A Survey on the Integration of NAND Flash Storage in the Design of File Systems and the Host Storage Software Stack

Tehrany, Nick; Doekemeijer, Krijn; Trivedi, Animesh

2023

A Survey on the Integration of NAND Flash Storage in the Design of File Systems and the Host Storage Software Stack

Survey done: July 2022

Nick Tehrany
Delft University of Technology
n.a.tehrany@vu.nl

Krijn Doekemeijer
Vrije Universiteit Amsterdam
k.doekemeijer@vu.nl

Animesh Trivedi
Vrije Universiteit Amsterdam
a.trivedi@vu.nl

Abstract

With the ever-increasing amount of data generate in the world, estimated to reach over 200 Zettabytes by 2025, pressure on efficient data storage systems is intensifying. The shift from HDD to flash-based SSD provides one of the most fundamental shifts in storage technology, increasing performance capabilities significantly. However, flash storage comes with different characteristics than prior HDD storage technology. Therefore, storage software was unsuitable for leveraging the capabilities of flash storage. As a result, a plethora of storage applications have been design to better integrate with flash storage and align with flash characteristics.

In this literature study we evaluate the effect the introduction of flash storage has had on the design of file systems, which providing one of the most essential mechanisms for managing persistent storage. We analyze the mechanisms for effectively managing flash storage, managing overheads of introduced design requirements, and leverage the capabilities of flash storage. Numerous methods have been adopted in file systems, however prominently revolve around similar design decisions, adhering to the flash hardware constrains, and limiting software intervention. Future design of storage software remains prominent with the constant growth in flash-based storage devices and interfaces, providing an increasing possibility to enhance flash integration in the host storage software stack.

1 Introduction

With the increasing amount of data, estimated to reach 200 Zettabytes by the year 2025 [182], efficient storage systems are becoming imperative. A large contribution factor to increased data generation is the gain in popularity for big data [9, 80, 124] and cloud services [6, 215]. While there exist a plethora of different storage technologies, the most prevalent type is *Hard Disk Drive (HDD)* [7, 48], which are now largely being replaced by *Solid State Drive (SSD)* [45]. HDD is one of the cheapest forms of storage, however is limited in performance due to requiring on mechanical movement to access data on the disk. This results in high latency for random access patterns [53, 102] and additionally increases power demand [29, 73]. While SSD is more expensive than HDD, it is becoming more affordable [180] and provides increased performance over HDD [119], resulting in a growing adoption for enterprise businesses [46, 146].

One of the most fundamental mechanisms of storing and organizing data on HDD, SSD, and other storage technologies, is through the use of file systems, enabling the structural organization of data on persistent storage media. Building efficient and performant file systems for the evolving storage media technologies and progressing with future demands of data storage is of paramount importance. With HDD having been the prevailing storage technology for decades, file system and application design revolved around the intrinsic characteristics of these devices. In particular, aiming to limit access patterns to sequential accesses [25, 190], in an effort to minimize mechanical movement on the disk and thus optimize their performance.

The most widely adopted type of SSD is based on *flash storage*, having different characteristics than traditional HDD. Performance of flash storage achieves several GB/s, with millions of *I/O Operations per Second (IOPS)* [91, 220], and access latency as low as single digit μ-second latency. However, flash storage has its own characteristics different from HDD. In particular, flash storage does not support in-place updates, requiring data to be erased at a larger unit in order to be written again. Additionally, the cost of erase operations is substantially higher than read and write operations [91, 235]. In order to hide these constraints from host systems, flash SSD employs firmware, called the *Flash Translation Layer (FTL)*, that exposes a sector-addressable interface. This allows SSD to be addressed in the same way as conventional HDD, requiring no changes in host software for accessing the different storage technologies.

While SSD and HDD utilize the same interfaces to be addressed, in order to exploit the increased performance benefits of flash storage, software must integrate with the characteris-

tics of flash storage. Adapting software design to align with flash storage characteristics helps minimize FTL overheads to manage the flash storage. With the increasing adoption of flash SSD in enterprise, a plethora of applications and file systems have been proposed aiming at integrating software design with flash storage characteristics. In this survey we evaluate the changes in software, particularly in file systems, caused by integrating with flash storage characteristics, and how these changes have affected file system design. We additionally assess the future implications of evolving flash storage technologies to file system and software design. In order to evaluate the various work on flash storage implications for file system design, we devise three key *Survey Research Question (SRQ)* that aim at analyzing past, current, and future trends.

SRQ1. What are the main challenges arising from NAND flash characteristics and its integration into file system design?
Flash storage has particular characteristics, such as sequential writing, no in-place updates, and requiring erasing of flash blocks. This SRQ aims at analyzing what particular challenges arise for storage software from the flash-specific constraints and resulting effects of on-device operations. Devising a list of key challenges provides the foundation based on which relevant work in this literature study is selected, and the final report is structured.

SRQ2. How has NAND flash storage influenced the design and development of file system and the storage software stack?
Using the identified challenges in **SRQ1**, this SRQ evaluates for each of the challenges, how file system design has changed to integrate with it. As file systems are commonly built on top of existing storage software layers, such as the Linux Block I/O layer, we include methods and mechanisms in the storage software stack particularly devised for file systems and flash storage integration. As a result, this SRQ evaluates how the depicted challenges are addressed throughout the various software stack layers, up to the file system.

SRQ3. How will NAND flash storage and newly introduced NAND flash-based storage devices and interfaces affect future file system design and development?
With a particular goal of this literature study being to evaluate the validity of data structures, algorithms, and mechanisms of flash, and understanding the applicability to ZNS, a newly arising storage technology, this research question furthermore aims at evaluating future challenges that may arise from new technology.

Furthermore, this literature study makes the following contributions:

- We devise six key challenges for storage software arising from the integration of flash-based SSD, particularly focusing on leveraging its capabilities and enhancing device utilization.
- For each of the six devised flash integration challenges, we summarize the main methods of relevant work on dealing with and integration the particular challenge(s) into file system design.
- Based on the findings of this literature study, we present a discussion on the future applicability of the presented methods during this study, and evaluate the effects of newly arising flash-based SSD devices.

2 Literate Study Research Methodology

Several methodologies for conducting literature studies exist such as an unguided traversal of the literature [82] and using *snowballing* [253, 256]. However, we find that said methods lack systematic mechanisms in their evaluation, where the search space of relevant literature with unguided is not clearly defined and can become incomprehensibly large. Snowballing on the other hand allows limiting the search space by evaluation going forward, studies that reference the seed paper, and backwards, studies that the seed paper references, in the citations from a set of seed papers systematically. However, it can introduce possible bias from the seed paper selection. Therefore, we utilize a combination of approaches and additionally apply the *Systematic Literature Review (SLR)* presented by Kitchenham et al. [134]. The SLR approach relies on three separate stages to construct a systematic literature review, depicted in Table 1. As we do not inclusively apply every possible stage of the SLR method, the table additionally depicts which methods we apply in this study (indicated with a ✓), and where relevant information can be found in this literature study.

With these three stages, clear protocol and process definitions are established prior to conducting the review (discussed in Section 2.1), limiting possible reviewer bias and additionally enhancing reproducibility. The initial stage consists of planning the review, which includes establishing the need for this certain review and developing the review protocol. The protocol encompasses the inclusion and exclusion criteria, as well as research question definition, and the establishment of the review process. Based on an established protocol we conduct the review, establish the selection of studies to evaluate, and proceed to extract and analyze the studies. Lastly, with all collected data on selected studies, we format this survey to present the review in a comprehensible manner.

	Planning the review	
✓	Identification of the need for a review	(§1)
✗	Commissioning a review	-
✓	Specifying the research question(s)	(§1)
✓	Developing a review protocol	(§2.1)
✗	Evaluating the review protocol	-
	Conducting the review	
✓	Identification of research	(§2.2)
✓	Selection of primary studies	(§2.3)
✗	Study quality assessment	-
✓	Data extraction and monitoring	(§2.4)
✓	Data synthesis	(§2.4)
	Reporting the review	
✗	Specifying dissemination mechanism	-
✓	Formatting the main report	-
✗	Evaluating the report	-

Table 1: Outline of the Systematic Literature Review approach presented by Kitchenham et al. [134], inspired by the structure of the literature study on Graph Analysis by Hegeman and Iosup [82].

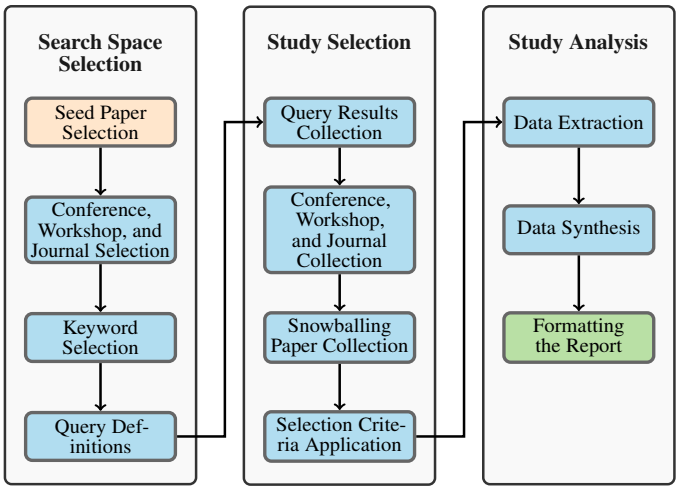

Figure 1: Review Protocol phases applied in this literature study.

2.1 Review Protocol

By making use of the SLR process, we initially define a review protocol and review processes that we detail for this study. With emphasis on making this study reproducible, we provide detailed descriptions of each phase in the review process. A visual representation of the application of the review protocol phases is depicted in Figure 1. It revolves around three phases, firstly establishing the search space from which we extract relevant studies for this literature study. In the second phase we collect the studies and apply the defined selection criteria (explained in Section 2.3). Lastly, we analyze the collected relevant studies. The following sections explain each of the phases in detail, Section 2.2 explains methods used for establishing the search space of literature. Next, Section 2.3 depicts the selection criteria for extracting relevant studies from the defined search space. Lastly, Section 2.4 provides the methods of data extraction from studies and gives the organization of studies for this literature review.

2.2 Search Space Selection

The first stage of our review protocol defines the definition of the search space from which relevant studies are extracted. We make use of several approaches for identifying relevant studies. Firstly, we apply the snowballing method on a set of seed papers. Seed papers selected for this literature review are depicted in Table 2. With these seed papers, we analyze the studies from forward and backward citations of the seed paper. To further expand the search space of relevant studies, we examine the publications of numerous conferences, workshops, and journals which are focused on the area of systems and storage research. These conferences are analyzed in the range of 2010-2022, if present, as some are bi-annual or may not have been established in the given time range. The following venues are checked in this literature review:

- USENIX Annual Technical Conference (*USENIX ATC*)
- USENIX Conference on File and Storage Technologies (*FAST*)
- Networked Systems Design & Implementation (*NSDI*)
- European Conference on Computer Systems (*EuroSys*)
- USENIX Symposium on Operating Systems Design and Implementation (*OSDI*)
- Symposium on Operating Systems Principles (*SOSP*)
- ACM International Systems and Storage Conference (*SYSTOR*)
- ACM Workshop on Hot Topics in Storage and File Systems (*HotStorage*)
- Architectural Support for Programming Languages and Operating Systems (*ASPLOS*)
- ACM Special Interest Group on Management of Data (*SIGMOD*)
- International Conference on Very Large Data Bases (*VLDB*)
- IEEE International System-on-Chip Conference (*SOCC*)
- International Conference on Distributed Computing Systems (*ICDCS*)

Title	Venue	Publication Year
F2FS [143]	FAST	2016
JFFS [257]	OLS	2001
LogFS [59]	Linux Kongress	2005

Table 2: Seed papers used for this literature review. Titles are shortened.

- ACM/IFIP Middleware Conference (*Middleware*)
- ACM Transactions on Storage (*TOS*)
- International Conference for High Performance Computing, Networking, Storage, and Analysis (*SC*)
- International Conference on Massive Storage Systems and Technology (*MSST*)
- IEEE International Conference on Computer Design (*ICCD*)

Lastly, we run individual queries on academic search engines for scholarly literature; Google Scholar, Semantic Scholar, and dblp. We utilize two types of queries; *Relevant Studies Query (RSQ)* for finding of relevant studies for this literature study, and *Related Literature Studies Query (RLSQ)* for finding related work to this literature review. Related work encompasses surveys on file systems for flash, flash specific algorithms and data structures, and additional studies of flash related application and system integration. For each query, with each search engine, we analyze the 100 most relevant results (or less if there are fewer query results). The keyword queries for finding relevant studies and relevant related literature studies are, respectively:

RSQ1. Flash File System

RSQ2. NVM File System

RSQ3. SSD File System

RSQ4. File System SPDK [1]

RLSQ1. Flash File System Survey

RLSQ2. NVM File System Survey

RLSQ3. SSD File System Survey

Finding of related surveys is not limited to the established queries, but additionally during the snowballing and synthesis of conference, workshop, and journal publications we identify related work based on the prior defined classifications. Inclusion of relevant studies outside of the time range from 2010-2022 is only applicable when the study is retrieved using snowballing from seed papers, or the study is present in one of the respective query results. The timing constraint is thus only applied to the extraction of relevant studies by analyzing publications at conferences, workshops, and journals.

[1] *Storage Performance Development Kit (SPDK)* [271] provides a number of tools and libraries for building high performant user-level storage software over NVMe, making it applicable to file system development on flash SSDs.

2.3 Study Selection Criteria

The second phase of the review protocol defines the study selection, which extracts the relevant studies from the defined search space with a set of established criteria. We define a specific Inclusion/Exclusion criteria, with which studies are selected for this literature review based on the appropriateness for the criteria. These criteria are based on the defined research questions and are aimed to narrow down the search space to a particular set of studies of interest in this review, and enforce only relevant work is included. While studies do not have to exclusively meet all the inclusion requirements to be included in this review, any if any of the exclusion criteria is present, the study is not included in this review.

I1. The work is novel.

I2. The work designs a file system specifically for NAND flash storage.

I3. The work adapts an existing file system to integrate with NAND flash storage.

E1. The work designs or adapts a file system not specifically for NAND flash storage.

E2. The work designs or adapts a hybrid/tiered file system that utilizes various storage technologies, not focusing file system design to NAND flash storage.

E3. The work designs or adapts a file system which is evaluated on NAND flash storage, but not particularly built for NAND flash storage.

E4. The work designs or adapts a file system for SSD, but not specifically for NAND flash storage.

While meeting inclusion requirements does not mean a study is guaranteed to be included, it is more likely to be included. In the case of exclusion criteria, exceptions are made in cases of using papers to establish background knowledge or building context, however they are not the core focus of the respective section where its content is discussed.

Furthermore, there has been a plethora of flash-based file systems which utilize hybrid/tiered storage devices, including NOR-based flash [144], *Storage Class Memory (SCM)* [109,156,174,203,210], byte addressable Non-Volatile Memory (NVM) [142], and methods that expose byte addressable NVRAM on SSDs with custom firmware for metadata placement [104,279]. Our focus is on block address storage,

which is the most prevalent for storage. Additionally, combining of multiple storage technologies, such as SSD and HDD, commonly utilize SSD as a cache for the file system on the HDD, similarly shifting focus away from flash storage integration for the file system. For this reason we exclude such file system designs from the core study of this literature review. Similarly, file systems that are not designed specifically for flash storage, but have flash-friendly characteristics are also excluded. This mainly includes log-structured file systems that are intended for HDD, with a focus on writing sequentially to minimize arm movement on the disk, which coincidentally matches the sequential write requirement of flash.

2.4 Study Analysis

The last stage of the review protocol defines the extraction of data from the relevant studies, and establishing the final report. During evaluation of the different studies selected for this literature review, we disseminate the information presented based on their answering of the defined research questions. For this, we define the various key integration challenges of flash storage integration (Section 5), based on the hardware and software characteristics of flash storage. Using the defined integration challenges, we divide the contributions of the studies evaluated in this literature study into the respective challenge, and discuss its mechanisms for solving the particular flash integration challenge. These challenges are evaluated in Sections 6 to 12. Lastly, we discuss on the findings of the main findings from this literature study, followed by the relevant related literature studies for flash storage in Section 14.

2.5 Limitations

Albeit this literature study being extensively defined and established through clear protocol definitions, there are several limitations that remain. Firstly, the search space selection is only based on studies that have scientific literature. Therefore, file systems for flash which may be in the mainline Linux kernel, are not guaranteed to be included in this survey, if no scientific literature on it exists. Secondly, the search space is limited to only scientific literature written in English. This additionally limits the inclusion of conferences, workshops, and journals to venues with proceedings in English. Thirdly, given that snowballing uses a manual selection of meeting inclusion/exclusion criteria, possible bias is inherent. Lastly, as the resulting queries on the selected literature search engines produces several hundred thousand results, and we select to evaluate the first 100 results, we rely on the sorting of results based on relevance that each search engine provides. While we utilize multiple search engines in an effort to avoid bias from each search engine, it does not completely eliminate it.

3 Background

The increasing adoption of flash-based SSD [46, 146], due to its higher performance capabilities compared to conventional HDD, and decreasing cost is making it a ubiquitous storage media for storage systems. In this section we explain the high-level concepts of the construction of flash-based SSD (Sections 3.1 to 3.4), technical details of the newly standardizes ZNS SSD (Section 3.5), and how the de facto standard file system F2FS is designed for flash storage (Section 3.6).

3.1 Flash Storage Building Block: Flash Cells

Flash storage is based on *flash cells* providing the persistent storage capabilities through programming of the cell and erasing it to clear its content. Each flash cell is constructed with a floating gate, which can hold an electrical charge, and a control gate, inside a *Complementary Metal-Oxide Semiconductor (CMOS)* transistor [191]. Using the electrical charge present inside the flash cell, each cell is capable of representing a logical value (0 or 1). Such flash cells, capable of representing a single bit, are called *Single-Level Cell (SLC)*. Data is written to flash cells by *programming* the cell. Updating existing data in flash cells requires the cell to be *erased*, followed by being programmed. In addition to SLC, other types of flash cells are available, ranging from *Multi-Level Cell (MLC)* (2 bits per flash cell) to *Penta-Level Cell (PLC)* (5 bits per cell) [170]. To represent more bits with a single flash cell, the charge inside the flash cell is divided into a respective number of ranges in order to represent the number of bit combinations (e.g., 4 ranges to represent all combinations of 2 bits in MLC).

Increasing the number of levels in flash cells however results in slower access time [7], increased wear on the flash cells and a lower lifetime [181, 223], as the cells erode over time with it being programmed. The wear on flash cells is accelerated with more bits being stored in the cell (i.e., increasing the cell level). For SLC the number of program/erase cycles is typically 100K, and decreases to 10K or less for MLC [223]. As a result enterprise flash SSD most commonly employ SLC for increased reliability and lifetime.

Similarly, the organization of flash cells dictates the resulting type of storage device. Flash cells can be organized in the form of *NOR* or *NAND*, representing the connection architecture of the cells. With NOR flash, the resulting flash is byte-addressable, whereas NAND flash provides a page-addressable structure, with a page representing the unit of read and write. As a result NOR flash is commonly applied in *Basic I/O System (BIOS)* [139], however NAND architecture is suitable for mass data storage, making it the primary architecture for flash SSD. Throughout this literature study we focus purely on NAND flash, however for more information on flash cell architecture consult reports such as [12] and [43].

3.2 Building Mass Storage SSD with Flash Cells

Utilizing the building block of flash cells, building mass storage devices is achieved by grouping together several flash cells into *pages* (typically 8-16KiB in size), depicting the unit of access (read/write). A flash page additional has a small extra space, called the *Out-Of-Band (OOB)* area, which is utilized for *Error Correction Codes (ECC)* and small metadata. As the operation for erasing of a flash page is costly [72], multiple pages are grouped into a *block* (several MiB in size), which is the unit of erase. Increasing the erase unit from individual pages to a block helps amortize the erase overheads. Pages within a block have to be written sequentially and prior to a page being overwritten, the entire block must be erased. A number of blocks are then grouped into a *plane*, of which multiple planes are combined into a *die* [2]. Lastly, numerous chips are combined into a single flash package. Figure 2 shows a visual representation of a flash-based SSD architecture.

In order to build full storage devices, multiple flash packages are packed together into a SSD. While a SSD can be constructed with any solid state technology such as Optane [260, 266], we focus purely on NAND flash based SSD. The SSD contains a controller, which is responsible for processing requests, managing data buffers, and contains the NAND flash controller that manages the flash packages. An additional *Random Access Memory (RAM)* buffer, most commonly in the form of *Dynamic RAM (DRAM)*, is present on the device for maintaining address mappings. Lastly, a SSD contains the host interface that provides the means to connect the storage device to the host system over connection interfaces such as *Serial Advanced Technology Attachment (SATA)* and *PCI Express (PCIe)*, and defining standards such as *Advanced Host Controller Interface (AHCI)* and *Non-Volatile Memory Express (NVMe)* [141]. NVMe is the interface specification particularly designed for fast SSD devices, capable of outperforming legacy protocols such as SATA with 8x higher performance [263], due to its increased number of I/O queues to which requests can be issued in parallel. Similarly, PCIe protocol achieves the highest throughput and lowest latency compared to SATA [60]. As a result, flash SSD are commonly connected to systems through NVMe over PCI.

3.3 Increasing Flash SSD Performance

Given the architecture of flash SSD, the it contains a large degree of possible parallelism (i.e., multiple channels, flash packages, dies, and planes). Furthermore, based on a study on a Samsung SSD, the bandwidth of NAND flash *Input/Output (I/O)* buses, connecting the flash to the flash controller, is limited to 32MB/s because of physical restrictions, and 40MB/s with interleaving in dies [2]. A write operation to flash storage firstly writes the data to a data register, from which the data in the register is then programmed into the flash page. Because the programming operation takes longer than loading of data, these operations can be interleaved, such that while a page is being programmed, data of another write operation is loaded [131]. This avoids stalling whenever a page programming finishes and data needs to be loaded again, which is fundamentally similar to the concept of CPU pipelining [240].

In order to increase performance, parallelism is utilized on the different flash entities in the SSD [31]. Depending on the hardware configurations, the different types of parallelism are referred to as *channel-level parallelism*, where requests are distributed individually across the flash channels, *package-level parallelism* where flash packages on the same channel are accessed in parallel, *chip-level parallelism* where flash packages with multiple chips access these in parallel, and *plane-level parallelism* where the same operation can be run on multiple planes, on the same chip, in parallel. A commonly applied technique for enhanced parallelism is the utilization of *clustered blocks*, where requests are issued to blocks in parallel to different chips and planes.

There are several additional performance optimizations for accessing flash storage. Other than accessing a flash entity in parallel and operation interleaving, flash packages can be accessed synchronously with a single request queue through *ganging* [2]. Flash packages within a gang share the same control bus from the flash controller, however can utilize different data buses, thus allowing a single command request queue with multiple data buses to provide the resulting data. Therefore, requests that require data from multiple pages can be split among a gang from a single request queue, and provide the data on different buses, avoiding the bottleneck of a single data bus.

3.4 Hiding Flash Management Idiosyncrasies on the SSD

With the flash characteristic requiring sequential writing within blocks, and lacking support for in-place updates, flash SSD employs a *Flash Translation Layer (FTL)* to hide these management idiosyncrasies, and provide seemingly in-place updates. As a result of the sequential write constraint, if data inside a flash page is updated, the page is simply marked as invalid and the new data is appended to a new a flash page, possibly in the same block. The FTL is responsible for managing flash page information on their validity and its mappings to *Logical Block Address (LBA)*, which is the mechanism for storage software to address the storage device.

Different implementations of a FTL can utilize different mapping levels, such as block- and page-level mappings. A naive FTL design is to maintain a fully associative mapping of each *Logical Block Address (LBA)* to every possible *Physical Block Address (PBA)* [72], referred to as *Logical-to-Physical (L2P)* mapping, and inversely *Physical-to-Logical*

[2] *Dies* are also referred to as *chips*, we use the terms interchangeably and mean the same entity.

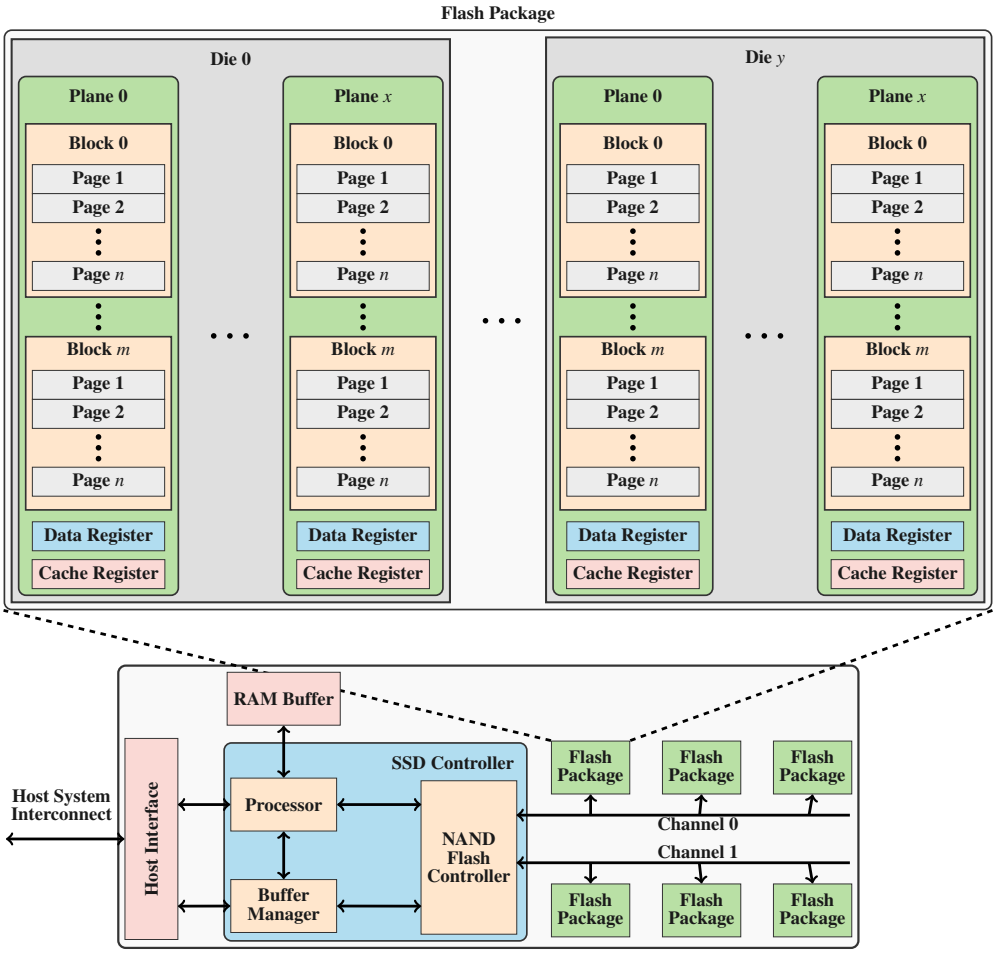

Figure 2: Internal architecture of an SSD, with n pages in a block, m blocks in a plane, x planes in a die, and y dies in a flash package. The example SSD has two channels however any architecture with various numbers of channels is possible. Adapted from [2, 69].

(P2L) mapping. These mappings are maintained in a *mapping table*, which is kept in the RAM of the flash device for faster accesses. For consistency the mapping table is also maintained in the persistent flash storage, where on startup time of the devices the mapping table is reconstructed in the device RAM [2, 30]. With this, the SSD can provide seemingly in-place updates by simply writing data to a free page and invalidating the overwritten data. Further details on FTL mappings are not required for the remainder of this literature study, however for a detailed explanation of FTL mapping algorithms consult [37] and [72].

As over time an increasing number of flash pages become invalid due to data updates, blocks contain valid and invalid data, requiring the FTL to run *Garbage Collection (GC)*. During GC, the FTL selects a block from which it reads the still valid flash pages, writes these to an empty block, followed by erasing of the original block. The process of GC requires the device to provide an empty space that cannot be used as storage, but is only used for the FTL to move valid pages, referred to as the *Over-Provisioning Space (OPS)*. If a device has fully utilized all its capacity, the FTL must be able to write out valid pages, which the overprovisioning space serves at. Therefore, SSD commonly ship with an overprovisioning space of 10-28%, which is only usable by the FTL. In addition to GC, the FTL is responsible to ensure even wear across flash cells, as a flash cell has a limited number of program/erase cycles. This process is referred to as *Wear Leveling (WL)*, where the FTL ensures the flash wears out evenly across the entire device, and no particular parts are burnt out faster than others.

3.5 Zoned Namespace SSD

While the FTL provides the seamless integration of flash SSD into storage systems, details about the flash characteristics, such as the garbage collection unit, are commonly hidden from users, making reverse engineering of such informa-

tion increasingly complex [54, 154]. Furthermore, its unpredictable performance as a result of GC [130, 267] has shifted the research community to new flash interfaces that expose flash management to the host system. Such interfaces allow for better coordination between the storage device and the host software by decreasing FTL responsibility, and increasing control for host software of the flash storage. Efforts for open flash SSD interfaces include *Software-Defined Flash (SDF)* [188] and *Open-Channel SSD (OCSSD)* [16, 205], which however failed to gain large scale adoption due to the lack of standardization, resulting in device-specific implementations, and complex interfaces, requiring software developers to have extensive knowledge of flash in order to be able to build flash-specific software.

The newest addition to opening flash SSD interfaces comes with arrival of *Zoned Namespace (ZNS)* SSD. The 2.0 base specification of NVMe [258] (published in June 2021), establishes the standardization of ZNS with the concept of splitting the address space of the storage device into a number of *zones*, which are independently addressable. This concept of representing the storage space with zones has previously been introduced with the addition of SMR HDDs [61, 68, 237]. These are a particular type of HDD that increase the storage density. Its concept of zones was included in the Linux kernel through the *Zoned Block Device ATA Command Set (ZAC)* and *Zoned Block Command (ZBC)* specifications [40, 41]. Similar to SMR, with ZNS zones have to be written sequentially, and must be reset prior to being overwritten, matching the internal characteristics of flash.

ZNS Interface. Figure 3 depicts a simplified layout of zones on a ZNS device, illustrating the management of the sequential write requirement within zones with a *Write Pointer (WP)* for each zone. The WP indicates the next *Logical Block Address (LBA)* to be written in the particular zone. The starting LBA of a zone is represented by a *Zone Start LBA (ZSLBA)*, identifying the first LBA of each zone. For zone management, each zone has an associated state, such as *FULL* and *EMPTY*.

In addition to the zone management, the NVMe standardization of ZNS introduces three new concepts. Firstly, the specification details the zone capacity, which limits the addressable region within a zone. In order to integrate into the Linux kernel, the zone size must be a power of two value, since kernel operations rely on bit shifts for addressing. However, the addressable space in a zone may not be a power of two value, and can therefore be less than or equal to the zone size. Any LBA beyond the zone capacity is not addressable.

Secondly, the specification adds a limit on the number of concurrently active zones. Zone states indicate the state of the zone, where zones that are currently in an *OPEN* or *CLOSED* state are considered to be active. As the device must allocate resources for active zones, such as write buffers, it enforces a limit on concurrently active resources. Lastly, ZNS introduces a new *zone-append* command, which instead of requiring the host to manage I/Os, such that they adhere to

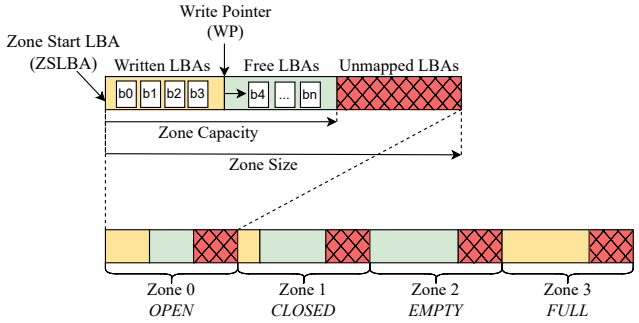

Figure 3: Layout of a ZNS SSD, depicting zone capacity, write pointer, and zone states associated to each zone. Adapted from [243].

the sequential write constraint within a zone, allows the host to issue write I/Os to a zone without specifying the LBA. The device handles the write and returns the address at which the data is written.

The *zone append* command is particularly beneficial with large queue depths (submitting numerous asynchronous write I/O requests), which is not possible with write commands, as these must be issued at consecutive addresses and writes can be reordered in the block layer of the Linux kernel or inside the storage device. In order to adhere to the write constraint in a zone without relying on the zone append command, the `mq-deadline` scheduler within the Linux kernel must be enabled. The `mq-deadline` scheduler holds back I/Os and only submits a single I/O at a time to the ZNS device. Furthermore, this allows to merge I/O requests in the scheduler, enhancing performance by issuing a smaller number of larger I/O requests. Evaluations on the performance of the different schedulers show the benefits of merging I/O requests with larger I/Os of $\geq$ 16KiB being required to saturate the device bandwidth [229, 243].

3.6 F2FS: Flash-Friendly File System

A ubiquitous approach of managing persistent storage devices is with file systems, providing the familiar file and directory interface for organizing storage. The lack of in-place updates on flash pages, enforcing sequential writes, makes *Log-Structured File System (LFS)* [135, 219, 224] a suitable file system design. LFS revolves around writing data as a log, appending new data sequentially on the storage device. In this section, we describe the de facto standard LFS for flash-based storage devices, *Flash-Friendly File System (F2FS)* [143], a plethora of file systems base their design on the foundations presented by F2FS.

3.6.1 F2FS Data Layout

Internally, F2FS utilizes a data allocation unit referred to as a *block*, of 4KiB, in which blocks are allocated on the log.

Consecutive blocks are collected into a 2MiB *segment*, of which one or multiple segments are further grouped into a *section*, that are combined into a *zone*. Figure 4 shows the layout of segments, sections, and zones for the data logs (on the right half of the figure). The left half of figure Figure 4 shows the metadata structures in F2FS to manage the file system, consisting of a *Checkpoint (CP)*, *Segment Information Table (SIT)*, *Node Address Table (NAT)*, and *Segment Summary Area (SSA)*. We now explain each of these data structures.

Checkpoint. F2FS utilizes *checkpointing*, in which all essential metadata for the file system is stored to provide recovery in the case of system failure. A checkpoint is periodically generated, or explicitly triggered, and persists all metadata information from memory. In the case of a system crash or power loss, the file system can recover the state from the latest checkpoint, referred to as *roll-back recovery*, since the latest changes which are not in the checkpoint are reverted. In order to recover the latest changes, the host must call *fsync()* to ensure that metadata and data are flushed from memory to the device. This recovery is referred *roll-forward recovery* since it recovers the state past the latest checkpoint. F2FS can only guarantee roll-forward recovery with *fsync()*.

Segment Information Table. With the data allocation in F2FS being organized in 2MiB segments on the log, the SIT maintains information on each of the segments. It maintains bitmaps for each segment to indicate valid and invalid blocks (blocks that have been overwritten).

Node Address Table. Similar to other file systems, a file in F2FS is managed through an *index node (inode)*, and contains all the file information, including the file specific metadata on creation time, access permissions, file name, and more. Figure 5 shows the inode of F2FS. For identifying the data blocks of the associated file, inodes contain a fixed number of *pointers* to the addresses of the file data. Since an inode is allocated in a block (4KiB), they can often contain inline data of the file, if the file data fits in the available space of the inode. If data for a file is updated, a new block is allocated on the log for the new data, followed by an update to the inode, in order to modify the pointer to the data to point to the newly written data. However, this allocates a new block for the inode of the file, requiring the metadata to track the inode locations to similarly be updated. These changes continue propagating to the parent nodes, resulting in a high increase of required metadata updates. Therefore, F2FS utilizes the NAT, to maintain an identifier of each node and its corresponding block address. Upon an update of a node, only the block address in the NAT is modified to depict the new block address of the node. Finding a node address then checks the NAT entry for the respective node identifier.

Segment Summary Area. In addition to the segment information in the SIT, F2FS tracks information such as the owner of segments in the SSA. Furthermore, the SSA provides a cache for frequently accessed NAT and SIT information.

Main Area Logs. The right half of Figure 4 shows the layout of the main area logs, to which data and inodes are written. F2FS utilizes multiple concurrently writable logs to enhance data grouping and performance, with 3 logs to which nodes are written, and 3 logs for data. An essential mechanism for limiting the required GC, which F2FS must run to free space on the logs, comes from efficient *data grouping*. With data grouping, data that has a similar lifetime is grouped together. Given that data has a similar lifetime, it is likely to be updated within close proximity. Therefore, when GC is run, fewer valid blocks are present, as the data with the same lifetime has likely been updated, reducing the amount of valid data that must be moved by the GC process.

To support data grouping F2FS utilizes the three types of lifetime classes (hot/warm/cold), which are separated into the three different logs for node and data. The lifetime of data can be explicitly set for each file by an application through the passing of a *lifetime hint* with the `fcntl()` function. The Linux kernel provides a total of 5 different lifetime hints, which F2FS reduces to the three lifetime hints it utilizes. If a lifetime hint for a file is not set, F2FS either assigns the default warm lifetime classification, or assigns a lifetime classification based on the file type. With the extension of a file (e.g., `.txt`, `.pdf`), F2FS identifies which files are likely to be updated in the future. Multi-media files (e.g., `.mp4`, `.gif`, `.png`) are less likely to be updated and are directly classified as cold data.

3.6.2 F2FS Garbage Collection

As F2FS is log-structured, over time it contains valid and invalid blocks, similar to the FTL, and must therefore also run GC. In F2FS the process of GC is done at the unit of a section, where valid blocks in all the segments of the section are read and written to a free space, prior to all the segments in the section being freed. In F2FS GC is referred to as *cleaning*, and is run periodically (called *background cleaning*) or when free space for writing is needed (called *foreground cleaning*). The foreground cleaning utilizes a *greedy* approach for finding the section to garbage collect, which results in the largest amount of space being freed by erasing the section. Background cleaning on the other hand utilizes a *cost-benefit* method that considers the required data blocks to be moved during the GC of a section, and the resulting free space that is generated by the erasing of the section.

Given that during F2FS GC block addresses are modified, as the cleaning moves still valid blocks to free space, the metadata is not directly updated to depict these changes, in order to provide recovery. Therefore, F2FS is required to create a checkpoint after each GC call. Similarly, discard commands, issued after GC calls to delete data from the flash SSD, can only be issued after a checkpoint, such that in the case of a necessary recovery, the prior checkpoint still points to existing data that has not been discarded. Once a new checkpoint has been written a discard command can be issued.

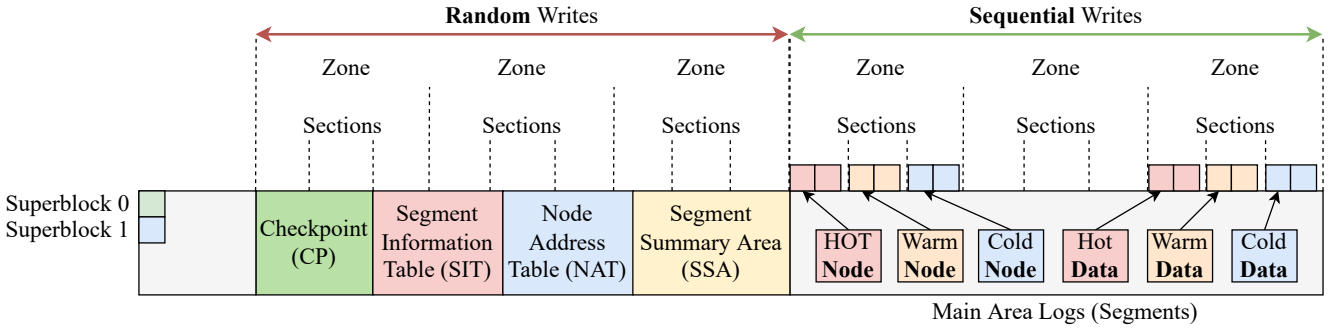

Figure 4: On-device layout of F2FS data. Adapted from F2FS [143].

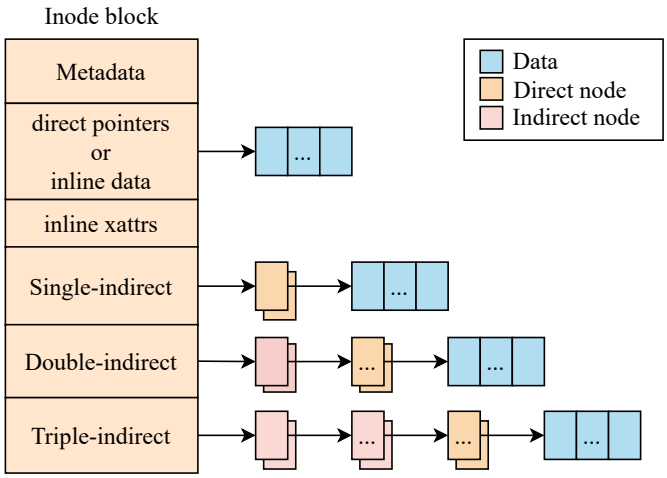

Figure 5: The inode structure of F2FS. Retrieved from F2FS [143].

3.6.3 Aligning F2FS Data Layout with the FTL

To ensure the data grouping of F2FS is similarly depicted by the mapping of data to flash pages in the FTL, F2FS utilizes the separation provided by sections and zones. The goal of sections is to align the F2FS allocation with the GC unit of the underlying FTL. Therefore, the F2FS GC occurring at the unit of a section, matches the FTL GC. Zones are utilized to avoid sections in different zones to be mapped into the same on-device erase unit by the FTL. With a mapping that results in different sections of different lifetimes (e.g., hot and cold data) being in written to the same erase unit on the flash SSD, as is illustrated in Figure 6a, data grouping is violated, furthermore resulting in possible GC overheads if only the hot data is updated while the cold data remains valid. This allocation of inadequately separating sections is referred to as *zone-blind allocation*.

With *zone-aware allocation*, illustrated in Figure 6b, the zone serves the purpose to provide a large enough separation between particular sections, such that the FTL similarly separates the written flash pages for the different file data into different erase units. As a result, only data of similar lifetime is within the same erase unit, resulting in reduction of GC overheads. As the erase unit of flash SSD is commonly hidden from users, the default F2FS configuration utilizes a single segment in each section, and a single section in a zone. However, if the flash storage device characteristics are known, these values can be configured.

3.7 Summary

The foundational building block of flash SSD is based on the *flash cell*. However, the characteristics of flash cells, and their utilization to construct mass storage flash SSD introduce flash management idiosyncrasies, such as having to erase flash prior to updating the data. Due to the resulting lack of in-place updates for flash storage, flash SSD employ firmware called the FTL, that hides the complexity of managing the flash, which however introduces garbage collection overheads. File system design, particularly F2FS, has therefore focused on utilizing flash-friendly data structures and mechanisms to integrate with flash storage. The hiding of flash management idiosyncrasies, specifically the process of GC, however results in unpredictable performance and high tail latency [47, 130, 267]. Therefore, interfaces that expose flash storage characteristics are appearing, with ZNS being the first standardized effort. The *zone* interface of ZNS eliminates the flash SSD GC, moving the responsibility of GC to the storage software on the host (e.g., the file system).

4 Flash Storage Integrations

As there are various methods for integrating flash into storage devices, in addition building full SSD devices, ranging from directly attaching the flash chip to the motherboard, as is common with embedded mobile and IoT devices, or custom integrations of flash chips, we evaluate the file systems based on their level of integration. Given that a different integration exposes a different interface, the possibility to enhance

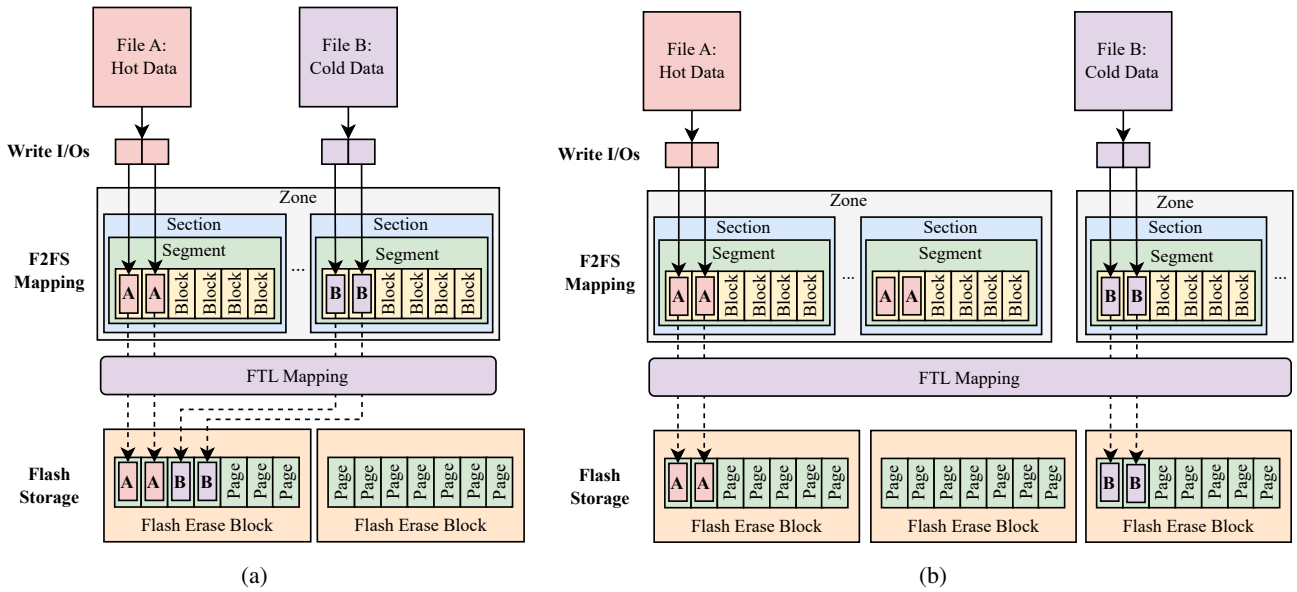

Figure 6: (a) *zone-blind allocation* suffering from inadequate physical data separation, (b) *zone-aware allocation* utilizing larger zones to ensure physical data separation.

particular operations is highly dependent on the integration.

Therefore, throughout this literature study, we divide the relevant work based on the type of flash integration. Figure 7 shows three integration levels for flash, where Figure 7a depicts the conventional integration with a SSD. Figure 7b shows a custom integration of flash storage for devices such as *Open-Channel SSD (OCSSD)* [16, 167], multi-stream SSD [13, 113], and *Software-Defined Flash (SDF)* [188]. The main benefit of these types of integration is that the flash characteristics are no longer hidden behind the device, giving the host an increasing level of storage control. OCSSD is a type of SSD that exposes the device geometry to the host, allowing the host to manage device parallelism and allocation. While such a device allows increased data management for the host software, it comes at increased complexity for managing the device constraints.

Lastly, Figure 7c shows flash integration at the embedded level, such as is commonly used in mobile devices and IoT devices. In embedded flash configuration the flash chip is commonly directly attached to the motherboard, giving the host system full control over the underlying flash storage. Throughout this literature study, we group file system design and mechanisms based on these three integration levels, as different levels of integration allow different degrees of flash management and ranging possibility for flash integration.

5 Challenges of Flash Storage Integration

While SSD uses the same block interface that is used with HDD, flash has different characteristics that software must account for to better integrate flash storage. This section details the challenges that arise from integrating flash storage into systems, providing the guidelines along which we dictate the bottom up view of changes in the host storage software stack, up to the file system. We define the challenges to account for the characteristics of flash storage, as well as enhance its integration into host systems. In the case of SSD devices, these challenges often largely depend on the underlying FTL, as it is making the final decision, independent of what data placement the host implements, however aiding the FTL can increase the performance. Embedded devices provide a higher level of host data placement by eliminating the FTL and directly attaching flash chips to the motherboard. Each challenge is assigned a specific identifier with the *Flash Integration Challenge (FIC)*, in order to refer back to the specific challenge throughout this literature review. Table 3 summarizes the 6 key challenges arising from flash storage devices.

FIC1: Asymmetric Read and Write Performance. On flash storage write operations require more time than read operations [30, 77, 91, 160, 235], making it important for software to limit write operations. Particularly, frequent small writes that are smaller than the allocation unit, referred to as *microwrites* incur significant performance penalties, and should be avoided where possible. Similarly, methods for enhancing the write performance are important to account for the lower write performance, compared to read performance.

FIC2: Garbage Collection (GC). While the FTL hides the flash access constraints from host applications, providing seemingly in-place data updates, it adds the cost of performing garbage collection to free up space. GC overheads have unpredictable performance penalties for the host system [130, 267], resulting in large tail latency [47]. Dealing with, and aiming

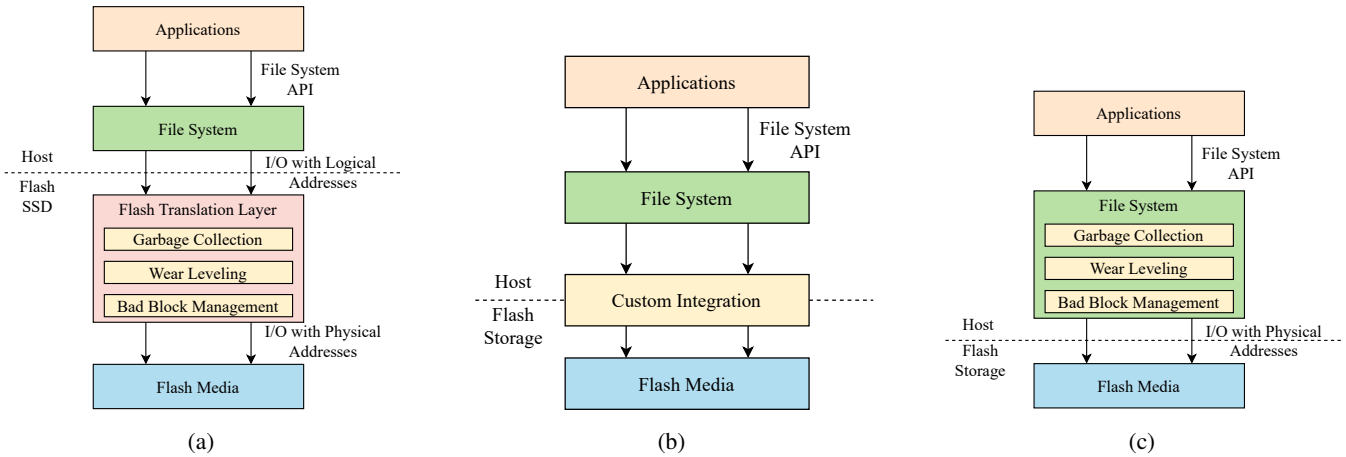

Figure 7: Integration of flash storage into host systems with (a) showing a conventional SSD with a FTL on the storage device, (b) a custom flash integration, through interfaces such as OCSSD, custom FTL, custom device drivers, and multi-stream SSD and (c) flash storage on embedded systems with direct management for flash from the file system.

ID	Flash Integration Challenge	Description
FIC 1	Asymmetric Read and Write Performance	Write operations require more time than read operations [30, 77, 91, 160, 235]
FIC 2	Garbage Collection	The lack of in-place updates results in flash storage running garbage collection to free space and clear invalid pages.
FIC 3	I/O Amplification	The lack of in-place and the required garbage collection introduce write amplification, writing more flash pages than the size of the I/O issued by the host.
FIC 4	Flash Parallelism	The architecture of flash utilizes a high degree of parallelism (channels/chips/planes) to be utilized to enhance performance.
FIC 5	Wear Leveling	Limited lifetime of flash cells requires careful consideration during writes to ensure flash is worn out evenly across the storage space.
FIC 6	I/O Management	Optimizations on the I/O requests, such as merging, aims at leveraging the flash storage capabilities and reducing I/O latency.

Table 3: Overview of the challenges arising from integrating flash storage. The identifier corresponds to the respective *Flash Integration Challenge (FIC)* referred to throughout this literature study.

to minimize required garbage collection for the flash device is a key challenge in integrating flash storage.

FIC3: I/O Amplification. Due to the characteristics of flash avoiding in-place updates of flash pages, writes often encounter *Write Amplification (WA)*. With this the amount of data that is written on the flash storage is larger than the write that is issued by the host system. For example a 4KiB issued write may increase to 16KiB being written on the device, due to possible garbage collection requiring to copy data, resulting in a WA factor of 4x. WA furthermore adds to an increase in wear on the flash cells [168]. *Read Amplification (RA)* similarly is caused by requiring to read a larger amount of data than is issued in the read I/O request. RA most commonly happens when reading metadata in order to locate data, thus requiring an additional read of metadata on top of the request read I/O request for the data. This is most often inevitable, as all data requires metadata for management, however this should be kept to a minimum at application-level. Furthermore, minimization of WA is more important than RA, since write requests have a higher latency than read requests, and writing has a more significant impact on the flash storage, resulting in increased flash wear. While read requests also incur wear on the flash cell, called read disturbance [161], it is not as significant as for write requests.

FIC4: Flash Parallelism. With the various possible levels of parallelism on flash storage devices (discussed in Section 3.3), exploiting of the various possibilities requires software design consideration to aligning with these. Although the I/O scheduling of on-device parallelism, such for channel-level parallelism, is responsibility of the FTL (on devices at SSD

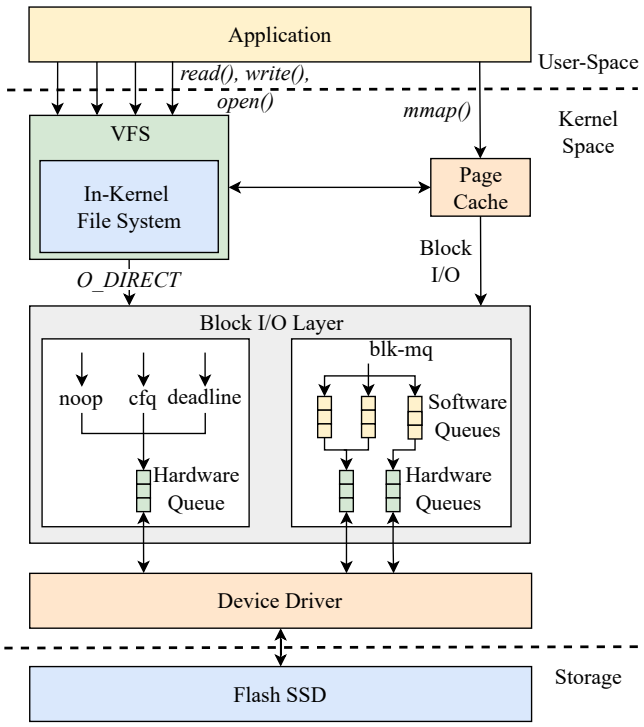

Figure 8: Visual overview of the storage stack in the Linux kernel. Adapted from [62, 178].

integration level), the FTL implements particular parallelism, given that the host I/O requests aligning with the possibility of parallelizing the request, such as with large enough I/Os to stripe across channels and dies. Embedded device and custom flash integrations have more possibility to manage flash device parallelism at the host software level.

FIC5: Wear Leveling (WL). Given limited program/erase cycles for flash cells, even wear over the entire device is required to ensures that no specific areas of the device are burnt out faster than others. Similar to flash parallelism, this largely depends on the flash integration level, as the FTL at the SSD integration level ensures WL, however embedded flash integration and custom flash integration is required to place more significance on ensuring even wear across the flash cells. Strongly related to prior flash integration challenges, wear is commonly a result of GC, which in turn increases the I/O amplification, and particularly the WA and RA [49, 85].

FIC6: I/O Management. As SSD ships with integrated firmware to expose the flash storage as a block addressable storage device, integration into the current software stack is seamless. Figure 8 shows the integration of a flash SSD into the Linux kernel storage stack. Since flash storage devices are significantly faster than prior storage technologies, such as HDD, the storage software stack becomes the dominating factor in I/O latency [21, 22]. One particular optimization for performance of I/O requests to flash storage devices is provided by an I/O scheduler, deciding when to issue I/O

Integration Level	File Systems
SSD Flash Integration	[1, 74, 86, 98, 103, 114, 117, 143, 157, 162, 166, 187, 217, 245, 273]
Custom Flash Integration	[57, 106, 121, 149, 150, 167, 209, 214, 259, 272, 276]
Embedded Flash Integration	[4, 59, 89, 90, 99, 108, 126, 129, 148, 151, 158, 159, 171, 172, 184, 193, 194, 197–199, 201, 222, 244, 257, 269, 277, 278, 283]

Table 4: Classification on the flash integration level utilised for the file systems evaluated in this literature study.

requests to the storage device. As is visible in Figure 8, the block I/O layer implements various schedulers with different functionality. Providing a ranging degree of optimizations for I/O requests, such as varying scheduling policies and merging of I/O requests, or possible reordering, specific configurations are favorable to increase performance with flash storage. Particularly the utilization of multiple queues, with multiple software and hardware dispatch queues (visible in the `blk-mq` configuration of the block I/O layer), allows better exploitation of flash storage capabilities, and avoids certain Linux kernel overheads. Furthermore, evaluating mechanisms that reduce the latency of I/O operations, and particularly write I/O operations.

5.1 Flash Integration Organization

With the different possible levels for integration of flash storage (recall Figure 7), and while mechanisms for solving flash integration challenges are frequently applicable at various integration levels, several of the mechanisms we present require deeper integration of flash storage, levering increased control, in order to be implemented. For instance, the incorporation of the various levels of on-device parallelism is not directly possible at the SSD integration level, as the FTL hides the parallelism on the physical device from the host system. The custom and embedded level flash integration provide the host with more possibility to manage these. In order to separate the possibility of mechanisms to be implemented with a particular flash integration level, Table 4 provides a classification of each evaluated file system in this literature study to the respective integration level. During the evaluation we present the different mechanisms to solve a FIC, and indicate which file systems utilize these. Therefore, when considering the feasibility of a mechanism for a particular flash integration consult this table to see its applicability. Note that file systems are not limited to the classification we provide, as for instance file systems designed for SSD flash integration also work on some embedded flash integration. However, we utilize only a

Mechanism	File Systems
Write Optimized Data Structures (§6.1)	[57, 89, 103, 217, 245]
Write Buffering (§6.2)	[105–107, 117, 167, 198–200]
Deduplication (§6.3.1)	[57, 86, 158]
Compression (§6.3.2)	[57, 90, 99, 185, 257]
Delta-Encoding (§6.3.3)	[57, 86]
Virtualization (§6.3.4)	[151, 283]
Flash Dual Mode Switching (§6.4)	[148, 259]

Table 5: Mechanisms for file systems to deal with **FIC1**, asymmetric read and write performance of flash storage, and the respective file systems that implement a particular mechanism.

single classification for each file system to avoid confusion. Exceptions are made only in specific cases where an existing file system is adapted for a different flash integration level.

We divide the discussion of mechanism by the FIC for which the evaluated study presents a novel solution. This implies that mechanisms that solve multiple FIC are discussed in detail in the first section they appear in, however are also mentioned in all latter sections for the FIC that the mechanism solves. Therefore, each FIC section contains a table of the respective mechanisms presented to solve that particular FIC, along with a reference to the corresponding section of its detailed discussion.

6 FIC-1: Asymmetric Read and Write Performance

Given that read and write performance on flash storage is asymmetric [30, 91, 235], this section provides the various mechanisms to handle the asymmetric performance. Table 5 shows the various methods discussed in this section, for dealing with asymmetric performance, and how to increase file system performance. Such methods and mechanisms include data structures particularly optimized for characteristics of flash storage, efficient methods of data caching, and effective organization of data on the storage device.

6.1 Write Optimized Data Structures

The characteristic of flash having lower write than read performance requires that data structures for flash are write optimized. Such data structures which are optimized for write operations are referred to as *Write Optimized Data Structure (WODS)* [11]. With the addition of the missing support for in-place updates on flash, the best suiting data structure for flash-based file systems is a log-based structure. As a result, all file systems discussed in this section are LFS. The nature of a LFS being append-only writes accounts for the lower write performance on flash, which matches the write updates to the operations more optimal for flash, which are smaller fine-fine grained updates [169] in a log structured fashion [143].

However, while the log provides increased write performance, metadata is scattered throughout the log, requiring a full scan of the log to locate metadata. Therefore, file systems commonly employ tree-based data structures for metadata, decreasing the worst case time complexity from $O(n)$ to $O(\log n)$. B-tree is a commonly used data structure for storage systems, including databases [39, 71] and file systems [217]. Nodes in a B-tree can be larger than in conventional binary search trees, allowing the node to align to a unit of the underlying storage. B-trees furthermore are maximizing the breadth of the tree, instead of its height, in order to minimize the I/O requests in order to locate data, since each traversal to a child node requires an I/O request. The tree itself is sorted and self-balancing, making the worst case complexity for search relative to the tree height, however also requiring to balance the tree. The B+tree further reduce required I/O by only having data in the leaf nodes, such that higher nodes only contain keys, increasing the number of keys that fit inside a block. Leaf nodes are linked in a linked list, for faster node traversal, which in turn speeds up searching.

In order to write optimize these trees, B$^\varepsilon$-trees [11, 19] adapt the node structure of the B-tree to include a buffer to which updates to its children nodes are written. As node updates are initially written in memory, this allows to gather a larger number of small writes, encode these in the added buffer of the respective nodes, and write these as a larger unit, avoiding frequent small updates to nodes. The most recent version of BetrFS [103] (published in 2022) implements such a B$^\varepsilon$-tree as the metadata storage for indexing data. It is implemented in the Linux kernel as a key-value store, based on TokuDB [204], which exposes a key-value addressable interface. The benefit of this B$^\varepsilon$-tree is that the nodes have a larger (2-4MiB) sequentially written log, batching updates into larger units and thus avoiding small updates. Initially all updates go into the root node message log, which when full gets flushed to its child nodes. While WODS provide optimized write performance by batching updates, read requests require reading an entire node (2-4MiB), causing small read requests to suffer from significant read amplification.

SSDFS [57] adapts the tree design to utilize hybrid nodes, since the node allocation uses blocks, which are several KB in size, and may not directly be filled directly. Therefore, hybrid nodes in the tree adapt the size of the node, such that if a hybrid node is allocated and filled, it allocates an additional hybrid node, which upon being filled is merged with the first hybrid node and becomes a leaf node. This allows to reduce write amplification (solving **FIC3**) if a node is not filled enough. An additional optimization to B-trees is the *Parallel I/O B-Tree (PIO B-Tree)* [218], which implements a paral-

lel flash-optimized B-Tree variant (solving **FIC4**), utilizing a larger I/O granularity to exploit package-level parallelism, maintain a high number of outstanding I/O requests to utilize the channel-level parallelism, and avoid mixed read and write operations.

6.2 Write Buffering

In addition to WODS optimizing write operations file systems need to also avoid small writes, referred to as *microwrites*. Particularly, as file systems write in units of blocks, which commonly are 4KiB, small writes require a full unit to be filled. The majority of file systems provide the possibility for inline data in inodes, such that the inode data, which is written regardless, has a small capacity to include data. However, this still requires that for small files inodes are directly written to flash. Therefore, several schemes that involve buffering and caching of data in memory, before flushing to the flash storage, provide increased write performance, and additionally increase overall performance.

While buffering of I/O prior to flushing to flash storage provides performance gains, buffers are often limited in size and are much smaller than the persistent storage. Furthermore, in addition to buffering write requests for new data, accessing or updating existing data is also cached in the buffer. Therefore, the caches utilize effective methods that minimize the cache misses by optimizing the eviction policy. *Least Recently Used (LRU)* [241] is a common caching policy that maintains the items in the cache in the order of usage, where the least recently used item is selected for eviction. This mechanism is extended by DFS [106, 107], which utilizes *lazy LRU*, which does not insert accessed data into the buffer upon a cache miss, but rather inserts the data into the buffer only on an additional cache. Requiring of two cache misses implies that the cache only contains data that is frequently accessed, instead of caching all data from a cache miss.

Clean First Least Recently Used (CFLRU) [200] is another extension on the LRU algorithm, which splits the buffer into two segments, one as the working region in which recently accessed pages reside, which are managed in *Most Recently Used (MRU)* fashion, therefore depicting the frequently accessed pages. The second region, called the clean-first region contains the less commonly accessed pages in LRU fashion. On eviction (e.g., when writing a new page and freeing space in the buffer), it first attempts to evict a clean page, rather than a dirty page from the clean-first region, as this does not require a flush of the dirty data to the flash storage, and only resorts to evicting dirty pages as last resort. *Flash Aware Buffer (FAB)* [105] optimizes the caching policy to align with the flash characteristics by organizing pages in the cache in a larger unit, called *block*, and upon eviction flushes entire blocks to the flash storage, issuing larger I/Os and aligning better to the flash erase unit.

Similarly, NAFS [198] implements a *double list cache*, containing a clean list with only clean pages for caching of data, and a second dirty list for writing of modified pages and to prefetch of pages based on the access patters, only containing dirty pages. The benefit of two separate lists is that firstly it allows caching write operations and avoid small writes to the flash device, and secondly it prefetches data into the clean list in order to minimize cache misses. EnFFiS [199] presents the *dirty-last cache* policy, which considers the flash characteristics by utilizing a *delay region* and a *replacement region*, which correspond to a multiple of the flash blocks. By using a multiple of the flash blocks, the data written is sequentialized and written to the flash as a larger unit, where dirty pages are initially moved from the replacement region to the delay region, before being flushed to the flash. This buffering in the delay region allows collecting more dirty pages, improving performance and avoiding smaller writes to flash.

StageFS [167] likewise utilizes two stages, where write operations are initially written into the first stage, called the *file system staging area*. The issued writes to the staging area are completed in a *persistence-efficient* way, which utilizes a log to account for asymmetric flash performance for the staging area. Subsequently, writes are then regrouped, based on the file system structure and hot/cold identification, and are written to the second stage, based on the group assignment. The staging area allows writing synchronous I/O directly to the staging log with optimal flash write characteristics, lowering completion latency, followed by better grouping of data when writing from the staging to the second stage file system area.

In an effort to limit the required memory for data caching, DevFS [117] proposes the *reverse caching* mechanism for effectively managing host memory and device memory. Reverse caching aims at keeping only active files in the device memory, which is limited in size, and upon closing of a file migrates the metadata to the host memory. Reopening a file migrates it back to the device memory, and consistency of metadata is not violated as any actively modified metadata is in the device memory, and only inactive metadata is in the host memory. To efficiently utilize reverse caching, DevFS uses a host-memory cache that is able to use *Direct Memory Access (DMA)* to move metadata between itself and the device memory, additionally minimizing host overheads.

6.3 Reducing Write Traffic

In order to avoid the slower write performance of flash storage another mechanisms aims at minimizing the amount of data that is written to the storage device. Whenever data is updated on the flash storage, updating all metadata and writing the new data incurs a significant amount of write traffic, in addition to causing WA. Therefore, reducing the amount of data being written through mechanisms such as deduplication, compression, and delta-encoding helps at avoiding the

increased WA.

6.3.1 Deduplication

File systems commonly contain duplicate data, from backups or archival copies made and general work. Therefore, file systems often aim to avoid creating duplicates of existing data, which is referred to as *deduplication*. Evaluations show that in high-performance computing centers on average about 20-30%, with peaks of 70%, of the stored data can be removed through deduplication [175]. Deduplication avoids writing the same data multiple times, which in turn helps reduce the write traffic and minimizes write and space amplification (solving **FIC2**) and prolong the device lifetime. Effective deduplication relies on hash functions, which provide a deterministic output, called the *digest* or commonly referred to as the *fingerprint* of the data, based on which duplicates can be identified. The utilized hash functions for deduplication are *one way hash functions* [176], which calculate a fingerprint of the data, or digest, but given the digest or fingerprint, the data cannot be generated. Given data from a file for instance, hashing the data provides a digest, and if the same data is to be written again, the same digest is generated. By identifying if a digest already exists in the file system, can it identify if the data is being duplicated, and avoid writing the duplicate by making the metadata of the newly written data point to the existing data.

In order to apply deduplication on file system data, two methods exits. **(1)** Deduplication with fixed-sized chunks, where hashes are generated based on the entire file or a predetermined chunk size, and **(2)** variable-sized chunks where the chunk size that is hashed depends on the file and content. The effectiveness of deduplication with fixed-sized chunks is limited and highly dependent on the chunk size and modification sequences in the chunks [158], as for example a small change in a large file, where the chunk size is the entire file, results in a completely different hash, even if much of the data is duplicated. While variable-sized chunks reduces the configuration dependability it is significantly more complex to implement. DeFFS [158] implements a duplicate elimination algorithm that reduces the complexity of identifying duplicates. As comparing duplicates of every byte is not possible, the algorithm finds the smallest modified region in a file adapting the chunk size, which is initially assigned the default value that corresponds to the flash page size.

The idea of eliminating duplication through fingerprints is not new for SSD, as existing FTL implementations implement such mechanism. *Content Aware FTL (CAFTL)* [32] generates collision-free fingerprints of write requests and maintains fingerprints of all flash resident data in order to avoid writing the same data twice. Flash Saver [152] is a similar architecture running between the SSD and the file system, which manages the file system I/O requests and ensures deduplication using SHA-1 fingerprints [252]. A plethora of similar hash-based deduplication mechanisms exist [65, 75, 133, 261], which may not be particular to file systems, but can be adopted by a wide range of storage systems. File systems with deduplication are CSA-FS [86], which implements deduplication by calculating the MD5 digest [216] (MD5 is a hash function applied to the input) and looks up the result in a hash table which provides the corresponding LBA of the block. If it already exists and the user requested a new file with it, the new inode simply uses the given LBA from the hash table, instead of duplicating the data again. Similarly, SSDFS [57] maintains fingerprints with its metadata B-Tree, and DeFFS [158] stores hash keys with all inodes for their data to avoid duplicates.

6.3.2 Compression

An effective method for reducing the required storage space for data is to utilize *compression*. While there exist lossy compression algorithms [94, 120], which discard parts of the data, and lossless compression algorithms [120, 228], which maintain all data, lossy compression is very application dependent, and in the case of file systems we assume a lossless data storage and therefore focus on lossless compression algorithms. Lossless compression methods rely on three methods. **(1)** *Run Length Encoding (RLE)* which minimizes size by taking the repeating characters, referred to as the run, and replacing them with a 2 byte sequence of the number of repetitions of the character, called the run count, followed by the replaced character. For example the sequence of “aaabbccc“ is represented as “a3b2c3“ with RLE. **(2)** *Lempel-Ziv (LZ)* [281, 282] which utilizes a dictionary to replace strings with their dictionary value. Variations of the LZ algorithm exist, such as LZ77 [281], LZ78 [282], and LZW [255]. The dictionary is constructed at the compression time and used a decompression time [118]. A well-known compression algorithm GZIP [50] is based on two LZ algorithms. **(3)** Huffman Coding [88] which creates a full binary tree based the frequency of occurring characters and generating code of the character and the frequency.

JFFS [257] and several other file systems [42, 57, 70, 90, 99, 185, 257] provide a data compression, however metadata compression has shown to cause decompression overheads [115]. Applying compression at the file system level allows reducing the utilized storage space, especially on flash this reduces the space and write amplification factors (solving **FIC3**), and in turn prolonging the flash lifetime as shown by Li et al. [153] on the benefits of different compression algorithms for NAND flash lifetime. SSDFS [57] utilizes LZO compression [186], a library for data compression with LZ algorithms, for data and metadata. Given that different data benefits from different compression mechanisms, where for instance data is less frequently read can be compressed more efficiently than frequently read data, which may require more simplistic decompression to minimize overheads, or embed metadata in the compressed data to avoid decrypting data and metadata separately. Adaptive compression selection is employed in an

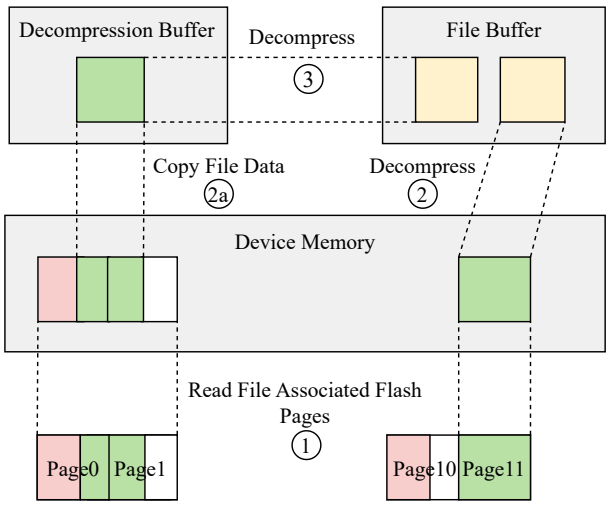

Figure 9: The benefit of *Page Boundary Alignment (PBA)* during decompression arises from the elimination of copying associated file data.

extension of F2FS [99], where *File Access Pattern-Guided Compression (FPC)* selects the compression method for files based on how compressible they are.

An issue of compression is that it must be aligned to the flash unit, in order to avoid unnecessary read amplification and required data copying. Figure 9 shows that if the compressed data is not aligned to the flash unit of a page, such that compressed data can extend across pages, the decompression process becomes more complex. This is due to first having to read all the associated flash pages into the device memory. However, if compressed data can extend across flash pages, this implies that non-associated data can be included. Therefore, only the associated data must be copied to a *decompression buffer* (step 2a), from which the data can be decompressed (step 3). LeCramFS [90] (short for less crammed FS) implements a read-only file system with compression for embedded devices, that avoids this additional copy of data by using PBA. Any compressed data will not extend across page boundaries, implying that no additional pages with non-associated data are read, and the data can be decompressed directly (step 2). Therefore, it avoids the additional copy of data and reduces the read amplification (solving **FIC**3). In order to avoid wasting space if the page is not fully utilized from a single compression, LeCramFS extends the compression implementation with a *partial compression*, which splits the data into parts that fit into the available space. These parts are then compressed separately, allowing to utilize all available space.

6.3.3 Delta-Encoding

Similar to deduplication, delta-encoding lowers the write traffic, further limiting the space and write amplification (solving **FIC2**) by instead of writing out entire data when updated, with delta-encoding the new data is compared to the old data and only the differences, called the *delta*, are written. This is especially beneficial in the case of small changes in large files, where it avoids writing the entire file again and writes only the changes in the data. While delta-encoding provides a similar space reduction to compression, particular file characteristics of what is being encoded can affect resulting performance. For instance, data sets that have a significant redundancy across them, such as email data sets, require significantly less space than being compressed [56]. However, non-textual data, such as pdf documents, will have large deltas for even small changes, where compression is likely to be a better choice. Therefore, the application of delta-encoding depends on the data characteristics and what type of content is being encoded, in order to achieve better utilization of delta-encoding compared to simpler compression. CSA-FS [86] implements delta encoding on its file system metadata (superblock, descriptor, bitmaps, and tables), allowing minor updates such as the access time of metadata and bitmaps to only write the new changes. Similarly, SSDFS [57] uses delta-encoding for user data.

6.3.4 Virtualization

As Butler Lampson mentioned in his famous quote from 1972 (which was originally stated by David Wheeler), “Any problem in computer science can be solved with another level of indirection“ [140]. By adding a layer on flash storage the write traffic can be reduced as a result of avoiding metadata update propagation (as with the wandering tree problem [14]). Adding a virtual layer is conceptually identical to *Logical Block Address (LBA)* on top of PBA. Figure 10 shows an example of how virtualization maintains the same *Virtual Block Address (VBA)* when file data is updated, eliminating a need for metadata updates. Note, for simplicity we assume the virtual layer to be on top of the physical layer, providing PBAs, however it can be directly on the physical layer of the flash storage, depending on the flash integration level, and would therefore be using *Logical-to-Physical (L2P)* mappings. In the illustrated scenario, three blocks are mapped to PBA0, PBA1, and PBA2, respectively. Assuming these are part of a file, metadata points to these respective virtual block addresses. If the first block of the file is modified, a new data block is written at PBA3. With virtualization, the VBA remains unchanged, only the V2P mapping table is modified to depict the new mapping of PBA3. As a result, file metadata remains unchanged, avoiding write amplification for updating file metadata.

A similar mechanism is implemented in NANDFS [283], using a *sequencing layer* that implements the block allocation as immutable storage on the logical layer. Therefore, when file system data is updated, it cannot overwrite and simply marks the data as obsolete, and the new updated data is written

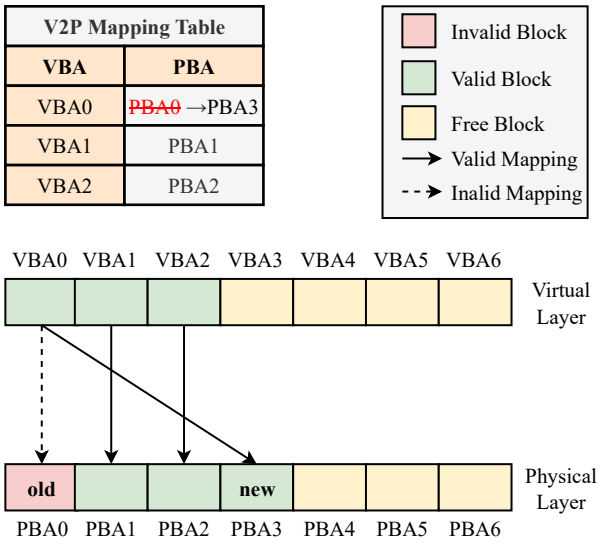

Figure 10: A simplified example scenario of virtualization on top of the physical storage.

in a newly allocated block. The sequencing layer implements the L2P mapping, such that the LBA of the new data is the same as the old LBA, avoiding the updating of data addresses in the file system metadata. This implies that the file system does less writing by eliminating metadata updates, which in turn also reduces the required GC.

The concept of virtualization can similarly be applied through *Address Remapping (AR)*. With AR only the L2P mapping table of the storage device is modified, which similar to virtualization avoids rewriting of metadata. AR is an effective mechanism for solving additional overheads in file systems, such as file system garbage collection [112], journaling [36, 112, 254], and data duplication [279]. Lee et al. [151] propose *Remap-Based In-Place-Updates (RM-IPU)*, an address remapping scheme implemented in F2FS to solve issues of out-of-place updates. Instead of applying AR into a single file system operation, RM-IPU includes all write operations to utilize AR. All updated data is first stored in the log, followed by AR to update data pointers, thus avoiding metadata updates for file overwrite operations. File write operations to new files are appended to the log as usual, and the contiguity in LBAs for files is maintained.

6.4 Flash Dual Mode Switching

Some modern flash devices allow the host to switch the cell level of underlying flash blocks (e.g., switching MLC to SLC) [155, 259]. This provides the benefit that a lower cell level, representing fewer bits, has a lower read, write, and erase latency, thus providing the flash with a higher performance than with a larger cell level configuration [259]. This switching between flash modes is referred to as *flash dual mode*, which however comes at the cost of being able to store less data in the same amount of flash, as the block is only able to store half the data in the example of switching from MLC to SLC. DualFS [259] utilizes the flash dual mode feature to provide a dynamically sized SLC area, alongside the remaining MLC area, to accelerate the performance of critical I/O requests. By evaluating the I/O queue depth of I/O requests, DualFS determines the criticality of the incoming request, and maps it to the SLC area for increased performance. It further profiles incoming request based on the hotness and allocates hot data into the SLC mode for lower request latency.

FlexFS [148] implements a similar mechanism for increased performance on the SLC area. The drawback of a design that utilizes a SLC area for incoming write I/O, is that data is required to be moved from the SLC area to the MLC area as it has a lower write lifetime. With the SLC are being used for critical I/O requests that require lower completion latency, it introduces *data migration overheads*. To solve this, FlexFS implements several migration techniques that aim to hide the overheads for the data migration from the host. The first technique revolves around *background migration*, which pushes the migration to happen when the system is idle. The second technique is *dynamic allocation*, which writes non-critical requests to the MLC area, saving on flash degradation in the SLC area, as well as avoiding future data migration. The dynamic allocator functions based on measurements of prior system idle times from I/O requests, to predict current idle time, and if sufficient idle time is predicted in order to complete data migration, the data is written to the SLC, and otherwise part of the data, depending on required migration and idle time, is written to the MLC area. The last technique, *locality-aware data management*, takes into account the hotness of data, and the dynamic allocator attempts to migrate only cold data from the SLC area.

6.5 Summary

With the write performance of flash storage being lower than its read performance, particular attention is paid to designing effective mechanisms and methods for achieving faster write performance. Effective caching methods, buffering data before writing, allow to reduce write latency. Similarly, mechanisms that reduce the write traffic to the device, such as deduplication, delta-encoding, and virtualization allow are effective for dealing with the lower write performance of flash storage. Lastly, the possibility of flash-dual mode switching allows to change the flash cell level to provide lower latency write request completion for critical write request.

7 FIC-2: Garbage Collection

A significant challenge of flash storage is GC overheads having unpredictable performance penalties for the host system [130, 267], resulting in large tail latency [47]. Dealing

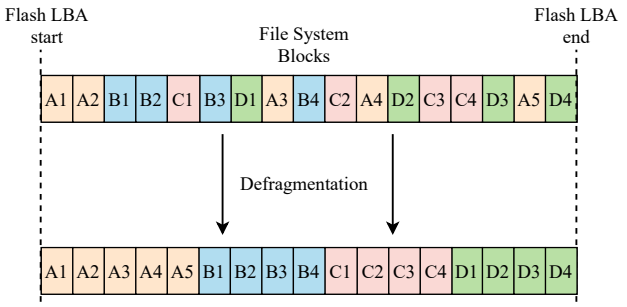

Figure 11: Illustration of single file fragmentation on the storage over time, as parts of file data are overwritten and, due to the LFS design new data is appended at the head of the log.

with, and aiming to minimize required garbage collection for the flash device is a key challenge in integrating flash storage. Naturally, as flash storage does not provide in-place updates, data is written in a log-based fashion, sequentially in the flash blocks. Therefore, over time as data is overwritten, the blocks contain an increasing number of invalid pages that must be erased to free space. However, as the block also contains valid data, and the erase unit is a block, the still valid pages are moved to a new block, such that the old black can be erased.

In a LFS updates to file data are written at the head of the log, resulting in the parts of the file that are updated to be located in a different flash block than the parts of the original file data. Furthermore, GC causes file data to moved around the storage space as well, resulting in scattering of parts of files, referred to as *fragmentation*. Figure 11 shows the resulting fragmentation for several files that occurs over time from file updates and GC. Fragmentation results in increased read time [30], due to files being in non-contiguous regions, and introduces increased garbage collection overheads due to the failed grouping of data. Ji et al. [100] show an empirical study on fragmentation in mobile devices, identifying that fragmentation introduces performance degradation due to increased I/O requests, and it further produces increased pressure on host caching. The effects on caching are due to increased difficulty of prefetching data, since it no longer is in contiguous physical ranges, but scattered throughout the physical space. Therefore, prefetching cannot bring correct data into the caches, resulting in an increase in cache misses.

In addition to increased I/O requests for reading and rising cache pressure, increased garbage collection is caused when frequently modified files or file fragments, referred to as the *hot data*, are in the same block as rarely accessed files, referred to as the *cold data*. When hot data is modified, its flash pages are invalidated. Once enough flash pages in a block are invalidated, it can be erased. However, if it is co-located with cold data, the cold must be copied to a free space, as it is still valid. If this cold data is then again co-located with hot data, the modifications of the hot data cause the block to be cleared during GC, which requires the cold data to be moved again. Therefore, co-locating hot and cold data in the same physical erase unit results in significant GC increase due to the unnecessary moving of the cold data.

Mechanism	File Systems
Reducing Write Traffic (§6.3)	[57, 86, 90, 99, 151, 158, 185, 257, 283]
Aligning the Allocation Unit (§7.1)	[197, 245]
Data Grouping (§7.2)	[143, 159, 214, 277, 278]
GC Policies (§7.3)	[1, 74, 193, 272]
Coordinating the Software Stack Layers (§7.4)	[149, 150, 259, 276]

Table 6: Mechanisms for file systems to deal with GC overheads from flash storage, and the respective file systems that implement a particular mechanism. Green highlighted table cells depict previously discussed mechanisms with their respective section.

Reducing the write traffic to the storage device, an effective method to handle the asymmetric flash performance (see Section 6.3), is a solution to minimize fragmentation and reduce possible future GC overheads. However, additional mechanisms are required for effective GC management. With fragmentation being a significant contribution to increased garbage collection, avoiding it is a key objective to reducing garbage collection overheads. Fragmentation is classified into three different types [221]. Firstly, *single file fragmentation*, where data in a single file is dispersed over the storage (as is shown in Figure 11). Secondly, *relevant file fragmentation*, where files that are relevant to each other and should be grouped together are split over the storage, such as co-locating hot and cold data in the same erase unit. Lastly, *free space fragmentation*, where the file system has a large amount of small free space, because of deletion of dispersed small files. The cause of fragmentation occurring over time is referred to as *file system aging* [232]. While several tools exist that implement *defragmentation* [76, 128, 195], additional mechanism can be utilized to avoid fragmentation and GC overheads. Table 6 depicts the mechanisms for file systems to deal with and minimize garbage collection overheads. The process of countering the different types of fragmentation is commonly referred to as *storage gardening* [122, 123], and for file system development various aging tools exist in order to generate real-world file system workloads and simulate file system aging [44, 111].

7.1 Aligning the Allocation Unit

GC is a result of having to move valid blocks in the erase unit to a free space, in order erase the flash block. While data

grouping allows to align the validity inside the block, such that blocks are likely to be updated within close proximity, multiple files may be co-located in the same block. Therefore, a similar method is to align the allocation unit of data blocks for a single file to the erase unit, resulting in only a single file being located in a block. Such a mechanism is implemented in URFS [245], which aligns the data allocation unit for large files to the flash erase unit, allowing files to be erased as a single unit, limiting required GC. However, as the erase unit of flash can be several hundreds MBs, resulting in significant over-allocation for small files, it makes such a mechanism only beneficial with large files. NAMU [197] similarly showcases a file system that aligns its content with the requirement of large files. Focused on the multimedia domain, where files have the particular characteristics of rarely being modified, and if removed all file data blocks are erased in one unit, GC in NAMU is done at the granularity of a file. In addition to improving on GC, the memory requirements for mapping tables are also minimized. For generic file systems that vary in file characteristics, *Adaptive Reserved Space (ARS)* [269] minimizes the issue of over-allocating space by allocating in a smaller unit of 2MB (a single segment). File data is written to the space until it is exhausted, upon which a new segment is allocated. While this does not map entire files to the flash erase unit, it allows writing of file data sequentially for each segment, eliminating fragmentation to a degree. Therefore, the resulting reduction in fragmentation provides benefits in reducing the amount of data that is required to be moved during GC.

7.2 Data Grouping

A key circumvention method for fragmentation relies on grouping of related data. Most commonly this is applied in the type grouping data by its access and modification frequency into hot and cold data, however other less commonly used groupings based on *death-time prediction* exist [24]. Commonly more classifications than simply hot and cold are utilized for more effective grouping. A plethora of methods for grouping data in such a way have been proposed, which we split by its application type into several groups.

7.2.1 Data Type Grouping

Common write patterns in storage systems follow a bi-modal distribution, where many very small write requests and a numerous very large write requests are issued [27]. This stems from the fact that small changes are caused by metadata updates, which occur the most frequently, whereas large changes are file updates. Given that metadata is more likely to be updated frequently, separating metadata from data improves on required garbage collection. Specifically, since metadata is often updated even if the data is not updated, for example in scenarios where the file attributes (access time, permissions, etc.) are updated, or the file is moved. Therefore, based on the request size, the data can be classified to be metadata and be grouped accordingly. F2FS also groups based on the data type, where metadata is considered to be always hot data, which is implemented by similar file systems [159].

Dividing of file system data is similarly applied in ELOFS [277, 278], which splits the flash storage two partitions, where a directory partition contains the data of directory entries, and a data partition contains the file system data, which is compacted with the inodes. Jung et al. [110] propose the addition of classifying data based on the *Process Identifier (PID)*, as a process is likely to generate similar access patterns and data types throughout its lifetime, classifying by the PID allows to indirectly infer a data type. A similar data type separation is implemented by Fstream [214], for which the authors modify ext4 [20] and xfs [238] to map different operations to different streams on a stream SSD. Ext4Stream, the modified ext4 to support streams, maps different metadata operations to different streams, including the journal writes for consistency, the inode writes, as well as different streams for the directory blocks and the bitmaps (inode and block). Furthermore, it utilizes different streams that can be created for different files and for different file extensions. The goal of such streams is to map particular files, such as `LOG` files (temporary hot data) for key-value stores for example to a particular stream, separating its access patterns from that of other files and file system data. Similarly, the modified XFStream utilizes different streams for the log, inodes, and specific files.

7.2.2 Dynamic Grouping

While grouping is an effective method for minimizing GC, it commonly relies on a static definition on classification targets for the number of hot/cold degrees to classify to (e.g, hot/warm/cold). Shafaei et al. [227] identify that the majority of hot/cold data grouping methods fail to account for the accuracy in the hot/cold grouping mechanism, as well as relying on an individual classification of each LBA, making the management of increasingly larger flash storage difficult. Therefore, Shafaei et al. [227] propose an extent-based temperature identification mechanism. It is based on the density stream clustering problem [33, 64, 93, 101], which is a common approach of classification in artificial intelligence and stream processing, however has not been applied to storage before. The density-based stream clustering groups data in a one dimensional space as the data arrives, hence its applicability for stream processing.

Applying this method to storage, the one-dimensional space is the range of LBAs, and extent-based clustering splits the available space into a number of extents to group by. Initially, the entire space is a single extent and as writes occur the extent is split into smaller extents with different classifications. Over time as more writes are issued, extents are expanded and merged (merging of extents with the same classification).

Such a grouping allows a more detailed grouping due to the increase in classification targets, compared to binary hot/cold grouping. However, an evaluation by Yang and Zhu [270] on a configurable garbage collection policy, where the number of hotness classification targets is evaluated, shows that various hotness classification targets can significantly increase the write amplification during garbage collection.

7.2.3 LBA Hotness Classification

Different to grouping data based on its type, hotness can be classified on the LBA based on its access frequency. The naive approach at modeling hotness for each LBA is with table-based classification model [84]. This however comes at a high overhead cost, as an entry for each LBA is needed, which becomes increasingly expensive as flash storage grows. Therefore, a more non-trivial method is based on two-level LRU classification [28], with two LRU lists. Upon an initial LBA access, the LBA is stored in the first list, and a subsequent access moves it to the next list, which is referred to as the *hot list*. Therefore, if a LBA is in the hot list, it is considered to be frequently accessed.

A different approach is implemented by *Multiple Bloom Filters (MBF)* [97, 192], which uses bloom filters to identify if a LBA is hot. Bloom filters rely on a hash function that, given an input such as the LBA, provide an output which is mapped to a bit array, and sets the bit to true. Therefore, if a LBA is accessed, applying the hash function sets the respective bit in the array to true, and checking if a LBA is hot simply applies the hash function and checks if the bit is set. However, depending on the length of the array, multiple LBAs can map to the same bit location, as proven by the pigeonhole principle [3], resulting in false positive hotness classifications for a LBA. To avoid frequent false positive classifications, MBF utilizes multiple bloom filters at the same time. With multiple bloom filters, the same amount of arrays exist, applying all bloom filter hash functions to the LBA, and setting the respective bit in each of the arrays. As a result, collisions on all bloom filter hash functions are less likely, minimizing the possibility for false positives.

Similar to MBF, Kuo et al. [137] present a hot data identification method by using multiple hash functions and a hash table. Upon a write, the LBA is hashed by multiple hash functions, and a counter for each hash function is incremented in a hash table. To check if a LBA is hot, the LBA is hashed and a configured H most significant bits of the resulting hash table indicate if the LBA is hot if they are non-zero, as the counter is increased on accesses the most significant bits are only non-zero if the LBA is frequently accessed. Multiple hash functions are used for the same reason multiple bloom filters are used in MBF, to avoid false positive classifications. Lee and Kim [145] provide a study into comparing performance of two-level LRU, MBF, and *Dynamic Data Clustering (DAC)*, which are similar to the density-based stream clustering by Shafaei et al. [227] (discussed in the prior Section 7.2.2). The authors show that on the evaluated synthetic workloads DAC provides the highest reduction in write amplification factor, which in turn leads to a decrease in GC overheads.

Unlike all prior approaches basing classification on the access frequency directly to identify hotness, Chakraborttii and Litz [24] propose a temporal convolutional network that predicts the *death-time* of a LBA, based on modification history. This allows to more optimally group data based on the death-time of individual LBA, which has been shown to be an effective grouping mechanism [81]. Grouping related death-time LBA reduces the required garbage collection, as blocks containing LBAs with similar death-times are erased together, which in turn reduces the write amplification (solving **FIC3**).

7.3 Garbage Collection Policies

While data grouping provides benefits of co-locating data based on their update likelihood, an additional essential part of garbage collection is the policy of victim selection for segments to clean. Since during garbage collection a segment to clean is required to be selected, where all still valid data in the segment is moved to a free space, selecting a victim becomes non-trivial. With the importance of data grouping with hotness for effective GC, conventional GC policies, such as greedy and cost-benefit lack their inclusion. SFS [1] proposes the *cost-hotness* policy to account for the hotness of segments instead of the segment age, better incorporating the data grouping into victim selection. The cost-hotness is calculated as

$$\text{cost-hotness} = \frac{\text{free space generated}}{\text{cost} * \text{segment hotness}} = \frac{(1-u) * \text{age}}{2u * h}$$

where the cost considers reading and writing the valid blocks (equivalent to $2u$) with the segment hotness. A further GC policy that is based on the cost-benefit policy, is the *Cost-Age-Time (CAT)* policy [35]. It extends cost-benefit by including an erase count for each block, improving wear leveling (solving **FIC5**).

The majority of GC policies have a fixed algorithm, limiting configuration possibility. The *d-Choice* algorithm [247] is a configurable GC policy that combines greedy selection with random selection. The tunable parameter d defines the number of blocks to be selected randomly out of the N total blocks. Therefore, configuring $d = 1$ results in fully random victim selection, as a single block is randomly selected from the total blocks, providing effective wear leveling through randomness [270] (solving **FIC5**). Configuring $d \to \infty$ selects a larger subset of blocks to use greedy selection on, such that $d = N$ is equivalent to fully greedy victim selection, allowing to provide the lowest cleaning latency. Configuration of d thus allows to define the tradeoff between wear leveling and performance.

An evaluation by Yang and Zhu [270] of the algorithm shows the significance of the number of hotness classification targets that are utilized, and the configuration of the d parameter, where various hotness classification targets can significantly increase the WA during GC. Similarly, *File-aware Garbage Collection (FaGC)* [264] is a GC policy that maintains an *Update Frequency Table (UFT)* for each LBA of a file, in order to group valid pages in the victim block based on the access frequency when these are copied to a new block during GC. This is similar to grouping hot and cold data, but at the level of GC for each LBA in a file. SFS [1] implements a GC policy that accounts for data grouping, with a lower overhead of having to maintain an UFT for each file. It maintains a hotness classification for each block, and combines blocks with k-means clustering [79], into groups with similar hotness classification.

The GC cost of foreground cleaning in F2FS can take up to several seconds [272], as it is not a preemptive task. Implementing preemptive scheduling in kernel file systems is challenging, as during the preemption the kernel has to store the file system state to continue after higher priority tasks have finished, and the state being restored when returning depicts possibly outdated data. Due to the continued writing on the file system, restoring the prior state may no longer be valid, as ongoing writes may have changed file system metadata. OrcFS [272] implements *Quasi Preemptive Segment Cleaning (QPSC)*, which sets a maximum time interval T_{max} (default of 100ms). After cleaning a segment it checks if the timer has expired, and if so it checks if outstanding writes are present from the host. If there are outstanding writes, the locks are released and the write is executed, and if there are no outstanding writes the next segment can be cleaned and the timer is reset. This allows any host write command to not encounter a segment cleaning overhead higher than T_{max}.

A further drawback of segment cleaning with LFS, in particular F2FS, is that the modification of metadata during segment cleaning requires a checkpoint to be created after each segment clean. This constitutes to a significant overhead for the segment cleaning process. To avoid the excessive checkpointing after segment cleaning, *Segment Cleaning Journal (SCJ)* [74] adds support to F2FS to journal metadata updates made during segment cleaning, instead of creating a checkpoint. This journal is stored in a journal area, which delays the updating of the original metadata until the journal becomes large enough or the checkpointing time interval is reached. However, metadata still points to old invalid data blocks (referred to as *pre-invalid blocks*), which requires that data only be invalidated once the metadata is updated by the SCJ. Therefore, SCJ implements an adaptive checkpointing that evaluates the cost of checkpointing to flush the metadata updates and the accumulation of pre-invalid blocks, and checkpoints if its cost is lower.

In addition to utilizing GC policies to reduce its overheads, the policy can further be used to incorporate management of fragmentation. Park el at. [196] propose to use a *Valid Block Queue (VBQueue)* in which valid blocks are sorted during garbage collection. Typically, a victim segment in the LFS is selected for cleaning, valid data is copied to the free space at the log head, and the old segment is erased. The VBQueue is added such that after a victim segment is selected, it is copied into the VBQueue, where the blocks it contains are sorted by the inode number. Then are the valid blocks written to a new segment and the old one is erased. This sorting allows maintaining of file associated blocks together based on their inode number.

A different approach to mitigate GC overheads is to design the GC procedure such that accesses do not suffer from high tail latency when GC is running. TTFlash [265] achieves this through several mechanisms. It implements *plane-blocking GC*, which limits any resources that are blocking I/Os to only the affected planes on the flash. However, this leads to blocking of requests to the GC affected planes. Therefore, TTFlash implements *rotating GC* that only runs at most one GC operation on a *plane group*. The plane group assignment is based on the possible parallelism of the device, such as plane- and channel-level parallelism. This ensures that a plane group is never blocked for more than a single GC operation, implying that any request will not be blocked for more than one GC operation.

On embedded devices, and in particular mobile devices, in order to save energy the device suspends all threads when not needed (e.g., when the mobile screen is turned off). This implies that all file system threads are also suspended, which means the file system cannot run the background GC during these inactive sessions. Therefore, *Suspend Aware Cleaning (SAC)* [193] is an addition to LFS, which is the time between the suspend initiation and the suspending of the file system threads, also referred to as the *slack time*. It uses this slack time to run background GC, however it does not write any data on the flash storage, but instead selects a victim block and brings the still valid pages in the page cache and marks these as dirty. As a result, all pages in the victim block to be dirty, which allows it to be erased. This process is referred to as *virtual segment cleaning*, which however is not run every time the screen is turned off, but rather based on the device utilization, which is called *utilization based segment cleaning*.

7.4 Coordinating the Software Stack Layers

The various layers in the storage software stack introduces several redundant operations, such as GC that is run in the LFS and GC run on the storage device. This duplicate work leads to significant performance impacts [267], requiring a co-design of the storage device software and the file system. Qiu et al. [211] show that the co-design of FTL and the file system show benefits of reduced memory requirements for L2P mappings, and increased device parallelism that can be exploited with better file system knowledge of the storage

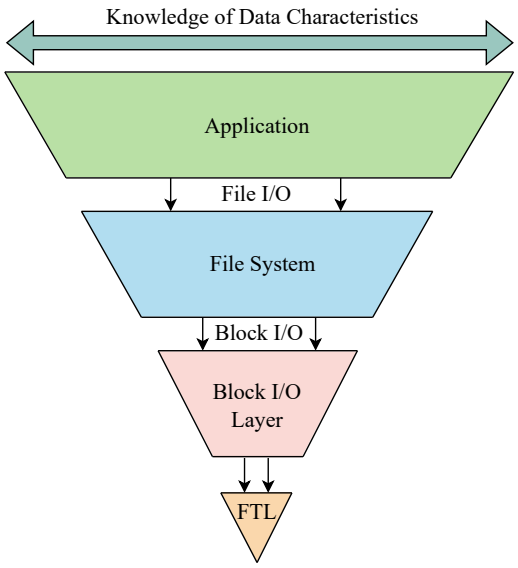

Figure 12: Layers in the storage software stack across which data specific knowledge decreases as I/O is passed downwards to the FTL, representing the semantic gap [275] with flash storage.

characteristics. In addition to duplicated work, information about data characteristics are lost across the various, since in order to integrate communication across the layers, each utilizes an interface for the other layers to communicate with. However, the expressiveness of the interfaces limits the capability to communicate data characteristics across the layers, as Figure 12 illustrates. Applications have the highest knowledge of data characteristics, how it is best allocated on the flash and managed in order to reduce GC, however cannot forward all of this information to the file system due to the lack of such APIs. Similarly, the file system may group specific data together, however cannot forward this information to the block layer, and similarly the FTL may take the submitted I/O requests and organize these different on the flash storage. This *semantic gap* between the storage device and storage software is a result of the device integrating into the existing block I/O interface [275], however failing to represent the flash-specific characteristics. Such mismatch between storage device and its accessing interface, requires storage software to enhance capabilities to pass information across the layers to avoid increasing the semantic gap across the layers.

DualFS [259] utilizes the custom integration with OCSSD to merge the garbage collection of the file system with that of the FTL, and present this scheme as *global garbage collection*. ParaFS [276] implements a similar coordination of file system GC and FTL GC. A different approach taken by Lee at al. [150] is to modify the block interface with the flash characteristics, moving responsibility directly to the file system, or other application built on top of it. The resulting interface called *Application Managed Flash (AMF)* exposes a block interface that does not allow overwriting unless an explicit erase is issued for the blocks. This matches the flash requirement that prior to overwriting data it has to be erased. The interface is implemented as a custom FTL, called AFTL, on top of which the ALFS file system is built. This avoids the duplicate garbage collection of the FTL and the file system, as the garbage collection of the ALFS erases blocks during garbage collection, informing the FTL to erase the physical block.

Co-designing the FTL and the file system allows removing uncoordinated duplicate work, and coordinate the flash management. To this end, Lee et al. [149] present a redesigned I/O architecture, called REDO, which avoids the duplicate operations from file system and FTL by implemented the new framework directly as the storage controller, and building the *Refactored File System (RFS)* on top of the new controller interface. By combining the file system operations with the storage controller, the file system is responsible for running GC and managing the storage by maintaining the L2P mappings.

7.5 Summary

With GC being a significant performance bottleneck for flash storage, adequate data grouping and effective GC policies allows to reduce and manage the GC. Furthermore, the complexity of storage system software lacks support for effective APIs for coordination on data placement between the storage stack layers. Such coordination, allows to reduce the semantic gap and eliminate redundant and duplicated operations of the various layers.

8 FIC-3: I/O Amplification

Several of the applied mechanisms for dealing with asymmetric flash performance and garbage collection are key mechanisms to eliminate I/O amplification. Table 7 shows the different mechanisms that can be applied to lower the various types of I/O amplification, including *Write Amplification (WA)*, *Read Amplification (RA)*, and *Space Amplification (SA)*. These include the benefit of write buffering, which in the case of write requests that are smaller than the flash allocation unit, avoids the unnecessary WA to fill the flash allocation unit. Similar buffering as is employed in WODS, such as the B^{ϵ}-trees, allows decreasing the WA. Another key mechanism is reducing the generated write traffic, which minimizes WA, RA, and SA through deduplication, compression, delta-encoding, and virtualization. Similarly, the discussed methods of grouping data, avoiding fragmentation, allows reducing the RA to locate data, and furthermore limits GC overheads and GC caused WA. Throughout this section we evaluate

Mechanism	Amplification Type	File Systems
Write Optimized Data Structures (§6.1)	WA	[57, 89, 103, 217, 245]
Write Buffering (§6.2)	WA	[105–107, 117, 167, 198–200]
Reducing Write Traffic (§6.3)	WA, RA, SA	[57, 86, 90, 99, 151, 158, 185, 257, 283]
Data Grouping (§7.2)	WA, RA	[143, 159, 214, 277, 278]
GC Policies (§7.3)	WA, RA	[1, 74, 193, 272]
Space Optimized Data Structure (§8.1)	SA	[143, 157, 166]
WA with Coarse Granularity Flash Mappings (§8.2)	WA	[126, 147, 272, 272]
Reverse Indexing (§8.3)	WA	[166]

Table 7: Mechanisms for file systems to deal with I/O amplification caused by flash storage integration, and the respective file systems that implement a particular mechanism. Green highlighted table cells depict previously discussed mechanisms with their respective section.

the additional methods for limiting the various types of I/O amplification.

8.1 Space Optimized Data Structures

Similar to the design of a WODS, data structures with particular focus on optimally utilizing available space are a mechanism to deal with SA. A commonly applied method for file systems to optimize space utilization is to possibly embed file data in the inode. Commonly the allocation of the file system inode occupies at least a block, such as 4KiB in F2FS, which however is more space than file metadata requires. Therefore, several bytes (∼3.4KB in F2FS) are free, which are used to inline file data in the inode. This particularly allows for small files to entirely fit into the inode, avoiding writing the inode and leave the unused space empty, and additionally write an additional data block, which also has free space. Different to inline data, ReconFS [166] uses an inode with a size of 128B, allowing to place numerous inodes in a single flash page. With such an inode size, writing each inode change directly requires filling the flash page with unnecessary data. Therefore, ReconFS implements a *metadata persistent log*, in which metadata changes are logged and compacted to align with pages, and are only written back to the storage when evicted or checkpointed, in order for the file system to remain consistent.

Similar to effective tree-based WODS, the radix tree is a space optimized tree variant, that is commonly used as the directory and inode tree for file systems [143, 157]. The directory tree is commonly constructed and maintained in memory and written to the persistent flash storage. Its space optimization revolves around merging of nodes that have a single child with that child node. This eliminates the need for an individual node that is assigned to each child, lowering the space requirement. As a result, the radix tree is also referred to as a compressed tree, due to the compression of single child nodes.

8.2 Coarse Granularity Flash Mappings

A similar goal of file systems is to reduce the amount of memory that is required for the mapping table to maintain the L2P mappings. L2P mappings are commonly persisted periodically, from the storage device RAM to the flash storage, such that in the case of system shutdown or the device is unplugged, upon reconnection the mapping information can be recovered. Hence, the mapping information similarly requires flash pages to be stored. A common solution is to increase the granularity of the mapping table (e.g., block-level mapping instead of page-level mapping), requiring fewer mappings. MNFS [126] manages flash storage with page-level and block-level mapping, depending on the update frequency. Metadata is updated more frequently and therefore utilizes a page-level mapping compared to larger mapping granularity for data. OrcFS [272] similarly utilizes a page-level for metadata, and a superblock-level mapping for data, which represents several flash blocks. The allocation unit is called a superblock as it consists of multiple blocks (not to be confused with the file system superblock). Furthermore, logical addresses are mapped to the same physical addresses in the data partition, requiring no mapping table, and file system sections are aligned to the superblock unit. Therefore, OrcFS only requires a block allocation information for each file in the superblock, which are stored in the inode block in the metadata area.

However, this comes at the cost of having a larger allocation unit, and if a host write is smaller than the allocation unit it causes WA, due to the partial flash page write when the flash page size and the allocation unit are not aligned [147, 272]. OrcFS [147, 272] implements *block patching* to solve this issue. It takes write requests that are smaller than the flash page size and pads the remaining space with dummy data to align the write request to flash page size. This mechanism avoids copying data if a flash page is partially written, and the next LBA in the same flash page is written, which triggers a copy of all LBAs in the flash page followed by writing the LBA for the new write. For instance, for a flash page containing 4 LBAs, if LBAs 1-3 are written by one request,

the first 3 LBAs are mapped to the data and the fourth holds dummy data, such that the page is fully filled. If a second request to LBA 4 is issued, it cannot fill the flash page as it has already been written. Therefore, to write the newly written data after the already written LBAs, it must copy LBAs 1-3, append the new write to LBA 4, and write the 4 LBAs to a new flash page. Adding of dummy data to fill pages reduces the WA, which would be caused by copying of all data in the flash page, as it now avoids copying the added dummy data on consecutive writes. While reduction in WA are presented, the adding of dummy data nonetheless adds WA to fill the flash page. However, as latter updates require less data written, and the importance of data grouping indicates, maintaining related data in the same flash page is more beneficial and possibly decreases future WA. Related data remains in the same flash page, as only the valid data in flash pages is copied on writes, as opposed to copying the entire flash page, introducing copied dummy data.

8.3 Reverse Indexing

As file system metadata is commonly maintained in a tree-based data structure, updates to metadata in the leaf nodes can propagate changes to the root node, known as the Wandering Tree Problem [14]. Due to the update of leaf metadata, such that when file data is modified, the metadata points to the new location of the file data, causing new metadata to be written, which in turn requires its parent to be updated to point to the new location of the metadata. This propagates up to the root note, causing significant WA. F2FS utilizes a table based indexing, with the NAT, such that only a table entry is required to change to update the data location, and metadata points to the table entry to locate the data. ReconFS [166] utilizes an inverted indexing tree, which also avoids the wandering tree problem. With such a tree, each node points to its parent node, instead of the parent node pointing to a child node. Therefore, upon address change of a child node, the parent does not need to be modified, since the child node points upwards to the parent node. Similar mechanisms are utilized in FTL design [169], where indexing data is written in the OOB space of the flash page from the data, in order to locate its metadata. In order to avoid increased scan times on failure recovery, which can no longer traverse the tree from the root, the updated pages are tracked to locate the most recently updated valid page, which is then periodically included in the checkpointing to ensure consistency.

8.4 Summary

The introduced I/O amplification of flash storage, particularly a result of GC, requires careful consideration to reduce the write requests, such that the flash device lifetime can be extended. Several of the previously discussed mechanisms, such as reducing write traffic and utilizing effective GC policies, aid in reducing the I/O amplification, however furthermore particular data structures optimized for space utilization similarly provide efficient methods for reducing I/O amplification.

Mechanism	File Systems
Aligning the Allocation Unit (§7.1)	[197, 245]
Clustered Allocation & Striping (§9.1)	[1, 4, 57, 171, 172, 198, 272, 276]
Concurrency (§9.2)	[114, 117, 142, 150, 157, 276]

Table 8: Mechanisms for file systems to exploit flash parallelism capabilities, and the respective file systems that implement a particular mechanism. Green highlighted table cells depict previously discussed mechanisms with their respective section.

9 FIC-4: Flash Parallelism

With the capabilities of flash storage relying largely on increased parallelism, several existing mechanisms are leveraging these. Depending on the level of flash integration, different mechanisms are possible, where at the SSD integration the host has not control over the possible physical parallel utilization of flash, as the FTL controls this, however deeper flash integration at the custom and embedded levels provide more possibility. Table 8 depicts the various mechanisms to aid the utilization of flash parallelism and exploit the physical characteristics of flash.

9.1 Clustered Allocation & Striping

As host software has no direct access to flash storage with SSD, the FTL implements and manages all device-level parallelism. The possibility for the host to utilize flash parallelism comes from aiding the FTL in providing large enough I/Os such that the FTL can stripe data across flash chips and channels. This is achieved with *clustered blocks/pages* [132], where blocks or pages on different units (such as blocks on different planes) are accessed in parallel. The FTL can possibly stripe data across these clustered blocks, given that the I/O request is large enough to fill the clustered unit. Such a mechanism aligns with prior discussed aligning of the allocation unit (Section 7.1) to a physical unit to reduce I/O amplification, such as making the file system segment unit a multiple of the flash allocation unit. SFS [1] takes advantage of the achieved device-level parallelism with clustered blocks by aligning segments to a multiple of the clustered block size. During garbage collection SFS ensures that cleaning of segments, that do not have enough blocks to fill the clustered block size, is delayed until enough data is present.

SSDFS [57] utilizes the custom flash integration to map data allocation of segments to the unit of a *Physical Erase Block (PEB)*, where the PEB is split over the parallel unit on the device, such as previously mentioned parallel erasing of flash blocks over channels. Therefore, utilizing PEBs over the parallel unit allows striping writes into a segment over the varying channels, increasing the device parallelism. In order to achieve this, I/O requests have to be large enough such that they can be striped across the channel and fill the PEBs mapped to the segment. For this, SSDFS utilizes aggressive merging of I/O requests to achieve the larger I/Os that can be striped across the parallel units (solving **FIC7**). Instead of merging I/O requests, OrcFS [272] increases its file system allocation unit to a *superblock* (not to be confused with the file system superblock), which represents a set of flash blocks. These flash blocks are then split over the parallel units of the flash storage by the file system for increased parallelism. The file system utilizes a custom flash integration, and hence is capable of managing the parallelism of the flash storage.

The large allocation unit however introduces increased GC overheads, and block-level striping has lower performance than page-level striping [276]. Therefore, ParaFS [276] implements a 2-D allocation scheme, with page-level striping over the flash channels, where striping is also based on the data hotness, hence having a 2-dimensional allocation scheme. Different groups are assigned for the hotness levels, where writes are issued to the corresponding hotness group striped over the flash channels. Several other file systems implement variations of striping across different parallel units on flash storage [4, 171, 172, 198]. These mechanisms are also present in the design of storage applications, such as key-value stores [251, 275].

9.2 Concurrency

Similar to increasing the data allocation unit, concurrency is a mechanism to exploit the flash parallelism. LFS design relies on a single append point at the head of the log, in the simplistic implementations, depending on methods such as locking to ensure only one write is issued at the log head. This has a significant impact on the performance where other I/Os are idle while a single I/O completes. F2FS however suffers from severe lock contention overheads, where the performance of the multi-headed logging is nearly fully deprecated due to the serialization of data updates [157]. In particular, as data has to be written persistently before inode and other metadata can be written. Furthermore, F2FS suffers from contention of the in-memory data structures, for which it uses reader-writer semaphores for read and write operations from the user (termed *external I/O operations*), and reader-writer locks for writing of checkpoints and other metadata (termed *internal operations*) [157]. As lock counters are shared among all cores, cache coherence adds a significant overhead that increases with more cores. Max [157] extends F2FS to increase the concurrency scalability with three main modifications.

Firstly, in order to eliminate cache coherence overheads, it introduces a *Reader Pass-through Semaphore (RPS)* that uses a per-process counter. Secondly, the shared data structures in memory are partitioned by the inode, such that concurrent accessing does not require locking on parts of the radix tree, but instead on an inode basis. Lastly, it utilizes multiple independent logs, called a *Minor Log (mlog)*, which are accessed concurrently. The different between mlog and multi-headed logging in F2FS is that atomic data blocks are mapped to the same mlog, eliminating the need to ensure concurrency control across different logs. Ordering for persistence, ensuring data blocks are written before metadata, is delegated to the recovery mechanism using a global versioning number in each inode to identify ordering across mlogs, and recover the most recent version number in case of a system crash. These mechanisms eliminate much of the needed concurrency control, which sequentialized major parts of operations and hindered multicore scalability.

With this increased concurrency capabilities, the file system can issue more I/O requests to the device, allowing to leverage a higher degree of on-device parallelism. Similarly, DevFS [117] utilizes the parallel capabilities by exploiting the high number of I/O queues supported by NVMe. It maps I/O queues to individual files, allowing single file operations to submit I/Os concurrently without interfering on the I/O queue, therefore increasing the per-file concurrency as well. Likewise to the concept of mlog, SpanFS [114] maps files and directories to different *domains*, such that individual domains can be accessed in parallel. Such methods have a higher lock granularity, where concurrency below the lock granularity is not possible, as a single process is holding the lock. Therefore, instead of holding locks for individual inodes or files, preventing concurrent writing to the same inode or file, Lee et al. [142] extend F2FS to utilize *Range Locking (RL)*, in which ranges of a file are locked, and different ranges can be written concurrently. Therefore, it provides the possibility for intra-file parallelism.

ALFS [150], exploits the flash parallelism by mapping consecutive file system segments to the flash channels and utilizing different I/O queues for each flash channel. Similarly, ParaFS [276] implements *parallelism-aware scheduling*, which also maintains different I/O queues for each channel. However, it extends this concept by using a *dispatching phase* and a *request scheduling phase*. The dispatching phase optimizes write I/Os by scheduling I/O requests to the respective channels based on the utilization of the channel, such that the least utilized channels receives more requests. All requests are assigned a weight, which indicates their priority in the queue, where read requests weight is lower than that of write requests, because of the asymmetric performance of flash storage. During the request scheduling phase the scheduler assigns slices to the read and write/erase operations in the individual queues, such that if the time from the slice of a read

operation is up and the queue contains no other read requests, a write or erase is scheduled, based on a fairness formula that incorporates the amount of free space in the block and concurrently active erase operations on other channels. This allows to minimize the erase operations on the flash, giving always free channels to utilize and maintain a fair schedule between write and erase operations.

9.3 Summary

Due to the architecture of flash storage providing a high degree of parallelism, numerous methods are employed to leverage these parallel units in order to maximize the performance. Depending on the level of flash integration, particular design choices can be made, such as clustered allocation and striping can be achieved with flash SSD integration, by providing large write I/O requests such that the FTL can stripe the data across parallel units. With a higher degree of control over the flash storage, file systems can directly rely on utilizing concurrency to leverage the parallelism of flash storage.

10 FIC-5: Wear Leveling

As flash cells wear out over time, it is important to utilize the flash evenly to avoid burning out particular flash cells faster than others. The possibility for ensuring even wear at the different levels of integration is limited, as at the SSD flash integration level, the FTL handles all wear leveling, without host considerations. However, similar to prior flash integration challenges, several mechanisms are nonetheless applicable. In particular, reducing the write traffic to the flash device, as less writing incurs less flash wear, and particularly flash-specific data structures inherently provide a degree of wear leveling. Based on the sequential write requirement, data structures, such as the LFS must write sequentially in an append-only fashion, which evenly writes the space. At closer to flash integrations, where host systems have more control over the flash management, there are particular mechanisms to ensure better flash wear. Table 9 depicts the methods we discuss in this section for enabling increased wear leveling for file systems.

Mechanism	File Systems
Write Optimised Data Structures (§6.1)	[57, 89, 103, 217, 245]
Reducing Write Traffic (§6.3)	[57, 86, 90, 99, 151, 158, 185, 257, 283]
GC Policies (§7.3)	[1, 74, 193, 272]
Write & Read Leveling (Sections 10.1 and 10.2)	[148, 150, 159, 259]

Table 9: Mechanisms for file systems to deal with wear leveling of the flash storage, and the respective file systems that implement a particular mechanism. Green highlighted table cells depict previously discussed mechanisms with their respective section.

10.1 Write Leveling

Several flash integration challenges proved data grouping to be an effective method for dealing with GC overheads and I/O amplification. However, this can have an effect on the flash storage. In particular the hot data, such as file system metadata which is more frequently updated and written, must be moved across the flash space more than cold data. CFFS [159] therefore switches the allocation areas for metadata and data blocks, such that an erased metadata block becomes a free data block. Therefore, cold data should be placed in blocks that have been written more frequently, whereas hot data should be placed in blocks with a lower write count history. The principle of migrating cold data from less written blocks, which are also referred to as *younger blocks*, to more frequently written, *older blocks*, is referred to as *cold-data migration*, and similarly moving hot data from old blocks to younger blocks is known as *hot-data migration* [26]. These methods are commonly used in FTL implementations due to their simplicity and effectiveness, and similarly in file systems such as ALFS [150].

Wear leveling is an increasingly vital concern on file systems that utilize flash dual mode [148, 259], where it switches the flash level to increase performance for critical I/O requests. Due to the lowering in flash cell level, the same amount of written data requires a larger amount of space, where a switch from MLC to SLC divides the capacity in half, requiring double the space for the same I/O request. Therefore, these file systems include a *write budget* that is maintained for the areas, and dynamically resizes the available lower cell level area, such as decreasing the space if the wear is reaching a threshold. This switches the cell level back to a higher number, allowing to write more with less wear. Additionally, the file systems utilize the wear budget in order to identify if an I/O request should be redirected to the larger cell area, instead of being written to the lower cell level area.

10.2 Read Leveling

While write operations are the major cause of flash cells burning out, read operations also pay a toll on flash cells, as the current flash cell technology utilizes flash cells that are only capable of holding very few electrons (determining the charge of the gate) due to their size [163, 231]. This makes the cells increasingly susceptible to *read disturbance* [161], where reading of a page results in shifting of voltages in nearby cells (typically in the same block), requiring frequent rewriting to ensure the charge stays consistent. In order to control read disturbance, Liu et al. [161] propose to read-leveling

mechanisms in the FTL. While their proposal is aimed at FTL implementations, the ideas are applicable to file systems for flash storage devices. With the proposed read-leveling, the read-hot data, that is read more frequently, is isolated from other data pages by placing the hot pages into *shadow blocks*, which contain no valid data, in order to avoid disturbing that data. However, this requires to identify the read-hot pages, where a tracking of read counters for each page would require significant resources. Therefore, a *second-chance monitoring strategy* is proposed, which initially tracks the reads for each block, therefore requiring counters at a higher block granularity, and upon reaching a threshold indicating the block contains read-hot pages, the individual pages in the block are tracked on their read counters. Finally, the pages in these blocks that reach a certain threshold are copied to the shadow blocks. Therefore, this avoids the tracking of read counters for individual pages and only copies read-hot pages into the shadow blocks. While this strategy requires copying of read-hot pages to shadow blocks, it minimizes read disturbance which in turn minimizes the WA it causes.

10.3 Summary

In addition to previously discussed mechanisms to reduce I/O amplification and GC, resulting in decreased wear of the flash storage, write leveling, ensuring that write requests are spread across the available storage space, and read leveling are important mechanisms for ensuring the longevity of the flash storage.

11 FIC-6: I/O Management

Given that flash storage has the capabilities to achieve single digit μ-second latency, whereas overheads in the software stack, such as context switching in the kernel caused by system calls, can already require μ-seconds to complete [233], making software the dominating factor in overheads [21, 22, 63, 225, 250]. In addition, *interrupts* cause significant overheads for systems. Aimed at slow storage devices, the I/O request is submitted to the device, the context is switched, such that the process can continue with other work, and upon completion of the I/O, the host is interrupted, and the context is switched again. Any added interrupt on the I/O path can cause significant delays [230]. Cache effects are another drawback of context switching, since other work is continued, replacing data in the caches, it requires bringing the replaced data back into the caches after the interrupt and resuming of the prior context. Similarly, it also causes *Translation Lookaside Buffer (TLB)* pollution on the host system. A different approach to submitting I/O requests is with *polling*, which eliminates the need for context switches. With polling, the I/O request is submitted and instead of continuing other work, the process regularly checks the I/O for completion. Using polling for I/O requests has been shown to be a favored method of building application for fast storage devices [52, 136, 268].

Mechanism	File Systems
I/O Operations (§11.1)	[245]
I/O Scheduler (§11.2)	[209]
I/O Path - User-Space File Systems (§11.3)	[117, 162, 245, 273]

Table 10: I/O scheduling mechanisms to exploit performance capabilities of flash storage, and the respective file systems that implement a particular mechanism.

Particular for fast storage devices, with the utilization of *synchronous I/O*, to saturate the storage device, additional threads are required, which each submit I/O requests to the numerous I/O queues in the of the storage device. However, this mechanism does not scale efficiently, where each thread must wait until the I/O request is completed, and thread scheduling overheads are introduced. Therefore, with asynchronous I/O the threads do not wait for completion, but instead submit a larger number of I/O requests each, allowing to fill the device I/O queues more effectively. The I/O requests for which a thread as submitted a request, but have not completed, are referred to as *outstanding* or *in-flight* I/O requests. Table 10 shows the mechanisms for host systems to better leverage the flash storage performance and minimize overheads. In addition to file systems implementing particular mechanisms, we discuss more general methods applicable to all applications for benefiting from flash storage and enhancing performance.

11.1 I/O Operations

Given that particular I/O patterns can have degrading affects on the SSD performance, such as mixing read and write operations, as they share resources on the device, including the mapping table and ECC engine, and furthermore possibly invalidating the cached data in the SSD RAM. Similarly, mixing I/O operations with different block sizes can result in increased fragmentation [245]. As the Linux kernel relies on a submission and completion queue for I/O, user-space frameworks such as SPDK and NVMeDirect provide more flexibility for user-space file systems to design different queues, depending on the requirements. URFS [245] utilizes this possibility to create adaptive queues that can better optimize I/O submissions to the device. Based on the workload characteristics URFS dynamically creates flexible I/O queues (e.g., group by size, read/write operation) to increase SSD performance. Similarly, Borge et al. [173] show with a case study on HDFS performance with SSD, that in order to leverage the capabilities of flash SSD, direct I/O, and increased parallel requests with buffered I/O are needed.

11.2 I/O Scheduler

With the possibility for asynchronous I/O to merge and reorder requests, the Linux kernel implements several schedulers, such as *NOOP*, *deadline*, and *CFQ* [83, 179, 208, 236]. NOOP being the least intrusive scheduler only merges I/O requests in its FIFO, but does not reorder them, which is beneficial on devices such as OCSSD, that require consecutive LBAs. Son et al. [234] showcase the benefits of merging random write requests, regardless of contiguity of the LBAs, in order to better enhance performance with fast storage devices. The deadline scheduler adds to NOOP by utilizing merging and reordering, however also applies a deadline for each I/O request to ensure requests are submitted to the device eventually. Two separate queues, one for read requests and an additional one for write requests are utilized, which are both ordered by the deadline of the request. Another scheduler variant of deadline exists, called *mq-deadline*, which is aimed at multi-queue devices, such as NVMe SSD. *Completely Fair Queuing (CFQ)* implements a round-robin based queue that assigns time slices to each in order to prevent starvation and provide fairness. While these are the common schedulers in the Linux kernel, to see details on all schedulers present in the Linux kernel consult [246]. Such scheduling configuration begs the question on which scheduler is best suited for file systems on flash storage. Several studies into performance of the schedulers exist [236, 274], showcasing that merging of read I/O requests in synchronous I/O provides beneficial performance gains, and similarly the merging of write I/Os in asynchronous I/O shows performance gains.

Qin et al. [209] argue that I/O ordering limits exploiting the parallelism of flash devices. Especially as the Linux block layer does not guarantee particular ordering, flags such as *Forced Unit Access (FUA)*, indicating that I/O completion is signaled upon arrival of data on the storage, and *PREFLUSH*, which before completing the request flushes the volatile caches, have to be set in order to ensure a specific ordering [209]. With file systems, the I/O of metadata and data has a particular ordering, such that metadata can only point to data that exists, needing to ensure that data is written prior to metadata. Removing of I/O ordering allows eliminating this need and better utilize the flash parallelism. Utilizing the OOB area on flash pages, the file system developed by Qin et al., called NBFS, maintains versioning in order to identify out of order updates. Furthermore, updates are done using atomic writes (discussed in Section 12). The issuing of FUA requests further implies that its I/Os cannot be merged in the scheduler [209], implying that if a smaller than flash page size FUA I/O request is issued, it is padded to the page size, causing WA. NBFS solves this with its atomic writes that imply that the FUA request does not immediately have to be written to the flash, but instead wait for all data blocks to arrive, which are then used to fill the pages, allowing to reduce the WA (solving **FIC3**).

11.3 I/O Path - User-Space File Systems

A mechanism that is gaining significant attention in the research community is the utilization of user-space file systems, bypassing the kernel layers and avoid its associated overheads. These file systems run only in the user-space, as opposed to commonly used file systems (e.g., F2FS) running in kernel space. In addition to the benefit of avoiding kernel overheads, user-space file systems are easier to develop, have increased reliability and security by avoiding kernel bugs and vulnerabilities, and provide increased portability [242]. A widely adopted framework for building user-space storage applications is *Filesystem in USErspace (FUSE)* [239]. It is implemented over a kernel module with which it exports a virtual file system from the kernel, where data and metadata are provided by a user-space process, hence allowing user-space applications to interact with it. Since FUSE is implemented with a kernel module, FUSE based file systems suffer significant performance penalties, requiring more CPU cycles than file systems in kernel space. Particularly contributing to overheads is the need to copy memory between user-space and kernel-space, caused by the way FUSE handles I/O requests [248, 249]. Furthermore, FUSE still suffers from context switching [212, 248, 249] overheads and *Inter-Process Communication (IPC)* between the FUSE kernel module and FUSE user-space daemon [280].

A similar framework for building user-space applications with direct storage access is NVMeDirect [125]. However, it also relies on a kernel driver to provide enhanced I/O policies. SPDK [271] is another framework for building user-space storage applications, however it provides the mechanisms to bypass the kernel and submit I/O directly to the device, by implementing a user-space driver for the storage device. Such a framework allows building high performance storage applications in user-space, which eliminate the overheads coming from the kernel I/O stack.

URFS [245] provides increased concurrency performance by implementing a multi-process shared cache in memory, in order to avoid the kernel overheads of copying data as is present in FUSE. It furthermore helps avoid contention on the storage device. Eliminating of data copy is also addressed in ZUFS (zero-copy user-mode file system) [78], which is a user-space file system for persistent memory, which completes I/O requests by requesting exact data locations instead of copying data into the own address space. A similar user-space file system that implements a shared cache for process is EvFS [273], which is SPDK-based. While this file system can also support multiple readers/writers in the page cache, it only supports these for a single user process. User-space frameworks often provide capabilities to either expose the storage device as a block device, which the user-space application then accesses, or build a custom block device module (e.g., with SPDK, which also has default driver modules such as NVMe). For NVMe devices that support NVMe controller memory buffer

management, the file system can manage parts of the device memory. DevFS [117] utilize such an integration to manage the device memory for file metadata and I/O queues.

Different from prior discussed development frameworks, *File System as Processes (FSP)* [162] provides a storage architecture design for user-space file systems. The emphasis of FSP is to scale with the arrival of faster storage, and similarly to other user-space frameworks, minimize the software stack. For this it bases development on running file systems as processes, providing safer metadata integrity, data sharing coordination, and consistency control, as the process running the file system is in control of everything, instead of trusting libraries. Furthermore, FSP relies on IPC for fast communication, which unlike FUSE has a low overhead since it does not require context switching. Inter-core message passing comes at a low overhead and cache-to-cache transfers on multi-core systems can complete in double-digit cycles [233]. DashFS [162] is built with FSP, providing a safe user-space file system with isolation of different user processes, and efficient IPC.

11.4 Summary.

With the complexity of the storage software stack and the high performance of flash storage, the storage software stack dominates the I/O latency [21, 22], requiring careful I/O management to enhance the performance. In addition to I/O scheduling, allowing to merge consecutive I/O requests and reduce the overall issued I/O requests, user-space file systems, bypassing the Linux storage stack, avoid the storage stack layer overheads.

12 Failure Consistent Operations

An important aspect of file systems is to ensure that in the case of power failure or system crashes, the file system state and its data remain in a consistent state, and can be recovered upon a subsequent mount. While we do not classify it as a challenge of flash storage, we discuss the methods for ensuring failure consistent operations for flash. Most commonly used mechanisms for ensuring this are journaling, CoW, and checkpointing. However, journaling suffers from having to write data twice, once for the metadata log and again for the data log, to ensure full consistency [206, 262], also referred to as the double-write problem [117]. Therefore, file systems commonly only provide metadata consistency by logging only the metadata. Furthermore, power failure is a concern with flash storage, as it has capacitors that can flush parts of the on-device memory buffers, unless more expensive capacitors are used that can flush the entirety of the memory buffers. In the case of power failure, partially written data or metadata updates can result in the file system being in an inconsistent state.

Mechanism	File Systems
Atomic Writes (§12.1)	[34, 127, 209]
Transactions (§12.2)	[187]

Table 11: Failure consistency mechanisms to for flash storage, and the respective file systems that implement a particular mechanism.

While checkpointing, CoW, and other consistency mechanisms aim to handle these failures and provide recovery after failure, other methods exist for providing interfaces that eliminate partial operations. Particularly, the design of flash storage, such as the available OOB space of pages, provides beneficial options for such implementations. Table 11 depicts the methods for ensuring failure consistency with flash storage, and the file systems implementing such methods.

12.1 Atomic Writes

A possible method for ensuring that operations are not completed partially is through atomic writes. This is important to file systems as an update of data requires its respective metadata to also be updated. Failure of one should result in the other not being completed. A variety of flash devices support atomic writes through mechanisms such as in FTL [189] and through user-programmable SSD [226], however it can also be implemented in the file system itself. F2FS supports multi-block atomic writing, which allows updating multiple file system blocks in a single *ioctl* command [34, 127, 209]. ReconFS [166] provides multi-page atomicity by using a flag in each page to indicate if it is valid. This is achieved by writing a 1 in the head of the last flash page of the metadata updated, and all other pages have a 0 in the head. Therefore, in the case of power failure, when the file system is reconstructed, any metadata pages in between two pages with a 1 in the head (depicting two ending page updates) are valid pages, and in the case when the log has no page with the flag set to 1, the update failed and all pages after the last 1 flag are invalid.

Similarly, Qin et al. [209] showcase the utilization of the OOB area on flash with OCSSD in order to store metadata for ensuring data integrity in the No-Barrier File System (NBFS), which is extending F2FS. The file system uses *Data Node Chain (DN-Chain)* to ensure consistency measurements. DN-Chain is a linked list of all the blocks in an atomic operation, which are stored through a pointer in the OOB space, thus representing a linked list of pointers in an atomic operation. The last block in the DN-Chain points to itself to indicate the end of the linked list and the atomic operation. Furthermore, each block contains a checkpoint version, allowing the recovery after a failure to traverse the DN-Chain and identify if an invalid checkpoint version number is present, which indicates a failed atomic operation. As the initial writing of journal or CoW is done in memory and later flushed to the flash storage,

mechanisms such as failure-atomic *msync()* [202], provide atomicity in such operations.

12.2 Transactions

An additional approach for ensuring consistency is to use transactions, enforcing an "all-or-nothing" mechanism that either writes all data or no data at all if there are any failures during the transaction. Transactional support can stem from transactional block devices, such as TxFlash (Transactional Flash) [207] LightTX with embedded transaction support in the flash storage FTL [164, 165], and similar block device implementations that expose transaction support [38, 87, 116]. Additions to file systems can similarly implement transactions for block devices without transaction support. exF2FS [187] is an extension on F2FS that provides it with support for transactions. In order to support transactions, exF2FS implements several features. Firstly, with transactions relying on either all updates being present or no updates at all, the GC policy is adapted such that the GC module cannot reclaim flash pages that contain data from a transaction that is in progress. If such a page is garbage collected without the transaction having finished, it can be recovered and thus violates the all-or-nothing mechanism.

For this, exF2FS implements *shadow garbage collection*, which prohibits the GC module from using pages involved in pending transactions. Secondly, in order to provide large scale transactions with multiple exF2FS maintains *transaction file groups*, which are kernel objects that identify the set of files involved in a transaction. Lastly, it implements *stealing*, which allows dirty pages of pending transactions to be committed to the flash storage, however do not allow it to be garbage collected until the transaction completed. This mechanism is referred to as *delayed invalidation and relocation record*, and the process of allowing dirty pages to be evicted in uncommited transactions is referred to as *stealing* [187].

12.3 Flash Failures

Flash failures are another important aspect of file systems on how they manage and recover from these failures. A study into file system reliability showed that 16% of injected faults resulted in an unmountable file system, which furthermore was not fixable by a file system checker [96]. Especially, as the density of flash is increasing, where more layers of NAND is stacked on top of each other, resulting in a decrease in the flash reliability [177]. While flash employs ECC to handle correcting of data, there are errors that are not fixable such as the *uncorrectable bit corruption* [10, 17, 18], which can be caused by flash wear, read disturbance, and other factors [96]. We do not go into detail of flash induced errors, for a detailed evaluation on flash errors recovery mechanisms consult [67, 96]. Jaffer et al. [96] provide several guidelines for file systems to enhance reliability with flash errors, including adding more sanity checks, especially of metadata, and finding better tradeoffs between checksums and the checksum granularity, where a large checksum granularity can result in significant data loss if it has an unrecoverable invalid checksum, and a small checksum adds overheads on the needed space and performance. While file system checkers aim to solve a degree of faults, they are no panacea to fixing corrupted file systems [67] and require better failure handling in the file system itself.

12.4 Summary

The importance of failure consistent operations for storage systems has resulted mechanisms, such as atomic writes, to utilize the available OOB area of flash pages for metadata to ensure consistency in the case of failures. Similarly, in order to avoid flash errors, checksum methods have been introduced to alleviate errors as a result of flash failures.

13 Discussion

With the multitude of methods for dealing with flash integration, there are several that are of key concern. In particular, GC and mechanisms of dealing with GC have shown to be applicable, and enhance other flash integration challenges. However, there is no panacea for solving all flash integration challenges. Incorporating a particular mechanism causes difficulty of integration with other flash challenges. Especially, depending on the level of integration of flash storage, there is limited possibility of integrating particular mechanisms. Therefore, the closer integration is largely necessary in order to more optimally integrate flash, however coming with the additional flash management complexity. Therefore, a tradeoff between the level of integratin and the required complexity must be made. A higher level of management allows the application to more optimally be designed for the flash. This however comes at the expense of increased complexity. Furthermore, generic file systems working on numerous integration levels, require more generic interface management, limiting their customizability for different integrations.

Clear trends in the storage community are becoming apparent, focusing on eliminating the hiding of flash management idiosyncrasies, and exposing its characteristics to the host. Therefore, allowing the host to integrate and optimize the software for its access patterns with increased knowledge of the underlying storage device characteristics [13, 15, 16, 23, 107, 113]. As stated in the CompSys Manifesto [92], "the grand challenge in storage systems is to combine heterogeneous storage layers, leveraging their programmability and capabilities to deliver a new class of cost, data, and performance efficiency for all kinds of applications". Reducing the semantic gap between storage hardware and software must be a driving concern in future storage system design, leveraging the existing hardware capabilities and enhancing integration for more efficient, effective, sustainable,

and reliable storage systems.

The recent addition of *Zoned Namespace (ZNS)* SSD [15, 243], standardized in the NVMe 2.0 specification [258], similarly provides host the control over GC management on the storage device, and matches the interface to the underlying flash characteristics. Leveraging such device interfaces allows the file system development to partially take control of on-device operations. In addition to eliminating duplicate work of file system and GC on the device, where the increased coordination provides better performance capabilities, the higher-level control of the file system allows it to apply its data knowledge, such as particular grouping based on file characteristics, without a significant increase in complexity through the zone interface of ZNS SSD. Such integration presents a well-defined interface aimed at eliminating the interface mismatch between flash storage and storage software, leaving a plethora of possibility for storage software to enhance flash integration. Exposing more storage device control to the host system, particularly in the configuration of ZNS, allows not only data placement to be better integrated into storage software, but furthermore allows to fully utilize the parallel units of the storage device. With ZNS devices, the parallelism unit of the storage device is clearly defined [8], and can be leveraged by the storage software.

We imagine interfaces exposing an increasing level of flash hardware characteristics to the host software, to begin appearing, in order to better coordinate and integrate the storage. Similarly, we envision future efforts to aim at further decreasing the semantic gap between storage hardware and software, and leverage a higher degree of coordination across storage software/hardware layers. The gain in popularity of user-space based applications, as we discussed in Section 11.3, presents prominent possibility for future development and limiting kernel involvement, which has become an increasing overhead in the software stack. Therefore, we furthermore picture an increase in user-space applications, and particularly file systems.

14 Related Work

With the growing adoption of flash based storage systems [46, 146], there has been a plethora of proposed systems to optimize for flash characteristics. We focus on the existing file systems for flash storage, but other aspects such as key-value stores are another popular use case for flash storage.

Flash Optimized File Systems. Egger [58] provide a survey on the file systems for flash storage at the time (2010), comparing the file systems in key features important for flash storage. This includes the feature set of the file system, time complexity of operations such as mount time, and space requirements for memory. However, evaluated file systems are not limited to flash specific file systems. While the survey presents an insightful summary of flash file systems, it was published in 2010, thus limiting the number of available file systems significantly. Similarly, Gal and Toledo [66] present a survey of algorithms and data structures for flash storage, which encompasses flash mappings and flash-specific file systems.

Jaffer [95] provide a comparison of five file systems for flash storage, analyzing the feature set of each and discussing limitations. In particular, the author focuses on the evolution of design trends for flash storage over the past decades, where the earliest file system included in the comparison was presented in 1994. The author additionally includes a discussion on file system optimizations for data management with Streams [113] and OCSSD [16, 205]. While the author presents an insightful analysis into design trends for flash storage, the literature review is limited to only the five discussed file systems and does not differentiate in the file system application domain.

Dubeyko [57] presents SSDFS, a file system designed for SSDs. Albeit not being a literature review of flash file systems, the author presents an extensive comparison of related work that proposes flash file systems. Including a discussion on flash-friendly and flash-oriented file systems, and summarizing the available methods for optimizing flash specific operations and storage management. While the discussion of flash file systems provides a comparison of flash file systems, it is similarly not differentiating between application domain of the file system. Munegowda et al. [183] showcase a study into several file systems on Windows and Linux for SSD and flash devices. Furthermore, the authors present a comparison of features for the varying file systems, including flash support and FTL integration, as well as a high level performance comparison. However, the study is focused on the main adopted file systems, lacking the inclusion of less adopted flash file systems.

Ramasamy and Karantharaj [213] survey the challenges of building file systems for flash storage, including the performance implications, caching techniques, and implications of the FTL. The authors additionally discuss available solutions to the presented design challenges for storage systems on flash memory. Similarly, Di Carlo et al. [51] present a study into the design issues and challenges of flash memory file systems. The authors analyze the inherent implications of the flash storage from the type of cell type used and required wear leveling, error correction, and bad block management. A comparison of several available flash file systems at the time of publication (2011) is presented, with focus on the discussed design challenges.

Flash Optimized Applications. Not focusing of file systems for flash storage, there have been several surveys into flash characteristics. Luo and Carey [5] present a survey into log-structured merge-tree (LSM-tree) design techniques for storage systems. With LSM-trees being a widely adopted and popular choice of database and key-value store design to optimize for flash storage, the presented survey showcases relevant flash storage optimizations for data management.

Similarly, Doekemeijer and Trivedi [55] present a study into key-value stores optimized for flash storage, showcasing several techniques that can likewise be applied to file system design for flash storage.

Flash Translation Layer. Chung et al. [37] provide a survey into the various FTL algorithms, discussing the design issues of various algorithms. Flash file system performance will largely depend on the FTL implementation for SSDs, making optimal FTL design an important aspect of optimizing file system performance. A similar survey on FTL algorithms is presented by Kwon et al. [138]. As embedded devices and possible custom integrations require management of flash at the file system level, the concepts for efficient and performant FTL algorithms are applicable to file system level management of flash storage.

15 Conclusion

The move from HDD to flash SSD, has been one of the most fundamental shifts in the storage hierarchy. The increased performance of flash SSD over HDD, capable of achieving single several GB/s bandwidth with millions of IOPS, however required adaptations in the software stack, changing the design of file systems and storage software to integrate with these devices. In this literature study we evaluate the current state of file systems for flash storage devices, how these file systems design to align with introduced flash characteristics, and how the integration of flash storage into file system design has affected storage stack design. We evaluate the findings of this literature study to each of the proposed *survey research questions (SRQs)*.

SRQ1: What are the main challenges arising from NAND flash characteristics and its integration into file system design?

The architecture of flash-based storage introduces six key challenges for file system and storage software developers to consider during software design and development. The first key challenge of flash storage is the (1) **asymmetric read and write performance**, for which file systems developers commonly resort to methods and data structures for enhancing write performance. Due to the architecture of flash storage, the lack of in-place updates introduces required (2) **garbage collection**, presenting a significant performance implication for flash storage. Furthermore, the implications of GC extend to introducing (3) **I/O amplification**, which additionally increases the wear of flash cells, requiring effective (4) **wear leveling** methods to employed. In order to leverage the performance capabilities of flash storage, the (5) **flash parallelism** must be exploited with particular methods that are capable of increasing the concurrency on the flash storage. The last key challenge of flash storage arises from the storage software stack into which the devices integrate, where the performance of the storage stack becomes the bottleneck. Therefore, (6) **I/O management** for flash storage is a vital aspect at limiting software overheads, and maximizing the utilization of the flash storage.

SRQ2: How has NAND flash storage influenced the design and development of file system and the storage software stack?

The main challenges of integrating with NAND flash storage resulted in file system design to utilize specific data structures, algorithms, and mechanisms. Log-based data structures are widely adopted for flash-based file system, due to the matching characteristics to the flash storage. In addition to log-based data structures, file system development has focused on several key methods to exploit the parallelism capabilities of flash storage, including clustered allocation, data striping, and increasing the I/O sizes. Similarly, the design of I/O management has propagated out of the file system design, into the I/O scheduler architecture, to optimize I/O activity for fast flash-based storage.

SRQ3: How will NAND flash storage and newly introduced NAND flash-based storage devices and interfaces affect future file system design and development?

Given the increasing rise in adoption for flash storage, and the introduction of new flash-based storage devices and interfaces, future implications of flash storage **(RQ3)** provide a promising ground of better integrating the flash storage with software, leveraging the increased performance capabilities further. Through new interfaces exposing a larger amount of flash characteristics, the host software gets an increasing level of possibility to design application specific flash management, integrating the application design with flash management. Future storage software developments are likely to continue integrating the closer integration of flash storage into host storage software design, optimizing the flash utilizing for full leveraging of flash performance. We envision the semantic gap between storage hardware and software to slowly decrease over time, allowing to build more performant, efficient, reliable, and responsible storage systems. Such efforts align with the grand challenges of future storage [92], particularly with increasing demand of systems due to the digitalization of the world, and the push towards a more sustainable future.

Acronyms

AHCI Advanced Host Controller Interface.

AMF Application Managed Flash.

API Application Programming Interface.

AR Address Remapping.

ARS Adaptive Reserved Space.

BIOS Basic I/O System.

CAFTL Content Aware FTL.

CAT Cost-Age-Time.

CFLRU Clean First Least Recently Used.

CFQ Completely Fair Queuing.

CMOS Complementary Metal-Oxide Semiconductor.

CoW Copy on Write.

CP Checkpoint.

CPU Central Processing Unit.

DAC Dynamic Data Clustering.

DMA Direct Memory Access.

DN-Chain Data Node Chain.

DRAM Dynamic RAM.

ECC Error Correction Codes.

F2FS Flash-Friendly File System.

FAB Flash Aware Buffer.

FaGC File-aware Garbage Collection.

FIC Flash Integration Challenge.

FIFO First in First out.

FPC File Access Pattern-Guided Compression.

FSP File System as Processes.

FTL Flash Translation Layer.

FUA Forced Unit Access.

FUSE Filesystem in USErspace.

GC Garbage Collection.

HDD Hard Disk Drive.

I/O Input/Output.

inode index node.

IOPS I/O Operations per Second.

IoT Internet of Things.

IPC Inter-Process Communication.

L2P Logical-to-Physical.

LBA Logical Block Address.

LFS Log-Structured File System.

LRU Least Recently Used.

LZ Lempel-Ziv.

MBF Multiple Bloom Filters.

MLC Multi-Level Cell.

mlog Minor Log.

MRU Most Recently Used.

NAT Node Address Table.

NVM Non-Volatile Memory.

NVMe Non-Volatile Memory Express.

OCSSD Open-Channel SSD.

OOB Out-Of-Band.

OPS Over-Provisioning Space.

P2L Physical-to-Logical.

PBA Page Boundary Alignment.

PBA Physical Block Address.

PCI Peripheral Component Interconnect.

PCIe PCI Express.

PEB Physical Erase Block.

PID Process Identifier.

PIO B-Tree Parallel I/O B-Tree.

PLC Penta-Level Cell.

QPSC Quasi Preemptive Segment Cleaning.

RA Read Amplification.

RAM Random Access Memory.

RFS Refactored File System.

RL Range Locking.

RLE Run Length Encoding.

RLSQ Related Literature Studies Query.

RM-IPU Remap-Based In-Place-Updates.

RPS Reader Pass-through Semaphore.

RSQ Relevant Studies Query.

SA Space Amplification.

SAC Suspend Aware Cleaning.

SATA Serial Advanced Technology Attachment.

SCJ Segment Cleaning Journal.

SCM Storage Class Memory.

SDF Software-Defined Flash.

SIT Segment Information Table.

SLC Single-Level Cell.

SLR Systematic Literature Review.

SMR Shingled Magnetic Recording.

SPDK Storage Performance Development Kit.

SRQ Survey Research Question.

SSA Segment Summary Area.

SSD Solid State Drive.

TLB Translation Lookaside Buffer.

UFT Update Frequency Table.

V2P Virtual-to-Physical.

VBA Virtual Block Address.

VBQueue Valid Block Queue.

WA Write Amplification.

WL Wear Leveling.

WODS Write Optimized Data Structure.

WP Write Pointer.

ZAC Zoned Block Device ATA Command Set.

ZBC Zoned Block Command.

ZNS Zoned Namespace.

ZSLBA Zone Start LBA.

References

[1] SFS: Random Write Considered Harmful in Solid State Drives. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*, San Jose, CA, 2012. USENIX Association.

[2] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *USENIX 2008 Annual Technical Conference*, ATC'08, page 57–70, USA, 2008. USENIX Association.

[3] Miklós Ajtai. The complexity of the pigeonhole principle. *Combinatorica*, 14(4):417–433, 1994.

[4] One Aleph. YAFFS: Yet another Flash file system. *http://www. yaffs. net*, 2001.

[5] Chen Luo an. LSM-based Storage Techniques: A Survey. *ArXiv preprint*, abs/1812.07527, 2018.

[6] Nick Antonopoulos and Lee Gillam. *Cloud computing*. Springer, 2010.

[7] Remzi H. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*, volume 42. 2017.

[8] Hanyeoreum Bae, Jiseon Kim, Miryeong Kwon, and Myoungsoo Jung. What you can't forget: exploiting parallelism for zoned namespaces. In Ali Anwar, Dimitris Skourtis, Sudarsun Kannan, and Xiaosong Ma, editors, *HotStorage '22: 14th ACM Workshop on Hot Topics in Storage and File Systems, Virtual Event, June 27 - 28, 2022*, pages 79–85. ACM, 2022.

[9] Maria Ijaz Baig, Liyana Shuib, and Elaheh Yadegaridehkordi. Big data adoption: State of the art and research challenges. *Information Processing & Management*, 56(6):102095, 2019.

[10] Hanmant P Belgal, Nick Righos, Ivan Kalastirsky, Jeff J Peterson, Robert Shiner, and Neal Mielke. A new reliability model for post-cycling charge retention of flash memories. In *2002 IEEE International Reliability Physics Symposium. Proceedings. 40th Annual (Cat. No. 02CH37320)*, pages 7–20. IEEE, 2002.

[11] Michael A Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C Kuszmaul, Donald E Porter, Jun Yuan, and Yang Zhan. An introduction to B-trees and write-optimization. *login; magazine*, 40(5), 2015.

[12] Roberto Bez, Emilio Camerlenghi, Alberto Modelli, and Angelo Visconti. Introduction to flash memory. *Proceedings of the IEEE*, 91(4):489–502, 2003.

[13] Janki Bhimani, Jingpei Yang, Zhengyu Yang, Ningfang Mi, NHV Krishna Giri, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. Enhancing ssds with multi-stream: What? why? how? In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, pages 1–2. IEEE, 2017.

[14] Artem B Bityutskiy. JFFS3 design issues, 2005.

[15] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 689–703. USENIX Association, 2021.

[16] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 359–374, Santa Clara, CA, 2017. USENIX Association.

[17] Simona Boboila and Peter Desnoyers. Write Endurance in Flash Drives: Measurements and Analysis. In *FAST*, pages 115–128, 2010.

[18] Adam Brand, Ken Wu, Sam Pan, and David Chin. Novel read disturb failure mechanism induced by FLASH cycling. In *31st Annual Proceedings Reliability Physics 1993*, pages 127–132. IEEE, 1993.

[19] Gerth Stølting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *SODA*, volume 3, pages 546–554. Citeseer, 2003.

[20] Mingming Cao, Suparna Bhattacharya, and Ted Ts'o. Ext4: The Next Generation of Ext2/3 Filesystem. In *LSF*, 2007.

[21] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 385–395, 2010.

[22] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing safe, user space access to fast, solid state disks. In Tim Harris and Michael L. Scott, editors, *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, pages 387–400. ACM, 2012.

[23] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing Safe, User Space Access to Fast, Solid State Disks. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, page 387–400, New York, NY, USA, 2012. Association for Computing Machinery.

[24] Chandranil Chakraborttii and Heiner Litz. Reducing write amplification in flash by death-time prediction of logical block addresses. In Bruno Wassermann, Michal Malka, Vijay Chidambaram, and Danny Raz, editors, *SYSTOR '21: The 14th ACM International Systems and Storage Conference, Haifa, Israel, June 14-16, 2021*, pages 11:1–11:12. ACM, 2021.

[25] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[26] Li-Pin Chang. On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 1126–1130, 2007.

[27] Li-Pin Chang. Hybrid solid-state disks: Combining heterogeneous NAND flash in large SSDs. In *2008 Asia and South Pacific Design Automation Conference*, pages 428–433. IEEE, 2008.

[28] Li-Pin Chang and Tei-Wei Kuo. An adaptive striping architecture for flash memory storage systems of embedded systems. In *Proceedings. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 187–196. IEEE, 2002.

[29] Feng Chen, Song Jiang, and Xiaodong Zhang. Smartsaver: turning flash drive into a disk energy saver for mobile computers. In Wolfgang Nebel, Mircea R. Stan, Anand Raghunathan, Jörg Henkel, and Diana Marculescu, editors, *Proceedings of the 2006 International Symposium on Low Power Electronics and Design, 2006, Tegernsee, Bavaria, Germany, October 4-6, 2006*, pages 412–417. ACM, 2006.

[30] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. pages 181–192, 2009.

[31] Feng Chen, Rubao Lee, and Xiaodong Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing.

In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 266–277. IEEE, 2011.

[32] Feng Chen, Tian Luo, and Xiaodong Zhang. *CAFTL: A Content – Aware* flash translation layer enhancing the lifespan of flash memory based solid state drives. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, 2011.

[33] Yixin Chen and Li Tu. Density-based clustering for real-time stream data. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 133–142, 2007.

[34] Seungyong Cheon and Youjip Won. Exploiting multi-block atomic write in SQLite transaction. In *Proceedings of the International Conference on High Performance Compilation, Computing and Communications*, pages 23–27, 2017.

[35] M-L Chiang and R-C Chang. Cleaning policies in mobile computers using flash memory. *Journal of Systems and Software*, 48(3):213–231, 1999.

[36] Hyun Jin Choi, Seung-Ho Lim, and Kyu Ho Park. JFTL: A flash translation layer based on a journal remapping for flash memory. *ACM Transactions on Storage (TOS)*, 4(4):1–22, 2009.

[37] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. A survey of flash translation layer. *Journal of Systems Architecture*, 55(5-6):332–343, 2009.

[38] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From ARIES to MARS: transaction support for next-generation, solid-state drives. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 197–212. ACM, 2013.

[39] Douglas Comer. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[40] INCITS T10 Technical Committee. Information technology - Zoned Block Commands (ZBC). Standard, American National Standards Institute, 2014. Available from: https://www.t10.org/.

[41] INCITS T13 Technical Committee. Information technology – Zoned Device ATA Command Set (ZAC). Standard, American National Standards Institute, 2015. Available from: https://www.t13.org/.

[42] Kernel Development Community. Cramfs - cram a filesystem onto a small ROM. https://www.kernel.org/doc/html/latest/filesystems/cramfs.html. Accessed: 2022-07-10.

[43] Christian Monzio Compagnoni, Akira Goda, Alessandro S Spinelli, Peter Feeley, Andrea L Lacaita, and Angelo Visconti. Reviewing the evolution of the NAND flash technology. *Proceedings of the IEEE*, 105(9):1609–1633, 2017.

[44] Alex Conway, Eric Knorr, Yizheng Jiao, Michael A. Bender, William Jannen, Rob Johnson, Donald Porter, and Martin Farach-Colton. Filesystem Aging: It's more Usage than Fullness. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, 2019. USENIX Association.

[45] Michael Cornwell. Anatomy of a solid-state drive. *Commun. ACM*, 55(12):59–63, 2012.

[46] Tugrul U Daim, Pattravadee Ploykitikoon, Elizabeth Kennedy, and Woraruthai Choothian. Forecasting the future of data storage: case of hard disk drive and flash memory. *Foresight*, 2008.

[47] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013.

[48] Yuhui Deng. What is the future of disk drives, death or rebirth? *ACM Comput. Surv.*, 43(3):23:1–23:27, 2011.

[49] Peter Desnoyers. Analytic models of SSD write performance. *ACM Trans. Storage*, 10(2):8:1–8:25, 2014.

[50] Peter Deutsch. GZIP file format specification version 4.3. Technical report, 1996.

[51] Stefano Di Carlo, M Cramia, Paolo Prinetto, and Michele Fabiano. Chapter Design Issues and Challenges of File Systems for Flash Memories. 2011.

[52] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. Understanding modern storage APIs: a systematic study of libaio, SPDK, and io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage*, pages 120–127, 2022.

[53] Xiaoning Ding, Song Jiang, Feng Chen, Kei Davis, and Xiaodong Zhang. DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch. In *USENIX Annual Technical Conference*, volume 7, pages 261–274, 2007.

[54] Krijn Doekemeijer, Nick Tehrany, Balakrishnan Chandrasekaran, Matias Bjørling, and Animesh Trivedi. Performance characterization of nvme flash devices with zoned namespaces (zns). In *(to appear) IEEE International Conference on Cluster Computing, CLUSTER*

2023, October 31-November 3, 2023, Santa Fe, New Mexico, USA. IEEE, 2023.

[55] Krijn Doekemeijer and Animesh Trivedi. Key-Value Stores on Flash Storage Devices: A Survey. *ArXiv preprint*, abs/2205.07975, 2022.

[56] Fred Douglis and Arun Iyengar. Application-specific Delta-encoding via Resemblance Detection. In *USENIX annual technical conference, general track*, pages 113–126. San Antonio, TX, USA, 2003.

[57] Viacheslav Dubeyko. SSDFS: Towards LFS Flash-Friendly File System without GC operation. *ArXiv preprint*, abs/1907.11825, 2019.

[58] Christian Egger. File systems for flash devices, 2010.

[59] Jörn Engel and Robert Mertens. LogFS-finally a scalable flash file system. In *12th International Linux System Technology Conference*, 2005.

[60] K. Eshghi and R. Micheloni. *SSD Architecture and PCI Express Interface*, pages 19–45. Springer Netherlands, Dordrecht, 2013.

[61] Timothy R. Feldman and Garth A. Gibson. Shingled Magnetic Recording: Areal Density Increase Requires New Data Management. *login Usenix Mag.*, 38, 2013.

[62] Werner Fischer and Georg Schönberger. Linux Storage Stack Diagram. *Online] http://www. thomas-krenn. com/en/wiki/Linux_Storage_Stack_Diagram (License: CC-BY-SA 3.0, modified by Ingu Kang)*, 2017.

[63] Annie P Foong, Bryan Veal, and Frank T Hady. Towards SSD-Ready Enterprise Platforms. In *ADMS@ VLDB*, pages 15–21, 2010.

[64] Agostino Forestiero, Clara Pizzuti, and Giandomenico Spezzano. A single pass algorithm for clustering evolving data streams based on swarm intelligence. *Data Mining and Knowledge Discovery*, 26(1):1–26, 2013.

[65] Yinjin Fu, Hong Jiang, Nong Xiao, Lei Tian, and Fang Liu. Aa-dedupe: An application-aware source deduplication approach for cloud backup services in the personal computing environment. In *2011 IEEE International Conference on Cluster Computing*, pages 112–120. IEEE, 2011.

[66] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys (CSUR)*, 37(2):138–163, 2005.

[67] Om Rameshwar Gatla, Mai Zheng, Muhammad Hameed, Viacheslav Dubeyko, Adam Manzanares, Filip Blagojevic, Cyril Guyot, and Robert Mateescu. Towards robust file system checkers. *ACM Transactions on Storage (TOS)*, 14(4):1–25, 2018.

[68] Garth Gibson and Greg Ganger. Principles of Operation for Shingled Disk Devices. In *3rd Workshop on Hot Topics in Storage and File Systems (HotStorage 11)*, Portland, OR, 2011. USENIX Association.

[69] Emmanuel Goossaert. Coding for ssds–part 2: Architecture of an ssd and benchmarking, feb 2014.

[70] Nitesh Goyal and Rabi Mahapatra. Energy characterization of cramfs for embedded systems. In *International Workshop on Software Support for Portable Storage (IWSSPS) held in conjunction with the IEEE Real-Time and Embedded Systems and Applications Symposium (RTAS 2005)*, 2005.

[71] Goetz Graefe et al. Modern B-tree techniques. *Foundations and Trends® in Databases*, 3(4):203–402, 2011.

[72] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. pages 229–240, 2009.

[73] Sudhanva Gurumurthi, Anand Sivasubramaniam, Mahmut Kandemir, and Hubertus Franke. DRPM: dynamic speed control for power management in server class disks. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, pages 169–179. IEEE, 2003.

[74] Hyunho Gwak and Dongkun Shin. SCJ: Segment Cleaning Journaling for Log-Structured File Systems. *IEEE Access*, 9:142437–142448, 2021.

[75] Jin-Yong Ha, Young-Sik Lee, and Jin-Soo Kim. Deduplication with block-level content-aware chunking for solid state drives (SSDs). In *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 1982–1989. IEEE, 2013.

[76] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. Improving file system performance of mobile storage systems using a decoupled defragmenter. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 759–771, 2017.

[77] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. ZNS+: advanced zoned namespace interface for supporting in-storage zone compaction. In Angela Demke Brown and Jay R. Lorch, editors, *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 147–162. USENIX Association, 2021.

[78] Boaz Harrosh. zuf: ZUFS Zero-copy User-mode FileSystem. https://lwn.net/Articles/795996/. Accessed: 2022-07-10.

[79] John A Hartigan and Manchek A Wong. Algorithm AS 136: A k-means clustering algorithm. *Journal of the royal statistical society. series c (applied statistics)*, 28(1):100–108, 1979.

[80] Ibrahim Abaker Targio Hashem, Ibrar Yaqoob, Nor Badrul Anuar, Salimah Mokhtar, Abdullah Gani, and Samee Ullah Khan. The rise of "big data" on cloud computing: Review and open research issues. *Information Systems*, 47:98–115, 2015.

[81] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The unwritten contract of solid state drives. In Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic, editors, *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 127–144. ACM, 2017.

[82] Tim Hegeman and Alexandru Iosup. Survey of Graph Analysis Applications. *ArXiv preprint*, abs/1807.00382, 2018.

[83] Dominique A Heger and Richard Quinn. Linux 2.6 IO Performance Analysis, Quantification, and Optimization. In *Int. CMG Conference*, 2010.

[84] Jen-Wei Hsieh, Li-Pin Chang, and Tei-Wei Kuo. Efficient on-line identification of hot data for flash-memory management. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 838–842, 2005.

[85] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman A. Pletka. Write amplification analysis in flash-based solid state drives. In Miriam Allalouf, Michael Factor, and Dror G. Feitelson, editors, *Proceedings of of SYSTOR 2009: The Israeli Experimental Systems Conference 2009, Haifa, Israel, May 4-6, 2009*, ACM International Conference Proceeding Series, page 10. ACM, 2009.

[86] Ping Huang, Guangping Wan, Ke Zhou, Miaoqing Huang, Chunhua Li, and Hua Wang. Improve effective capacity and lifetime of solid state drives. In *2013 IEEE Eighth International Conference on Networking, Architecture and Storage*, pages 50–59. IEEE, 2013.

[87] Ping Huang, Ke Zhou, Hua Wang, and Chun Hua Li. BVSSD: Build built-in versioning flash-based solid state drives. In *Proceedings of the 5th Annual International Systems and Storage Conference*, pages 1–12, 2012.

[88] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[89] Adrian Hunter. A brief introduction to the design of UBIFS, 2008.

[90] Seunghwan Hyun, Hyokyung Bahn, and Kern Koh. LeCramFS: an efficient compressed file system for flash-based portable consumer devices. *IEEE Transactions on Consumer Electronics*, 53(2):481–488, 2007.

[91] Intel. Intel® SSD D7-P5600 Series. https://ark.intel.com/content/www/us/en/ark/products/202708/intel-ssd-d7p5600-series-6-4tb-2-5in-pcie-4-0-x4-3d3-tlc.html, Accessed: 2022-05-02.

[92] Alexandru Iosup, Fernando Kuipers, Ana Lucia Varbanescu, Paola Grosso, Animesh Trivedi, Jan Rellermeyer, Lin Wang, Alexandru Uta, and Francesco Regazzoni. Future computer systems and networking research in the netherlands: A manifesto, 2022.

[93] Charlie Isaksson, Margaret H Dunham, and Michael Hahsler. SOStream: Self organizing density-based clustering over data stream. In *International Workshop on Machine Learning and Data Mining in Pattern Recognition*, pages 264–278. Springer, 2012.

[94] Jeremy Iverson, Chandrika Kamath, and George Karypis. Fast and effective lossy compression algorithms for scientific datasets. In *European Conference on Parallel Processing*, pages 843–856. Springer, 2012.

[95] Shehbaz Jaffer. Evolution of File System design for Solid State Drives.

[96] Shehbaz Jaffer, Stathis Maneas, Andy A. Hwang, and Bianca Schroeder. The reliability of modern file systems in the face of SSD errors. *ACM Trans. Storage*, 16(1):2:1–2:28, 2020.

[97] Ashish Jagmohan, Michele Franceschini, and Luis Lastras. Write amplification reduction in NAND flash through multi-write coding. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–6. IEEE, 2010.

[98] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, et al. BetrFS: A Right-Optimized Write-Optimized File System. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 301–315, 2015.

[99] Cheng Ji, Li-Pin Chang, Riwei Pan, Chao Wu, Congming Gao, Liang Shi, Tei-Wei Kuo, and Chun Jason

Xue. Pattern-Guided File Compression with User-Experience Enhancement for Log-Structured File System on Mobile Devices. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 127–140. USENIX Association, 2021.

[100] Cheng Ji, Li-Pin Chang, Liang Shi, Chao Wu, Qiao Li, and Chun Jason Xue. An Empirical Study of File-System Fragmentation in Mobile Storage Systems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Denver, CO, 2016. USENIX Association.

[101] Chen Jia, ChengYu Tan, and Ai Yong. A grid and density-based clustering algorithm for processing data stream. In *2008 Second International Conference on Genetic and Evolutionary Computing*, pages 517–521. IEEE, 2008.

[102] Song Jiang, Xiaoning Ding, Feng Chen, Enhua Tan, and Xiaodong Zhang. DULO: an effective buffer cache management scheme to exploit both temporal and spatial locality. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, volume 4, pages 8–8, 2005.

[103] Yizheng Jiao, Simon Bertron, Sagar Patel, Luke Zeller, Rory Bennett, Nirjhar Mukherjee, Michael A. Bender, Michael Condict, Alex Conway, Martin Farach-Colton, Xiongzi Ge, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. Betrfs: a compleat file system for commodity ssds. In Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis, editors, *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, pages 610–627. ACM, 2022.

[104] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. Improving SSD Lifetime with Byte-Addressable Metadata. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '17, page 374–384, New York, NY, USA, 2017. Association for Computing Machinery.

[105] Heeseung Jo, Jeong-Uk Kang, Seon-Yeong Park, Jin-Soo Kim, and Joonwon Lee. FAB: Flash-aware buffer management policy for portable media players. *IEEE Transactions on Consumer Electronics*, 52(2):485–493, 2006.

[106] William K Josephson. *A Direct-Access File System for a New Generation of Flash Memory*. PhD thesis, Princeton University, 2011.

[107] William K. Josephson, Lars Ailo Bongo, Kai Li, and David Flynn. DFS: A file system for virtualized flash storage. *ACM Trans. Storage*, 6(3):14:1–14:25, 2010.

[108] Dawoon Jung, Jaegeuk Kim, Jinsoo Kim, and Joonwon Lee. Scaleffs: A scalable log-structured flash file system for mobile multimedia systems. *ACM Trans. Multim. Comput. Commun. Appl.*, 5(1):9:1–9:18, 2008.

[109] Jaemin Jung, Youjip Won, Eunki Kim, Hyungjong Shin, and Byeonggil Jeon. FRASH: Exploiting Storage Class Memory in Hybrid File System for Hierarchical Storage. *ACM Trans. Storage*, 6(1), 2010.

[110] Sanghyuk Jung, Yangsup Lee, and Yong Ho Song. A process-aware hot/cold identification scheme for flash memory storage systems. *IEEE Transactions on Consumer Electronics*, 56(2):339–347, 2010.

[111] Saurabh Kadekodi, Vaishnavh Nagarajan, and Gregory R. Ganger. Geriatrix: Aging what you see and what you don't see. A file system aging approach for modern storage systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 691–704, Boston, MA, 2018. USENIX Association.

[112] Dong Hyun Kang, Gihwan Oh, Dongki Kim, In Hwan Doh, Changwoo Min, Sang-Won Lee, and Young Ik Eom. When address remapping techniques meet consistency guarantee mechanisms. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[113] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The Multi-streamed $Solid-State$ Drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, 2014.

[114] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. SpanFS: A Scalable File System on Fast Storage Devices. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 249–261, Santa Clara, CA, 2015. USENIX Association.

[115] Mincheol Kang, Wonyoung Lee, Jinkwon Kim, and Soontae Kim. PR-SSD: Maximizing Partial Read Potential By Exploiting Compression and Channel-Level Parallelism. *IEEE Transactions on Computers*, 2022.

[116] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. X-FTL: transactional FTL for SQLite databases. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 97–108. ACM, 2013.

[117] Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. Designing a True Direct-Access

File System with DevFS. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 241–256, Oakland, CA, 2018. USENIX Association.

[118] S Kapoor and A Chopra. A review of Lempel Ziv compression techniques. *IJCST*, 4(2), 2013.

[119] Vamsee Kasavajhala. Solid state drive vs. hard disk drive price and performance study. *Proc. Dell Tech. White Paper*, pages 8–9, 2011.

[120] P Kavitha. A survey on lossless and lossy data compression methods. *International Journal of Computer Science & Engineering Technology*, 7(03):110–114, 2016.

[121] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *USENIX*, pages 155–164, 1995.

[122] Ram Kesavan, Matthew Curtis-Maury, Vinay Devadas, and Kesari Mishra. Storage Gardening: Using a Virtualization Layer for Efficient Defragmentation in the WAFL File System. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 65–78, Boston, MA, 2019. USENIX Association.

[123] Ram Kesavan, Matthew Curtis-Maury, Vinay Devadas, and Kesari Mishra. Countering fragmentation in an enterprise storage system. *ACM Trans. Storage*, 15(4):25:1–25:35, 2020.

[124] Nawsher Khan, Ibrar Yaqoob, Ibrahim Abaker Targio Hashem, Zakira Inayat, Waleed Kamaleldin Mahmoud Ali, Muhammad Alam, Muhammad Shiraz, and Abdullah Bin Gani. Big Data: Survey, Technologies, Opportunities, and Challenges. *The Scientific World Journal*, 2014, 2014.

[125] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. NVMeDirect: A user-space I/O framework for application-specific optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Denver, CO, 2016. USENIX Association.

[126] Hyojun Kim and Youjip Won. MNFS: mobile multimedia file system for NAND flash based storage device. In *CCNC 2006. 2006 3rd IEEE Consumer Communications and Networking Conference, 2006.*, volume 1, pages 208–212. IEEE, 2006.

[127] Jaegeuk Kim. [PATCH 2/4] f2fs: support atomic_write feature for database. https://lkml.org/lkml/2014/9/26/19, 9 2014.

[128] Jaegeuk Kim. DEFRAG.F2FS. https://manpages.debian.org/testing/f2fs-tools/defrag.f2fs.8.en.html, 2021.

[129] Jaegeuk Kim, Heeseung Jo, Hyotaek Shim, Jin-Soo Kim, and Seungryoul Maeng. Efficient Metadata Management for Flash File Systems. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 535–540, 2008.

[130] Jaeho Kim, Donghee Lee, and Sam H. Noh. Towards SLO Complying SSDs Through OPS Isolation. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 183–189, Santa Clara, CA, 2015. USENIX Association.

[131] Jaehong Kim, Sangwon Seo, Dawoon Jung, Jin-Soo Kim, and Jaehyuk Huh. Parameter-aware I/O management for solid state disks (SSDs). *IEEE Transactions on Computers*, 61(5):636–649, 2011.

[132] Jaehong Kim, Sangwon Seo, Dawoon Jung, Jin-Soo Kim, and Jaehyuk Huh. Parameter-Aware I/O Management for Solid State Disks (SSDs). *IEEE Transactions on Computers*, 61(5):636–649, 2012.

[133] Jonghwa Kim, Choonghyun Lee, Sangyup Lee, Ikjoon Son, Jongmoo Choi, Sungroh Yoon, Hu-ung Lee, Sooyong Kang, Youjip Won, and Jaehyuk Cha. Deduplication in SSDs: Model and quantitative analysis. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12. IEEE, 2012.

[134] Barbara Ann Kitchenham and Stuart Charters. Guidelines for performing Systematic Literature Reviews in Software Engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 2007.

[135] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The linux implementation of a log-structured file system. *ACM SIGOPS Oper. Syst. Rev.*, 40(3):102–107, 2006.

[136] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. Reaping the performance of fast NVM storage with uDepot. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 1–15, 2019.

[137] Tei-Wei Kuo, Jen-Wei Hsieh, Li-Pin Chang, and Yuan-Hao Chang. Configurability of performance and overheads in flash management. In *Asia and South Pacific Conference on Design Automation, 2006.*, pages 8–pp. IEEE, 2006.

[138] Se Jin Kwon, Arun Ranjitkar, Young-Bae Ko, and Tae-Sun Chung. FTL algorithms for NAND-type flash memories. *Design Automation for Embedded Systems*, 15(3):191–224, 2011.

[139] Stefan K. Lai. Brief History of ETOX NOR Flash Memory. *Journal of Nanoscience and Nanotechnology*, 12(10):7597–7603, 2012.

[140] Butler Lampson. Principles for computer system design. In *ACM Turing award lectures*, page 1992. 1993.

[141] Dave Landsman and D Walker. AHCI and NVMe as interfaces for SATA Express™ Devices, 2013.

[142] Chang-Gyu Lee, Hyunki Byun, Sunghyun Noh, Hyeongu Kang, and Youngjae Kim. Write optimization of log-structured flash file system for parallel I/O on manycore servers. In Moshik Hershcovitch, Ashvin Goel, and Adam Morrison, editors, *Proceedings of the 12th ACM International Conference on Systems and Storage, SYSTOR 2019, Haifa, Israel, June 3-5, 2019*, pages 21–32. ACM, 2019.

[143] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, Santa Clara, CA, 2015. USENIX Association.

[144] Chul Lee, Sung Hoon Baek, and Kyu Ho Park. A Hybrid Flash File System Based on NOR and NAND Flash Memories for Embedded Devices. *IEEE Transactions on Computers*, 57(7):1002–1008, 2008.

[145] Jongsung Lee and Jin-Soo Kim. An Empirical Study of Hot/Cold Data Separation Policies in Solid State Drives (SSDs). In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, New York, NY, USA, 2013. Association for Computing Machinery.

[146] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A case for flash memory ssd in enterprise database applications. In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 1075–1086. ACM, 2008.

[147] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(3):18–es, 2007.

[148] Sungjin Lee, Keonsoo Ha, Kangwon Zhang, Jihong Kim, and Junghwan Kim. FlexFS: A Flexible Flash File System for MLC NAND Flash Memory. In *2009 USENIX Annual Technical Conference (USENIX ATC 09)*, San Diego, CA, 2009. USENIX Association.

[149] Sungjin Lee, Jihong Kim, and Arvind Mithal. Refactored design of i/o architecture for flash storage. *IEEE Computer Architecture Letters*, 14(1):70–74, 2014.

[150] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind. Application-Managed flash. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 339–353, Santa Clara, CA, 2016. USENIX Association.

[151] Yongmyung Lee, Jong-Hyeok Park, Jonggyu Park, Hyunho Gwak, Dongkun Shin, Young Ik Eom, and Sang-Won Lee. When F2FS Meets Address Remapping. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '22, page 31–36, New York, NY, USA, 2022. Association for Computing Machinery.

[152] Hongyan Li. Flash Saver: Save the Flash-Based Solid State Drives through Deduplication and Delta-encoding. In *2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 436–441. IEEE, 2012.

[153] Jiangpeng Li, Kai Zhao, Xuebin Zhang, Jun Ma, Ming Zhao, and Tong Zhang. How Much Can Data Compressibility Help to Improve NAND Flash Memory Lifetime? In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 227–240, Santa Clara, CA, 2015. USENIX Association.

[154] Nanqinqin Li, Mingzhe Hao, Huaicheng Li, Xing Lin, Tim Emami, and Haryadi S. Gunawi. Fantastic ssd internals and how to learn and use them. In *Proceedings of the 15th ACM International Conference on Systems and Storage*, SYSTOR '22, page 72–84, New York, NY, USA, 2022. Association for Computing Machinery.

[155] Shuai Li, Wei Tong, Jingning Liu, Bing Wu, and Yazhi Feng. Accelerating garbage collection for 3D MLC flash memory with SLC blocks. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2019.

[156] Zheng Li, Shuangwu Zhang, Jingning Liu, Wei Tong, Yu Hua, Dan Feng, and Chenye Yu. A software-defined fusion storage system for PCM and NAND flash. In *2015 IEEE Non-Volatile Memory System and Applications Symposium (NVMSA)*, pages 1–6, 2015.

[157] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. Max: A Multicore-Accelerated File System for Flash Storage. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 877–891. USENIX Association, 2021.

[158] Seung-Ho Lim. DeFFS: Duplication-eliminated flash file system. *Computers & Electrical Engineering*, 37(6):1122–1136, 2011.

[159] Seung-Ho Lim and Kyu-Ho Park. An efficient NAND flash file system for flash memory storage. *IEEE Transactions on Computers*, 55(7):906–912, 2006.

[160] Yoohyuk Lim, Jaemin Lee, Cassiano Campes, and Euiseong Seo. Parity-Stream separation and SLC/MLC convertible programming for life span and performance improvement of SSD RAIDs. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, July 2017. USENIX Association.

[161] Chun-Yi Liu, Yu-Ming Chang, and Yuan-Hao Chang. Read leveling for flash storage systems. In Dalit Naor, Gernot Heiser, and Idit Keidar, editors, *Proceedings of the 8th ACM International Systems and Storage Conference, SYSTOR 2015, Haifa, Israel, May 26-28, 2015*, pages 5:1–5:10. ACM, 2015.

[162] Jing Liu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Sudarsun Kannan. File Systems as Processes. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, 2019. USENIX Association.

[163] Chih-Yuan Lu, Kuang-Yeu Hsieh, and Rich Liu. Future challenges of flash memory technologies. *Microelectronic engineering*, 86(3):283–286, 2009.

[164] Youyou Lu, Jiwu Shu, Jia Guo, Shuai Li, and Onur Mutlu. LightTx: A lightweight transactional design in flash-based SSDs to support flexible transactions. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 115–122, 2013.

[165] Youyou Lu, Jiwu Shu, Jia Guo, Shuai Li, and Onur Mutlu. High-performance and lightweight transaction support in flash-based SSDs. *IEEE Transactions on Computers*, 64(10):2819–2832, 2015.

[166] Youyou Lu, Jiwu Shu, and Wei Wang. ReconFS: A Reconstructable File System on Flash Storage. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 75–88, Santa Clara, CA, 2014. USENIX Association.

[167] Youyou Lu, Jiwu Shu, and Jiacheng Zhang. Mitigating Synchronous I/O Overhead in File Systems on Open-Channel SSDs. *ACM Trans. Storage*, 15(3), 2019.

[168] Youyou Lu, Jiwu Shu, and Weimin Zheng. Extending the Lifetime of Flash-based Storage through Reducing Write Amplification from File Systems. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 257–270, San Jose, CA, 2013. USENIX Association.

[169] Youyou Lu, Jiwu Shu, and Weimin Zheng. Extending the Lifetime of Flash-based Storage through Reducing Write Amplification from File Systems. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 257–270, San Jose, CA, 2013. USENIX Association.

[170] Kainan Ma, Ming Liu, Tao Li, Yibo Yin, and Hongda Chen. A Low-Cost Improved Method of Raw Bit Error Rate Estimation for NAND Flash Memory of High Storage Density, 2020.

[171] Charles Manning. YAFFS: the NAND-specific flash file system. *Linuxdevices. org*, 2002.

[172] Charles Manning. How YAFFS works. *Retrieved April*, 6:2011, 2010.

[173] Marí. Understanding and taming SSD read performance variability: HDFS. *ArXiv preprint*, abs/1903.09347, 2019.

[174] Biswajit Mazumder and Jason O. Hallstrom. A Fast, Lightweight, and Reliable File System for Wireless Sensor Networks. In *Proceedings of the 13th International Conference on Embedded Software*, EMSOFT '16, New York, NY, USA, 2016. Association for Computing Machinery.

[175] Dirk Meister, Jurgen Kaiser, Andre Brinkmann, Toni Cortes, Michael Kuhn, and Julian Kunkel. A study on data deduplication in HPC storage systems. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[176] Ralph C Merkle. One way hash functions and DES. In *Conference on the Theory and Application of Cryptology*, pages 428–446. Springer, 1989.

[177] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A large-scale study of flash memory failures in the field. *ACM SIGMETRICS Performance Evaluation Review*, 43(1):177–190, 2015.

[178] Pratik Mishra and Arun Somani. Host managed contention avoidance storage solutions for Big Data. *Journal of Big Data*, 4:18, 2017.

[179] Mahsa Moallem. *A study on the performance evaluation of Linux I/O schedulers*. 2008.

[180] Vidyabhushan Mohan, Sriram Sankar, Sudhanva Gurumurthi, and W Redmond. reFresh SSDs: Enabling

high endurance, low cost flash in datacenters. *Univ. of Virginia, Tech. Rep. CS-2012-05*, 2012.

[181] Vidyabhushan Mohan, Taniya Siddiqua, Sudhanva Gurumurthi, and Mircea R Stan. How I learned to stop worrying and love flash endurance. In *2nd Workshop on Hot Topics in Storage and File Systems (HotStorage 10)*, 2010.

[182] Steve Morgan. The 2020 Data Attack Surface Report. https://1c7fab3im83f5gqiow2qqs2k-wpengine.netdna-ssl.com/wp-content/uploads/2020/12/ArcserveDataReport2020.pdf, 2020. Accessed: 2022-04-28.

[183] Keshava Munegowda, GT Raju, and Veera Manikandan Raju. Evaluation of file systems for solid state drives. In *Proceedings of the Second International Conference on Emerging Research in Computing, Information, Communication and Applications*, pages 342–348, 2014.

[184] Benjamin Nahill and Zeljko Zilic. FLogFS: A lightweight flash log file system. In *2015 IEEE 12th International Conference on Wearable and Implantable Body Sensor Networks (BSN)*, pages 1–6. IEEE, 2015.

[185] FEI Ning, Yi Zhuang, Chun-Ling Chen, and YANG Liang. Design, implementation and evaluation of write-enabled CramFS. *The Journal of China Universities of Posts and Telecommunications*, 18(3):124–128, 2011.

[186] MFXJ Oberhumer. LZO-a real-time data compression library. *http://www.oberhumer.com/opensource/lzo/*, 2008.

[187] Joontaek Oh, Sion Ji, Yongjin Kim, and Youjip Won. exF2FS: Transaction Support in Log-Structured Filesystem. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 345–362, Santa Clara, CA, 2022. USENIX Association.

[188] Jian Ouyang, Shiding Lin, Jiang Song, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: software-defined flash for web-scale internet storage systems. In Rajeev Balasubramonian, Al Davis, and Sarita V. Adve, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014*, pages 471–484. ACM, 2014.

[189] Xiangyong Ouyang, David Nellans, Robert Wipfel, David Flynn, and Dhabaleswar K Panda. Beyond block I/O: Rethinking traditional storage primitives. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 301–311. IEEE, 2011.

[190] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[191] Krishna Parat and Chuck Dennison. A floating gate based 3D NAND technology with CMOS under array. In *2015 IEEE International Electron Devices Meeting (IEDM)*, pages 3–3. IEEE, 2015.

[192] Dongchul Park and David HC Du. Hot data identification for flash-based storage systems using multiple bloom filters. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11. IEEE, 2011.

[193] Dongil Park, Seungyong Cheon, and Youjip Won. Suspend-aware Segment Cleaning in Log-structured File System. In *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*, Santa Clara, CA, 2015. USENIX Association.

[194] Hyunchan Park, Sam H. Noh, and Chuck Yoo. O1FS: Flash file system with O(1) crash recovery time. *Journal of Systems and Software*, 97:86–96, 2014.

[195] Jonggyu Park and Young Ik Eom. Fragpicker: A new defragmentation tool for modern storage devices. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 280–294, 2021.

[196] Jonggyu Park, Dong Hyun Kang, and Young Ik Eom. File Defragmentation Scheme for a Log-Structured File System. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[197] Sang Oh Park and Sung Jo Kim. An efficient multimedia file system for NAND flash memory storage. *IEEE Transactions on Consumer Electronics*, 55(1):139–145, 2009.

[198] Sang Oh Park and Sung Jo Kim. An Efficient Array File System for Multiple Small-Capacity NAND Flash Memories. In *2011 14th International Conference on Network-Based Information Systems*, pages 569–572, 2011.

[199] Sang Oh Park and Sung Jo Kim. ENFFiS: an enhanced NAND flash memory file system for mobile embedded multimedia system. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2):1–13, 2013.

[200] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jinsoo Kim, and Joonwon Lee. CFLRU: a replacement algorithm for flash memory. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 234–241, 2006.

[201] Song-Hwa Park, Tae-Hoon Lee, and Ki-Dong Chung. A Flash file system to support fast mounting for NAND Flash memory based embedded systems. In *International Workshop on Embedded Computer Systems*, pages 415–424. Springer, 2006.

[202] Stan Park, Terence Kelly, and Kai Shen. Failure-atomic msync() a simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 225–238, 2013.

[203] Youngwoo Park, Seung-Ho Lim, Chul Lee, and Kyu Ho Park. PFFS: a scalable flash memory file system for the hybrid architecture of phase-change RAM and NAND flash. In Roger L. Wainwright and Hisham Haddad, editors, *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), Fortaleza, Ceara, Brazil, March 16-20, 2008*, pages 1498–1503. ACM, 2008.

[204] percona. Tokudb. https://github.com/percona/tokudb-engine. Accessed: 2022-07-10.

[205] Ivan Luiz Picoli, Niclas Hedam, Philippe Bonnet, and Pinar Tözün. Open-Channel SSD (What is it Good For). In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[206] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Application crash consistency and performance with CCFS. *ACM Transactions on Storage (TOS)*, 13(3):1–29, 2017.

[207] Vijayan Prabhakaran, Thomas L Rodeheffer, and Lidong Zhou. Transactional Flash. In *OSDI*, volume 8, 2008.

[208] Stephen Pratt and Dominique A Heger. Workload dependent performance evaluation of the linux 2.6 i/o schedulers. In *Proceedings of the Linux symposium*, volume 2, pages 425–448, 2004.

[209] Hongwei Qin, Dan Feng, Wei Tong, Yutong Zhao, Sheng Qiu, Fei Liu, and Shu Li. Better Atomic Writes by Exposing the Flash Out-of-Band Area to File Systems. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2021, page 12–23, New York, NY, USA, 2021. Association for Computing Machinery.

[210] Sheng Qiu and A. L. Narasimha Reddy. NVMFS: A hybrid file system for improving random write in nand-flash SSD. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–5, 2013.

[211] Wenwei Qiu, Xiang Chen, Nong Xiao, Fang Liu, and Zhiguang Chen. A New Exploration to Build Flash-Based Storage Systems by Co-designing File System and FTL. In *2013 IEEE 16th International Conference on Computational Science and Engineering*, pages 925–932, 2013.

[212] Aditya Rajgarhia and Ashish Gehani. Performance and extension of user space file systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 206–213, 2010.

[213] Arul Selvan Ramasamy and Porkumaran Karantharaj. File system and storage array design challenges for flash memory. In *2014 International Conference on Green Computing Communication and Electrical Engineering (ICGCCEE)*, pages 1–8, 2014.

[214] Eunhee Rho, Kanchan Joshi, Seung-Uk Shin, Nitesh Jagadeesh Shetty, Jooyoung Hwang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. FStream: Managing Flash Streams in the File System. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 257–264, Oakland, CA, 2018. USENIX Association.

[215] Bhaskar Prasad Rimal, Admela Jukan, Dimitrios Katsaros, and Yves Goeleven. Architectural requirements for cloud computing systems: an enterprise cloud approach. *Journal of Grid Computing*, 9(1):3–26, 2011.

[216] Ronald Rivest. The MD5 message-digest algorithm. Technical report, 1992.

[217] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[218] Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, and Sang-Won Lee. B+-Tree Index Optimization by Exploiting Internal Parallelism of Flash-Based Solid State Drives. *Proc. VLDB Endow.*, 5(4):286–297, 2011.

[219] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[220] Samsung. Ultra-Low Latency with Samsung Z-NAND SSD. https://semiconductor.samsung.com/resources/brochure/Ultra-Low%20Latency%20with%20Samsung%20Z-NAND%20SSD.pdf, Accessed: 2022-05-02.

[221] Takashi Sato. ext4 online defragmentation. In *Proceedings of the Linux Symposium*, pages 179–186, 2007.

[222] Sebastian Schildt, Wolf-Bastian Pottner, and Lars Wolf. Contiki ring file system for real-time applications. In *2012 IEEE 8th International Conference on Distributed Computing in Sensor Systems*, pages 364–371. IEEE, 2012.

[223] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash Reliability in Production: The Expected and the Unexpected. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 67–80, Santa Clara, CA, 2016. USENIX Association.

[224] Margo I Seltzer, Keith Bostic, Marshall K McKusick, Carl Staelin, et al. An Implementation of a Log-Structured File System for UNIX. In *USENIX Winter*, pages 307–326, 1993.

[225] Eric Seppanen, Matthew T O'Keefe, and David J Lilja. High performance solid state storage under linux. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12. IEEE, 2010.

[226] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A User-Programmable SSD. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 67–80, Broomfield, CO, 2014. USENIX Association.

[227] Mansour Shafaei, Peter Desnoyers, and Jim Fitzpatrick. Write Amplification Reduction in Flash-Based SSDs Through Extent-Based Temperature Identification. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Denver, CO, 2016. USENIX Association.

[228] Mamta Sharma et al. Compression using Huffman coding. *IJCSNS International Journal of Computer Science and Network Security*, 10(5):133–141, 2010.

[229] Hojin Shin, Myunghoon Oh, Gunhee Choi, and Jongmoo Choi. Exploring performance characteristics of ZNS ssds: Observation and implication. In *9th Non-Volatile Memory Systems and Applications Symposium, NVMSA 2020, Seoul, South Korea, August 19-21, 2020*, pages 1–5. IEEE, 2020.

[230] Woong Shin, Qichen Chen, Myoungwon Oh, Hyeonsang Eom, and Heon Y. Yeom. OS I/O Path Optimizations for Flash Solid-state Drives. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 483–488, Philadelphia, PA, 2014. USENIX Association.

[231] YunSeung Shin. Non-volatile memory technologies for beyond 2010. In *Digest of Technical Papers. 2005 Symposium on VLSI Circuits, 2005.*, pages 156–159. IEEE, 2005.

[232] Keith A Smith and Margo I Seltzer. File system aging—increasing the relevance of file system benchmarks. In *Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 203–213, 1997.

[233] Livio Soares and Michael Stumm. *FlexSC*: Flexible System Call Scheduling with *Exception − Less* System Calls. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[234] Yongseok Son, Hyuck Han, and Heon Young Yeom. Optimizing file systems for fast storage devices. In Dalit Naor, Gernot Heiser, and Idit Keidar, editors, *Proceedings of the 8th ACM International Systems and Storage Conference, SYSTOR 2015, Haifa, Israel, May 26-28, 2015*, pages 8:1–8:6. ACM, 2015.

[235] Radu Stoica, Manos Athanassoulis, Ryan Johnson, and Anastasia Ailamaki. Evaluating and repairing write performance on flash devices. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, pages 9–14, 2009.

[236] Hui Sun, Xiao Qin, and Chang-sheng Xie. Exploring optimal combination of a file system and an I/O scheduler for underlying solid state disks. *Journal of Zhejiang University Science C*, 15(8):607–621, 2014.

[237] Anand Suresh, Garth A. Gibson, and Gregory R. Ganger. Shingled Magnetic Recording for Big Data Applications. 2012.

[238] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *USENIX Annual Technical Conference*, volume 15, 1996.

[239] Miklos Szeredi. FUSE: Filesystem in userspace. *http://fuse. sourceforge. net*, 2010.

[240] Andrew S Tanenbaum. Computer systems organization. In *Structured computer organization*, chapter 2.1.5, pages 61–65. Pearson Education India, 2016.

[241] Andrew S Tanenbaum. The microarchitecture level. In *Structured computer organization*, chapter 4.5, page 298. Pearson Education India, 2016.

[242] Vasily Tarasov, Abhishek Gupta, Kumar Sourav, Sagar Trehan, and Erez Zadok. Terra Incognita: On the Practicality of User-Space File Systems. In *7th USENIX Workshop on Hot Topics in Storage and File Systems*

(HotStorage 15), Santa Clara, CA, 2015. USENIX Association.

[243] Nick Tehrany and Animesh Trivedi. Understanding NVMe Zoned Namespace (ZNS) Flash SSD Storage Devices. *ArXiv preprint*, abs/2206.01547, 2022.

[244] Nicolas Tsiftes, Adam Dunkels, Zhitao He, and Thiemo Voigt. Enabling large-scale storage in sensor networks with the Coffee file system. In *2009 International Conference on Information Processing in Sensor Networks*, pages 349–360, 2009.

[245] Yaofeng Tu, Yinjun Han, Zhenghua Chen, Zhengguang Chen, and Bing Chen. URFS: A User-space Raw File System based on NVMe SSD. In *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 494–501, 2020.

[246] ubuntu wiki. IOSchedulers. https://wiki.ubuntu.com/Kernel/Reference/IOSchedulers, 2019. Accessed: 2022-07-10.

[247] Benny Van Houdt. A mean field model for a class of garbage collection algorithms in flash-based solid state drives. *ACM SIGMETRICS Performance Evaluation Review*, 41(1):191–202, 2013.

[248] Bharath Kumar Reddy Vangoor, Prafful Agarwal, Manu Mathew, Arun Ramachandran, Swaminathan Sivaraman, Vasily Tarasov, and Erez Zadok. Performance and Resource Utilization of FUSE User-Space File Systems. *ACM Trans. Storage*, 15(2), 2019.

[249] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To $FUSE$ or Not to $FUSE$: Performance of $User-Space$ File Systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 59–72, 2017.

[250] Vijay Vasudevan, Michael Kaminsky, and David G Andersen. Using vector interfaces to deliver millions of iops from a networked key-value storage server. In *Proceedings of the Third ACM Symposium on Cloud Computing*, pages 1–13, 2012.

[251] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An efficient design and implementation of lsm-tree based key-value store on open-channel ssd. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[252] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In *Annual international cryptology conference*, pages 17–36. Springer, 2005.

[253] Jane Webster and Richard Thomas Watson. Analyzing the Past to Prepare for the Future: Writing a Literature Review. *MIS Q.*, 26, 2002.

[254] Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. ANViL: Advanced Virtualization for Modern Non-Volatile Memory Devices. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 111–118, 2015.

[255] Terry A. Welch. A technique for high-performance data compression. *Computer*, 17(06):8–19, 1984.

[256] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In Martin J. Shepperd, Tracy Hall, and Ingunn Myrtveit, editors, *18th International Conference on Evaluation and Assessment in Software Engineering, EASE '14, London, England, United Kingdom, May 13-14, 2014*, pages 38:1–38:10. ACM, 2014.

[257] David Woodhouse. JFFS: The journalling flash file system. In *Ottawa linux symposium*, volume 2001, 2001.

[258] NVM Express Workgroup. NVM Express NVM Command Set Specification 2.0. Standard, January 2022. Available from: https://nvmexpress.org/specifications.

[259] Bing Wu, Mengye Peng, Dan Feng, and Wei Tong. DualFS: A Coordinative Flash File System with Flash Block Dual-mode Switching. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 65–72, 2020.

[260] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Towards an Unwritten Contract of Intel Optane SSD. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, 2019. USENIX Association.

[261] Wen Xia, Hong Jiang, Dan Feng, and Lei Tian. Combining deduplication and delta compression to achieve low-overhead data reduction on backup datasets. In *2014 Data Compression Conference*, pages 203–212. IEEE, 2014.

[262] Jian Xu and Steven Swanson. $NOVA$: A Log-structured File System for Hybrid $Volatile/Non-volatile$ Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, 2016.

[263] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance analysis of NVMe SSDs and their implication on real world

databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, pages 1–11, 2015.

[264] Hua Yan and Qian Yao. An efficient file-aware garbage collection algorithm for NAND flash-based consumer electronics. *IEEE Transactions on Consumer Electronics*, 60(4):623–627, 2014.

[265] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. *ACM Trans. Storage*, 13(3), 2017.

[266] Jinfeng Yang, Bingzhe Li, and David J. Lilja. Exploring performance characteristics of the optane 3d xpoint storage technology. *ACM Trans. Model. Perform. Evaluation Comput. Syst.*, 5(1):4:1–4:28, 2020.

[267] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. Don't Stack Your Log On My Log. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)*, Broomfield, CO, 2014. USENIX Association.

[268] Jisoo Yang, Dave B Minturn, and Frank Hady. When poll is better than interrupt. In *FAST*, volume 12, pages 3–3, 2012.

[269] Lihua Yang, Fang Wang, Zhipeng Tan, Dan Feng, Jiaxing Qian, and Shiyun Tu. ARS: Reducing F2FS Fragmentation for Smartphones using Decision Trees. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1061–1066, 2020.

[270] Yue Yang and Jianwen Zhu. Algebraic Modeling of Write Amplification in Hotness-Aware SSD. In *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15, New York, NY, USA, 2015. Association for Computing Machinery.

[271] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. SPDK: A Development Kit to Build High Performance Storage Applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161, 2017.

[272] Jinsoo Yoo, Joontaek Oh, Seongjin Lee, Youjip Won, Jin-Yong Ha, Jongsung Lee, and Junseok Shim. Orcfs: Orchestrated file system for flash storage. *ACM Trans. Storage*, 14(2):17:1–17:26, 2018.

[273] Takeshi Yoshimura, Tatsuhiro Chiba, and Hiroshi Horii. EvFS: User-level, Event-Driven File System for Non-Volatile Memory. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, 2019. USENIX Association.

[274] Young Jin Yu, Dong In Shin, Woong Shin, Nae Young Song, Jae Woo Choi, Hyeong Seog Kim, Hyeonsang Eom, and Heon Young Yeom. Optimizing the block I/O subsystem for fast storage devices. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–48, 2014.

[275] Jiacheng Zhang, Youyou Lu, Jiwu Shu, and Xiongjun Qin. Flashkv: Accelerating kv performance with open-channel ssds. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):1–19, 2017.

[276] Jiacheng Zhang, Jiwu Shu, and Youyou Lu. ParaFS: A Log-Structured File System to Exploit the Internal Parallelism of Flash Devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 87–100, Denver, CO, 2016. USENIX Association.

[277] Runyu Zhang, Duo Liu, Xianzhang Chen, Xiongxiong She, Chaoshu Yang, Yujuan Tan, Zhaoyan Shen, and Zili Shao. LOFFS: A Low-Overhead File System for Large Flash Memory on Embedded Devices. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[278] Runyu Zhang, Duo Liu, Xianzhang Chen, Xiongxiong She, Chaoshu Yang, Yujuan Tan, Zhaoyan Shen, Zili Shao, and Lei Qiao. ELOFS: An Extensible Low-overhead Flash File System for Resource-scarce Embedded Devices. *IEEE Transactions on Computers*, pages 1–1, 2022.

[279] You Zhou, Qiulin Wu, Fei Wu, Hong Jiang, Jian Zhou, and Changsheng Xie. Remap-SSD: Safely and Efficiently Exploiting SSD Address Remapping to Eliminate Duplicate Writes. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 187–202. USENIX Association, 2021.

[280] Yue Zhu, Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, Muhib Khan, and Weikuan Yu. Direct-fuse: Removing the middleman for high-performance fuse file system support. In *Proceedings of the 8th International Workshop on Runtime and Operating Systems for Supercomputers*, pages 1–8, 2018.

[281] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

[282] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE*

transactions on Information Theory, 24(5):530–536, 1978.

[283] Aviad Zuck, Ohad Barzilay, and Sivan Toledo. NANDFS: A Flexible Flash File System for RAM-Constrained Systems. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, page 285–294, New York, NY, USA, 2009. Association for Computing Machinery.