SQLite databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. Association for Computing Machinery, 97–108. DOI: https://doi.org/10.1145/2463676.2465326

[88] Yangwook Kang, Yang-suk Kee, Ethan L. Miller, and Chanik Park. 2013. Enabling cost-effective data processing with smart SSD. In *Proceedings of the IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST'13)*. 1–12. DOI: https://doi.org/10.1109/MSST.2013.6558444

[89] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel D. G. Lee. 2019. Towards building a high-performance, scale-in key-value storage system. In *Proceedings*

*of the 12th ACM International Conference on Systems and Storage (SYSTOR'19)*. Association for Computing Machinery, 144–154. DOI: https://doi.org/10.1145/3319647.3325831

[90] Yunji Kang and Dongkun Shin. 2021. mStream: Stream management for mobile file system using Android file contexts. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing (SAC'21)*. Association for Computing Machinery, 1203–1208. DOI: https://doi.org/10.1145/3412841.3442115

[91] Bryan S. Kim, Jongmoo Choi, and Sang Lyul Min. 2019. Design tradeoffs for SSD reliability. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. USENIX Association, 281–294. Retrieved from https://www.usenix.org/conference/fast19/presentation/kim-bryan

[92] Jaegeuk Kim. 2015. [PATCH 5/5] f2fs: introduce a batched trim - Jaegeuk Kim. Retrieved from https://lore.kernel.org/all/1422401503-4769-5-git-send-email-jaegeuk@kernel.org/

[93] Joohyun Kim, Haesung Kim, Seongjin Lee, and Youjip Won. 2010. FTL design for TRIM command. In *Proceedings of the 5th International Workshop on Software Support for Portable Storage (IWSSPS'10)*. 7–12.

[94] Jaeho Kim, Donghee Lee, and Sam H. Noh. 2015. Towards SLO complying SSDs through OPS isolation. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, 183–189. Retrieved from https://www.usenix.org/conference/fast15/technical-sessions/presentation/kim_jaeho

[95] Taejin Kim, Sangwook Shane Hahn, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. 2018. PCStream: Automatic stream allocation using program contexts. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'18)*. USENIX Association. Retrieved from https://www.usenix.org/conference/hotstorage18/presentation/kim-taejin

[96] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. 2019. Fully automatic stream management for multi-streamed SSDs using program contexts. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. USENIX Association, 295–308. Retrieved from https://www.usenix.org/conference/fast19/presentation/kim-taejin

[97] Thomas Kim, Jekyeom Jeon, Nikhil Arora, Huaicheng Li, Michael Kaminsky, David G. Andersen, Gregory R. Ganger, George Amvrosiadis, and Matias Bjørling. 2023. RAIZN: Redundant array of independent zoned namespaces. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS'23)*. Association for Computing Machinery, 660–673. DOI: https://doi.org/10.1145/3575693.3575746

[98] Jinhyung Koo, Junsu Im, Jooyoung Song, Juhyung Park, Eunji Lee, Bryan S. Kim, and Sungjin Lee. 2021. Modernizing file system through in-storage indexing. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*. USENIX Association, 75–92. Retrieved from https://www.usenix.org/conference/osdi21/presentation/koo

[99] Dongup Kwon, Dongryeong Kim, Junehyuk Boo, Wonsik Lee, and Jangwoo Kim. 2021. A fast and flexible hardware-based virtualization mechanism for computational storage devices. In *Proceedings of the USENIX Annual Technical Conference (ATC'21)*. USENIX Association, 729–743. Retrieved from https://www.usenix.org/conference/atc21/presentation/kwon

[100] Miryeong Kwon, Donghyun Gouk, Changrim Lee, Byounggeun Kim, Jooyoung Hwang, and Myoungsoo Jung. 2020. DC-Store: Eliminating noisy neighbor containers using deterministic I/O performance and resource isolation. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*. USENIX Association, 183–191. Retrieved from https://www.usenix.org/conference/fast20/presentation/kwon

[101] Miryeong Kwon, Sangwon Lee, and Myoungsoo Jung. 2023. Cache in hand: Expander-driven CXL prefetcher for next generation CXL-SSD. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'23)*. Association for Computing Machinery, 24–30. DOI: https://doi.org/10.1145/3599691.3603406

[102] Hee-Rock Lee, Chang-Gyu Lee, Seungjin Lee, and Youngjae Kim. 2022. Compaction-aware zone allocation for LSM based key-value store on ZNS SSDs. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'22)*. Association for Computing Machinery, 93–99. DOI: https://doi.org/10.1145/3538643.3539743

[103] Jongsung Lee, Donguk Kim, and Jae W. Lee. 2023. WALTZ: Leveraging zone append to tighten the tail latency of LSM tree on ZNS SSD. *Proc. VLDB Endow.* 16, 11 (07 2023), 2884–2896. DOI: https://doi.org/10.14778/3611479.3611495

[104] Kitae Lee, Dong Hyun Kang, Daeho Jeong, and Young Ik Eom. 2018. Lazy TRIM: Optimizing the journaling overhead caused by TRIM commands on Ext4 file system. In *Proceedings of the IEEE International Conference on Consumer Electronics (ICCE'18)*. 1–3. DOI: https://doi.org/10.1109/ICCE.2018.8326258

[105] Manjong Lee. 2022. Re: [PATCH 2/2] block: remove the per-bio/request write hint. - Manjong Lee. Retrieved from https://lore.kernel.org/all/20220309133119.6915-1-mj0123.lee@samsung.com/

[106] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind. 2016. Application-managed flash. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. USENIX Association, 339–353. Retrieved from https://www.usenix.org/conference/fast16/technical-sessions/presentation/lee

[107] Young-Sik Lee, Sang-Hoon Kim, Jin-Soo Kim, Jaesoo Lee, Chanik Park, and Seungryoul Maeng. 2013. OSSD: A case for object-based solid state drives. In *Proceedings of the IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST'13)*. 1–13. DOI: https://doi.org/10.1109/MSST.2013.6558448

[108] Cangyuan Li, Ying Wang, Cheng Liu, Shengwen Liang, Huawei Li, and Xiaowei Li. 2021. GLIST: Towards in-storage graph learning. In *Proceedings of the USENIX Annual Technical Conference (ATC'21)*. USENIX Association, 225–238. Retrieved from https://www.usenix.org/conference/atc21/presentation/li-cangyuan

[109] Huaicheng Li, Martin L. Putra, Ronald Shi, Xing Lin, Gregory R. Ganger, and Haryadi S. Gunawi. 2021. IODA: A host/device co-design for strong predictability contract on modern flash storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*. Association for Computing Machinery, 263–279. DOI: https://doi.org/10.1145/3477132.3483573

[110] Shaohua Li. 2017. Stream - facebook/rocksdb@eefd75a. Retrieved from https://github.com/facebook/rocksdb/commit/eefd75a228fc2c50174c0a306918c73ded22ace7

[111] Sangwoo Lim and Dongkun Shin. 2019. DStream: Dynamic memory resizing for multi-streamed SSDs. In *Proceedings of the 34th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC'19)*. 1–4. DOI: https://doi.org/10.1109/ITC-CSCC.2019.8793432

[112] Biyong Liu, Yuan Xia, Xueliang Wei, and Wei Tong. 2023. LifetimeKV: Narrowing the lifetime gap of SSTs in LSMT-based KV stores for ZNS SSDs. In *Proceedings of the IEEE 41st International Conference on Computer Design (ICCD'23)*. 300–307. DOI: https://doi.org/10.1109/ICCD58817.2023.00053

[113] Jiahao Liu, Fang Wang, and Dan Feng. 2019. CostPI: Cost-effective performance isolation for shared NVMe SSDs. In *Proceedings of the 48th International Conference on Parallel Processing (ICPP'19)*. Association for Computing Machinery, Article 25, 10 pages. DOI: https://doi.org/10.1145/3337821.3337879

[114] Canonical Ltd. 2019. Ubuntu Manpage: fstrim - discard unused blocks on a mounted filesystem. Retrieved from https://manpages.ubuntu.com/manpages/xenial/en/man8/fstrim.8.html

[115] Mingchen Lu, Peiquan Jin, Xiaoliang Wang, Yongping Luo, and Kuankuan Guo. 2023. ZoneKV: A space-efficient key-value store for ZNS SSDs. In *Proceedings of the 60th ACM/IEEE Design Automation Conference (DAC'23)*. 1–6. DOI: https://doi.org/10.1109/DAC56929.2023.10247926

[116] Umesh Maheshwari. 2020. StripeFinder: Erasure coding of small objects over key-value storage devices (an uphill battle). In *Proceedings of the 12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'20)*. USENIX Association. Retrieved from https://www.usenix.org/conference/hotstorage20/presentation/maheshwari

[117] Umesh Maheshwari. 2021. From blocks to rocks: A natural extension of zoned namespaces. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'21)*. Association for Computing Machinery, 21–27. DOI: https://doi.org/10.1145/3465332.3470870

[118] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. 2020. A study of SSD reliability in large scale enterprise storage deployments. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*. USENIX Association, 137–149. Retrieved from https://www.usenix.org/conference/fast20/presentation/maneas

[119] Nika Mansouri Ghiasi, Jisung Park, Harun Mustafa, Jeremie Kim, Ataberk Olgun, Arvid Gollwitzer, Damla Senol Cali, Can Firtina, Haiyu Mao, Nour Almadhoun Alserr et al. 2022. GenStore: A high-performance in-storage processing system for genome sequence analysis. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*. Association for Computing Machinery, 635–654. DOI: https://doi.org/10.1145/3503222.3507702

[120] Bill Martin, Judy Brock, Dan Helmick, Robert Moss, Mike Allison, Benjamin Lim, Jiwon Chang, Ross Stenfort, Young Ahn, Wei Zhang et al. 2022. TP4146 Flexible Data Placement 2022.11.30 Ratified. Retrieved from https://nvmexpress.org/wp-content/uploads/NVM-Express-2.0-Ratified-TPs_20230111.zip

[121] SATA-IO Board Members. 2011. Serial ATA International Organization: Serial ATA Revision 3.1. Retrieved from https://sata-io.org/system/files/specifications/SerialATA_Revision_3_1_Gold.pdf

[122] Inc. Micron Technology. 2019. What is Trim? | Crucial.com. Retrieved from https://www.crucial.com/articles/about-ssd/what-is-trim

[123] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. 2023. eZNS: An elastic zoned namespace for commodity ZNS SSDs. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*. USENIX Association, 461–477. Retrieved from https://www.usenix.org/conference/osdi23/presentation/min

[124] Samsung MSL. 2023. SmartSSD2.0 – Samsung – Memory Solutions Lab. Retrieved from https://samsungmsl.com/smartssd2/

[125] NVM Express, Inc. 2022. NVM Express RDMA Transport Specification 1.0b. Retrieved from https://nvmexpress.org/wp-content/uploads/NVM-Express-RDMA-Transport-Specification-1.0b-2022.10.04-Ratified.pdf

[126] NVM Express, Inc. 2022. NVM Express TCP Transport Specification 1.0c. Retrieved from https://nvmexpress.org/wp-content/uploads/NVM-Express-TCP-Transport-Specification-1.0c-2022.10.03-Ratified.pdf

[127] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. 2014. SDF: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. Association for Computing Machinery, 471–484. DOI: https://doi.org/10.1145/2541940.2541959

[128] Inhyuk Park, Qing Zheng, Dominic Manno, Soonyeal Yang, Jason Lee, David Bonnie, Bradley Settlemyer, Youngjae Kim, Woosuk Chung, and Gary Grider. 2023. KV-CSD: A hardware-accelerated key-value store for data-intensive applications. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'23)*. 132–144. DOI: https://doi.org/10.1109/CLUSTER52292.2023.00019

[129] Rekha Pitchumani and Yang-Suk Kee. 2020. Hybrid data reliability for emerging key-value storage devices. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*. USENIX Association, 309–322. Retrieved from https://www.usenix.org/conference/fast20/presentation/pitchumani

[130] Luca Porzio. 2022. RE: [EXT] Re: [PATCH 2/2] block: remove the per-bio/request write hint. - Luca Porzio (lporzio). Retrieved from https://lore.kernel.org/all/CO3PR08MB797524ACBF04B861D48AF612DC0B9@CO3PR08MB7975.namprd08.prod.outlook.com/

[131] Devashish Purandare, Pete Wilcox, Heiner Litz, and Shel Finkelstein. 2022. Append is near: Log-based data management on ZNS SSDs. In *Proceedings of the 12th Annual Conference on Innovative Data Systems Research (CIDR'22)*. Retrieved from https://par.nsf.gov/biblio/10315205

[132] Devashish R. Purandare, Sam Schmidt, and Ethan L. Miller. 2023. Persimmon: An append-only ZNS-first filesystem. In *Proceedings of the IEEE 41st International Conference on Computer Design (ICCD'23)*. 308–315. DOI: https://doi.org/10.1109/ICCD58817.2023.00054

[133] Mian Qin, A. L. Narasimha Reddy, Paul V. Gratz, Rekha Pitchumani, and Yang Seok Ki. 2021. KVRAID: High performance, write efficient, update friendly erasure coding scheme for KV-SSDs. In *Proceedings of the 14th ACM International Conference on Systems and Storage (SYSTOR'21)*. Association for Computing Machinery. DOI: https://doi.org/10.1145/3456727.3463781

[134] Eunhee Rho, Kanchan Joshi, Seung-Uk Shin, Nitesh Jagadeesh Shetty, Jooyoung Hwang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. 2018. FStream: Managing flash streams in the file system. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*. USENIX Association, 257–264. Retrieved from https://www.usenix.org/conference/fast18/presentation/rho

[135] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle. 2001. Active disks for large-scale data processing. *Computer* 34, 6 (2001), 68–74. DOI: https://doi.org/10.1109/2.928624

[136] Zhenyuan Ruan, Tong He, and Jason Cong. 2019. INSIDER: Designing in-storage computing system for emerging high-performance drive. In *Proceedings of the USENIX Annual Technical Conference (ATC'19)*. USENIX Association, 379–394. Retrieved from https://www.usenix.org/conference/atc19/presentation/ruan

[137] Chris Sabol, Ross Stenfort, and Mike Allison. 2023. Flexible Data Placement FDP Overview - NVM Express. Retrieved from https://www.youtube.com/watch?v=BENgm5a17ws

[138] Samsung Semiconductors. 2023. UFS 4.0 | Universal Flash Storage | Samsung Semiconductor Global. Retrieved from https://semiconductor.samsung.com/estorage/ufs/ufs-4-0/

[139] Rachel Shaver. 2011. Retrieved from https://sata-io.org/system/files/member-downloads/SATA-IORevision31_PRfinal_0.pdf

[140] Avi Shchislowski. 2022. RE: [EXT] Re: [PATCH 2/2] block: remove the per-bio/request write hint. - Avi Shchislowski. Retrieved from https://lore.kernel.org/all/SN6PR04MB3872231050F8585FFC6824C59A0F9@SN6PR04MB3872.namprd04.prod.outlook.com/

[141] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Re, Ion Stoica, and Ce Zhang. 2023. FlexGen: High-throughput generative inference of large language models with a single GPU. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research,* Vol. 202*)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 31094–31116. Retrieved from https://proceedings.mlr.press/v202/sheng23a.html

[142] Yoni Shternhell and Matias Bjørling. 2022. TP4093a Zone Relative Data Lifetime Hint 2022.08.14 Ratified. Retrieved from https://nvmexpress.org/wp-content/uploads/NVM-Express-2.0-Ratified-TPs_20230111.zip

[143] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2004. Life or death at block-level. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*. 26–26. Retrieved from https://www.usenix.org/legacy/event/osdi04/tech/full_papers/sivathanu/sivathanu.pdf

[144] Eric Slivka. 2011. Mac OS X Lion Roundup: Recovery Partitions, TRIM Support, Core 2 Duo Minimum, Focus on Security - MacRumors. Retrieved from https://www.macrumors.com/2011/02/25/mac-os-x-lion-roundup-recovery-partitions-trim-support-core-2-duo-minimum-focus-on-security/

[145] David Sterba. 2020. Merge tag "for-5.6-tag" of git://git.kernel.org/pub/scm/linux/kernel/... · torvalds/linux@81a046b. Retrieved from https://github.com/torvalds/linux/commit/81a046b18b331ed6192e6fd9ff6d12a1f18058cf

[146] David Sterba. 2022. Re: Using async discard by default with SSDs? - David Sterba. Retrieved from https://lore.kernel.org/linux-btrfs/20220726213628.GO13489@twin.jikos.cz/

[147] David Sterba. 2023. Trim/discard — BTRFS documentation. Retrieved from https://btrfs.readthedocs.io/en/latest/Trim.html

[148] Devesh Tiwari, Simona Boboila, Sudharshan Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. 2013. Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*. USENIX Association, 119–132. Retrieved from https://www.usenix.org/conference/fast13/technical-sessions/presentation/tiwari

[149] Linus Torvalds. 2021. Merge tag "cxl-for-5.12" of git://git.kernel.org/pub/scm/linux/kernel.... Retrieved from https://github.com/torvalds/linux/commit/825d1508750c0cad13e5da564d47a6d59c7612d6

[150] Theodore Ts'o. 2016. Solved - SSD Trim Maintenance | The FreeBSD Forums. Retrieved from https://forums.freebsd.org/threads/ssd-trim-maintenance.56951/#post-328912

[151] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys'14)*. Association for Computing Machinery, Article 16, 14 pages. DOI: https://doi.org/10.1145/2592798.2592804

[152] Qiuping Wang and Patrick P. C. Lee. 2023. ZapRAID: Toward high-performance RAID for ZNS SSDs via zone append. In *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys'23)*. Association for Computing Machinery, 24–29. DOI: https://doi.org/10.1145/3609510.3609810

[153] Wei-Lin Wang, Tseng-Yi Chen, Yuan-Hao Chang, Hsin-Wen Wei, and Wei-Kuan Shih. 2020. How to cut out expired data with nearly zero overhead for solid-state drives. In *Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC'20)*. 1–6. DOI: https://doi.org/10.1109/DAC18072.2020.9218610

[154] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. 2021. RecSSD: Near data processing for solid state drive based recommendation inference. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*. Association for Computing Machinery, 717–729. DOI: https://doi.org/10.1145/3445814.3446763

[155] SerialATA Workgroup. 2001. Serial ATA: High Speed Serialized AT Attachment Revision 1.0. Retrieved from https://www.seagate.com/support/disc/manuals/sata/sata_im.pdf

[156] Zheng Xu and Yanxiang Zhang. 2024. Advances in private training for production on-device language models. Retrieved from https://research.google/blog/advances-in-private-training-for-production-on-device-language-models/

[157] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. 2017. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs. *ACM Trans. Stor.* 13, 3, Article 22 (10 2017), 26 pages. DOI: https://doi.org/10.1145/3121133

[158] Jingpei Yang, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. 2017. AutoStream: Automatic stream management for multi-streamed SSDs. In *Proceedings of the 10th ACM International Systems and Storage Conference (SYSTOR'17)*. Association for Computing Machinery, Article 3, 11 pages. DOI: https://doi.org/10.1145/3078468.3078469

[159] Jing Yang, Shuyi Pei, and Qing Yang. 2019. WARCIP: Write amplification reduction by clustering I/O pages. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR'19)*. Association for Computing Machinery, 155–166. DOI: https://doi.org/10.1145/3319647.3325840

[160] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. 2014. Don't stack your log on my log. In *Proceedings of the 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW'14)*. USENIX Association. Retrieved from https://www.usenix.org/conference/inflow14/workshop-program/presentation/yang

[161] Pan Yang, Ni Xue, Yuqi Zhang, Yangxu Zhou, Li Sun, Wenwen Chen, Zhonggang Chen, Wei Xia, Junke Li, and Kihyoun Kwon. 2019. Reducing garbage collection overhead in SSD based on workload prediction. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'19)*. USENIX Association. Retrieved from https://www.usenix.org/conference/hotstorage19/presentation/yang

[162] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jin yong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S. Kim. 2023. Overcoming the memory wall with CXL-enabled SSDs. In *Proceedings of the USENIX Annual Technical Conference (ATC'23)*. USENIX Association, 601–617. Retrieved from https://www.usenix.org/conference/atc23/presentation/yang-shao-peng

[163] Zhe Yang, Youyou Lu, Xiaojian Liao, Youmin Chen, Junru Li, Siyu He, and Jiwu Shu. 2023. $\lambda$-IO: A unified IO stack for computational storage. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST'23)*. USENIX Association, 347–362. Retrieved from https://www.usenix.org/conference/fast23/presentation/yang-zhe

[164] Hwanjin Yong, Kisik Jeong, Joonwon Lee, and Jin-Soo Kim. 2018. vStream: Virtual stream management for multi-streamed SSDs. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'18)*. USENIX Association. Retrieved from https://www.usenix.org/conference/hotstorage18/presentation/yong

[165] Jiacheng Zhang, Youyou Lu, Jiwu Shu, and Xiongjun Qin. 2017. FlashKV: Accelerating KV performance with open-channel SSDs. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 139 (09 2017), 19 pages. DOI: https://doi.org/10.1145/3126545

[166] Jian Zhang, Yujie Ren, and Sudarsun Kannan. 2022. FusionFS: Fusing I/O operations using CISCOps in firmware file systems. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST'22)*. USENIX Association, 297–312. Retrieved from https://www.usenix.org/conference/fast22/presentation/zhang-jian

[167] Xiangqun Zhang, Shuyi Pei, Jongmoo Choi, and Bryan S. Kim. 2023. Excessive SSD-internal parallelism considered harmful. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'23)*. Association for Computing Machinery, 65–72. DOI: https://doi.org/10.1145/3599691.3603412

[168] Yuqi Zhang, Ni Xue, and Yangxu Zhou. 2021. Automatic I/O stream management based on file characteristics. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'21)*. Association for Computing Machinery, 14–20. DOI: https://doi.org/10.1145/3465332.3470879

[169] Yi Zheng, Joshua Fixelle, Nagadastagiri Challapalle, Pingyi Huo, Zhaoyan Shen, Zili Shao, Mircea Stan, and Vijaykrishnan Narayanan. 2022. ISKEVA: in-SSD key-value database engine for video analytics applications. In *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'22)*. Association for Computing Machinery, 50–60. DOI: https://doi.org/10.1145/3519941.3535068

[170] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman et al. 2022. XRP: In-kernel storage functions with eBPF. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*. USENIX Association, 375–393. Retrieved from https://www.usenix.org/conference/osdi22/presentation/zhong

[171] Yuhong Zhong, Hongyi Wang, Yu Jian Wu, Asaf Cidon, Ryan Stutsman, Amy Tai, and Junfeng Yang. 2021. BPF for storage: An exokernel-inspired approach. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS'21)*. Association for Computing Machinery, 128–135. DOI: https://doi.org/10.1145/3458336.3465290

## Appendix

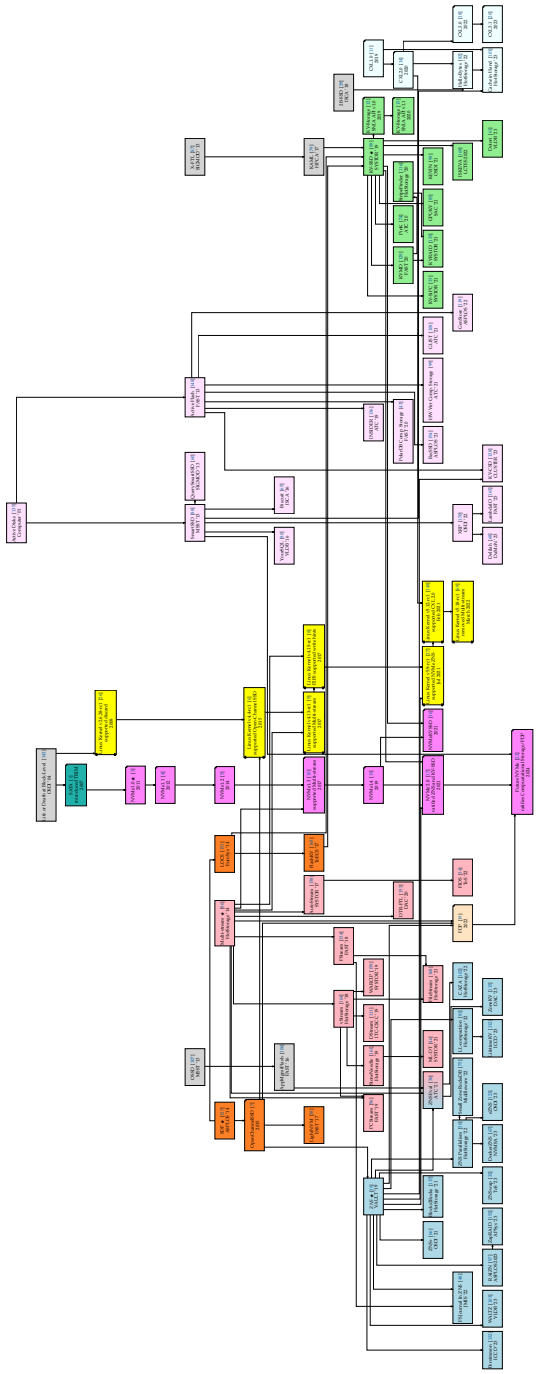

Fig. 12. Complete genealogy tree of the surveyed papers (Figure 1).

Received 13 November 2023; revised 1 August 2024; accepted 4 October 2024