LightFS: A Lightweight Host-CSD Coordinated File System Optimizing for Heavy Small File Accesses

Jiali Li, Zhaoyan Shen, Duo Liu, *Member, IEEE*, Xianzhang Chen, *Senior Member, IEEE*, Kan Zhong, Zhaoyang Zeng, and Yujuan Tan

***Abstract*—Computational storage drive (CSD) improves the data processing efficiency by processing the data within the storage. However, existing CSDs rely on the host-centric file systems to manage the data, where the layouts of files are retrieved by the host and sent to the CSD, resulting in additional I/O overhead and reduced processing efficiency, especially in heavy small file accesses. Moreover, the lack of consistency mechanisms poses potential consistency issues. To address these challenges, we propose LightFS, a lightweight host-CSD coordinated file system for the CSD file management. To reduce task offloading overhead, LightFS builds an index file *.ndpmeta* which summarizes the files' metadata and shares between the host and CSD to enable CSD to retrieve the file layout in storage directly. To ensure consistency, LightFS employs a metadata locker and an update synchronizer. The metadata locker leverages the out-of-place update feature of the flash to capture a snapshot of the file to be written without any data copy, while the update synchronizer triggers metadata updates by monitoring the addresses of written blocks to ensure that the modified file is successfully written to the CSD. We implement and evaluate LightFS on a real testbed, and the results demonstrate that LightFS achieves 3.66× performance improvement on the average in real-world operations.**

***Index Terms*—Computational storage, file system, in-storage computing, near-data processing (NDP).**

I. Introduction

THE AMOUNT of data generated worldwide is expected to grow to 175 ZB by 2025 [1], however, the "storage wall" problem posed by slow storage interfaces severely hampers the efficiency of the data processing [2], [3]. Near-data processing (NDP) architecture is seen as an effective way to solve the storage wall problem by processing the data within storage to avoid the heavy I/O transmission overhead. The storage device that supports in-storage computing tasks (i.e., NDP task) is called the computational storage drive (CSD), which is widely used in the database [4], [5], [6], recommendation system [7], [8], AI [9], etc. To keep compatibility with the existing I/O stacks, many CSDs [10], [11], [12], [13] utilize the file systems for the data management. However, the adoption of traditional file systems will lead to performance and consistency problems.

When the host offloads an NDP task to CSD, CSD needs to retrieve the file data in storage directly. However, the traditional storage devices only support the block interface [14], leaving CSDs unaware of file semantics. Thus, how to tell the CSD where to find the file, i.e., retrieve file layout is an essential step in CSD task offloading. There are two types of file management methods used in CSD, the host-centric file system [10], [11], [15], [16] and the in-storage file system [12], [17], [18], as shown in Fig. 1.

1) *Host-centric file system* applies the traditional file system architecture where the file is managed by the host completely. As shown in Fig. 1(a), when offloading the NDP task, the host needs to ❶ read the flash to get the file metadata block and ❷ retrieve the file layout (i.e., the block numbers of the file). The directories and inodes are read level by level, hence steps ❶ and ❷ typically need to be repeatedly executed several times before retrieving the file layout. After that, the host ❸ sends NDP request, including operations and file layout to CSD. The task manager in CSD ❹ retrieves the file data according to the given layout and ❺ sends it to the computing unit.
2) *In-storage file system* offloads the whole file system into storage devices and provides the file interface to the host. As shown in Fig. 1(b), host ❶ sends the NDP request, including operations and file path to CSD after ❷ permission check in kernel. After the task manager receives the file path, it will ❸ retrieve the file layout from the in-storage file system by ❹ multiple flash reads, like the host-centric file system. Finally, the retrieved file data is ❺ computed.

These methods incur two problems as follows.

Manuscript received August 2, 2024; revised GENMONTH, GENDAY GENYEAR; accepted August 3, 2024. This work was supported in part by the National Natural Science Foundation of China under Project 62372073 and Project 62072059; in part by the Fundamental Research Funds for the Central Universities under Project 2023CDJXY-039, Project 2024CDJGF-032, and Project 2024CDJGF-003; in part by the Chongqing Post-Doctoral Science Foundation under Project 2021LY75 and Project cx2023080; and in part by the Open Project Program of Wuhan National Laboratory for Optoelectronics under Project 2022WNLOKF018. This article was presented in the International conference on compilers, architectures, and synthesis for embedded systems (CASESs) 2024 and appears as part of the ESWEEK-TCAD Special Issue [DOI: 10.1109/TCAD.2024.3443010]. This article was recommended by Associate Editor S. Dailey. *(Corresponding author: Duo Liu.)*

Jiali Li, Xianzhang Chen, Zhaoyang Zeng, and Yujuan Tan are with the College of Computer Science, Chongqing University, Chongqing 400044, China.

Zhaoyan Shen is with the School of Computer Science and Technology, Shandong University, Shandong 266590, China, and also with the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, China.

Duo Liu and Kan Zhong is with the School of Big Data and Software Engineering, Chongqing University, Chongqing 400044, China (e-mail: liuduo@cqu.edu.cn).

Digital Object Identifier 10.1109/TCAD.2024.3443010

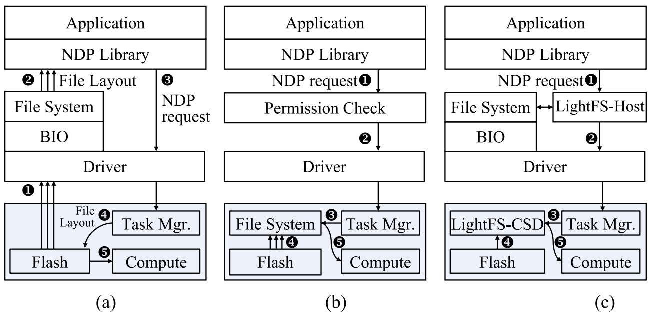


Fig. 1. Architecture comparisons between traditional architectures and LightFS. (a) Host-centric file system. (b) In-storage file system. (c) LightFS.

1) *Performance Degradation:* Is caused by the structure of the traditional file system. Before sending an NDP request, file path parsing and metadata reading need to be done level by level, resulting in significant additional I/O overhead, and the NDP request has to be blocked before the layout is retrieved. Even worse, with the development of AI, social networks, embedded devices, etc., the data is often stored as large numbers of small files [19], [20], [21], leading to heavy small file accesses. When confronted with this scenario, CSD must retrieve the layout of each file independently, resulting in additional I/O overhead that amounts to over 50% (detailed in Section II-C), diminishing the advantage of CSD in retrieving and processing the data within the storage. Although the in-storage file system can reduce the communication time with the host, it still requires a lot of additional flash reads to retrieve the file layout.
2) *Data Consistency:* Due to the presence of page cache, out-of-order writes[22], and I/O schedulers[23] in the operating system, the exact timing of when a file is written to the storage is unknown. If an NDP request is sent on a writing file, inconsistent data may be read by the NDP request (detailed in Section II-D). Typically, current CSDs address this issue by file locking and force synchronization [10], [11], but this approach results in request blocking and reduced processing efficiency.

To address these problems, we propose a lightweight file system LightFS that runs across the host and CSD. The architecture of LightFS is shown in Fig. 1(c). LightFS consists of two components: 1) LightFS-Host and 2) LightFS-CSD. Their primary roles are to handle the user requests in the host and manage the file metadata in the CSD. LightFS tackles the aforementioned problems with the following two key designs.

1) To reduce task offloading overhead, LightFS builds an in-storage index file, *.ndpmeta*, which is recognizable by both LightFS-Host and LightFS-CSD. This allows the CSD to directly retrieve the file layout from the storage. As shown in Fig. 1(c), when LightFS-Host ❶ receives an NDP request, it converts the file/directory path to an inode number (ino) list and ❷ sends it to the CSD after a permission check. LightFS-CSD ❸ retrieve the file layout using the ino rather than the full path name. The *.ndpmeta* file records the layout of each file, so LightFS-CSD can ❹ retrieve the layouts of a batch of files with a single flash read and quickly send the flash read requests for ❺ computing, avoiding separate handling for each file.
2) To ensure consistency, LightFS employs a metadata locker and an update synchronizer. The metadata locker locks the file before a file is written, and the update synchronizer monitors written blocks to update the metadata immediately when all the blocks of a file are written. To avoid NDP task blocking when file writing, LightFS takes advantage of the out-of-place update mechanism of flash, records the physical addresses of the file, and forbids garbage collection before the file is written to CSD. This mechanism improves concurrency performance while ensuring the file data consistency.

We implement the LightFS prototype based on Linux and a real testbed cosmos plus OpenSSD [24], to demonstrate the performance improvement of LightFS with the host-centric and in-storage file system. The experimental results show that LightFS can achieve an average of 88× (with warmup) and 26.8× (without warmup) task offload acceleration when processing heavy small files. For real-world operations, LightFS can achieve 3.66× data processing accelerations on average.

In summary, the contributions of this article include as follows.

1) As far as we know, we are the first to reveal the consistency problem and the I/O overhead in heavy small file access in existing CSDs, providing a comprehensive understanding through the detailed experimental analysis.
2) We propose LightFS, a lightweight host-CSD coordinated file system that innovatively builds and shares the index files between the host and CSD to reduce the file access overhead in NDP task offloading, while maintaining compatibility with the existing software.
3) We implement a LightFS prototype on a real testbed and demonstrate the performance improvement by compared with widely used file management methods in CSDs.

The remainder of this article is organized as follows. In Section II, we introduce the task offloading overhead and inconsistency problems in CSD. Section III presents the design of LightFS. Section IV evaluates the LightFS and analyses the result. Section V concludes this article.

II. Background and Motivation

In this section, we will introduce the task offloading workflow of current CSDs, then reveal the performance and inconsistency problems in detail.

A. Background of CSD and Flash

The concept of CSD was first proposed for the hard disk drive (HDD) [3]. However, due to the slow speed of HDD, storage interface bandwidth was not a performance bottleneck. With the development of flash, solid-state drive (SSD) bandwidth has increased significantly, making storage interfaces gradually become the bottleneck for the data processing [11], thus SSD-based CSD became the research hotspots [5]. Compared to the commercial SSD, CSD [25]

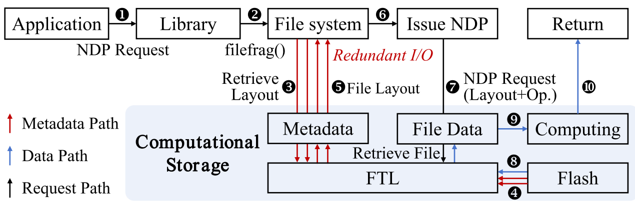


Fig. 2. NDP task offloading workflow.

can directly access data and process it by built-in hardware accelerators [26] or embedded processors [7], and then only return results back to the host. CSD avoids the transfer of large amounts of the raw data, thereby improving the data processing efficiency.

SSD and CSD typically use NAND flash as the storage medium [6], [27]. The read and write granularity of NAND flash is page, usually with sizes of 4, 16 KB, or higher. NAND flash chip is addressed by discrete physical addresses, but SSD needs to expose continuous logical addresses to the host. To bridge the gap, SSD utilizes a mapping table called flash translation layer (FTL) to translate the user-requested logical addresses into physical addresses of flash. NAND flash features write-after-erase, meaning before writing to a page, the block that the page locates needs to be erased first, a block typically comprising several dozen pages. Erasing incurs high overheads, so SSDs usually write modified data to another block's page, and then update the FTL to point that logical address to the new physical page. This strategy is known as the SSD's out-of-place update strategy [28]. Therefore, until a block is erased, the data inside can still be accessed via the physical addresses, a feature that LightFS utilized.

B. *NDP Task Offloading Workflow*

Many CSDs still utilize the block interface to communicate with the host, and the host employs the traditional file systems to manage the data on the CSD [10], [11], [15] for compatibility purposes. Due to this limitation, CSD can only read the data by the block number and lacks the file system semantics inside CSD. Therefore, these file system-based CSDs require the host to use the *filefrag* function to call the *fiemap* system call to obtain the layout of files.

The typical NDP task offloading workflow is depicted in Fig. 2. When an application ❶ sends an NDP request to the library, it utilizes the ❷ *filefrag* function to obtain the file layout from the file system. Subsequently, the file system executes the "❸ resolve the file path and sends a read request to retrieve the data block of the next path level, → ❹ read the flash, and → ❺ return the metadata flow" layer by layer until retrieving the layout of the given file path. Once the layout is ❻ returned to the library, the NDP request, including file layouts and operations, is ❼ sent to the CSD. CSD then ❽ retrieve the file data based on the given file layouts, then ❾ send the file to the computing unit, and finally ❿ return the result.

Although in-storage file systems can avoid frequent data transfers between CSD and host during the file layout retrieval, existing in-storage file systems still utilize the traditional file

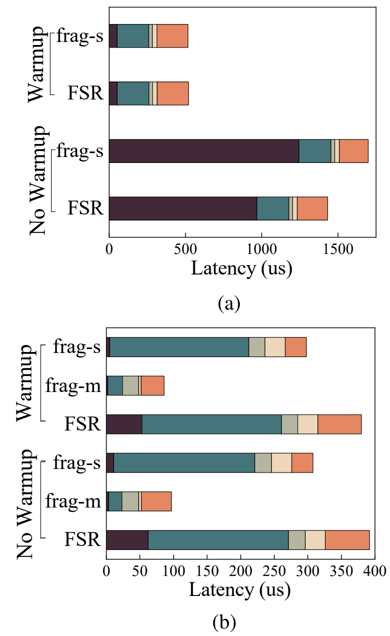


Fig. 3. Latency breakdown of motivation examples. (a) Single file request. (b) Multiple file request.

systems [12], [18], so multiple flash reads within the CSD are still required to obtain the layout of a file.

C. *NDP Task Offloading Overhead*

As described above, each NDP task needs to retrieve the file layout before offloading, which brings a lot of redundant I/O and reduces the CSD efficiency, especially when dealing with heavy small file access. Heavy small file accesses refer to scenarios, such as log analysis and retrieval [21], sensor data analysis [20], and dataset access during AI training [29], [30]. These applications need to access a batch of small files during execution. If the host sends an NDP request for each file separately, the additional overhead brought by *filefrag* and communication is significant. We demonstrate the overhead through an experiment. We employ three methods for offloading NDP tasks: 1) single-threaded *filefrag* (referred to as frag-s); 2) multithreaded *filefrag* (referred to as frag-m); and 3) an in-storage file system FSR [12], [31]. We offload STATS64 [10] NDP tasks for 1 [Fig. 3(a)] and 100 [Fig. 3(b)] 16 KB files, respectively, and measure the average latency per file with breakdowns. Details of the experimental setup can be found in Section IV-A.

The results are shown in Fig. 3. We define a metric, percentage of additional overhead (PAO), to quantify the unnecessary overhead during the NDP task processing of each file, which includes operations, such as retrieving and transferring the file layouts. PAO is defined as

$$\text{PAO} = \frac{\text{Retrieve File} + \text{DMA} + \text{Others}}{\text{Total Latency}}.$$

A higher PAO indicates lower efficiency of the NDP tasks, the ideal value of PAO is 0. When processing a single file [Fig. 3(a)], if the system is warmed up, both frag-s and FSR have a PAO of 55.3%; if not, frag-s and FSR have PAOs of

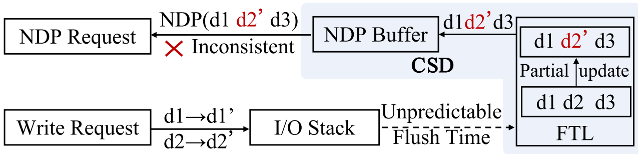


Fig. 4. Inconsistency problem in CSD.

86.2% and 83.7%, respectively, due to the need for multiple I/O operations to read the file metadata from the storage. When processing multiple warmed-up files [Fig. 3(b)], the PAOs for frag-s, frag-m, and FSR are 22.5%, 46.5%, and 39%, respectively. The higher PAO for frag-m is because it can leverage the parallelism of flash channels, resulting in much lower read times compared to the other methods. However, frag-m cannot eliminate additional overhead, such as individually sending requests for each file, resulting in high PAO, significantly impacting the execution efficiency of NDP tasks.

Therefore, frag-s and FSR fail to fully exploit the parallelism of flash channels, leading to significantly inferior performance compared to frag-m. However, in frag-m, the overhead of individual file layout retrieving and task sending takes up a lot of time. To address these issues, LightFS stores the layout of small files in an index file and supports in-storage layout retrieval. This approach optimizes the parallelism of flash channels, and minimizes the number of flash reads and communications between the host and CSD.

D. *Inconsistency Problem*

After a file is modified by an application, the actual written time of the file data is unknown due to the page cache and I/O scheduler in the operating system. This does not pose consistency problems in traditional storage devices because the host can track the write status. However, the CSD cannot access the host I/O stack status and must read directly from the storage, potentially reading a partially modified file.

We illustrate this potential consistency problem with a simple example as shown in Fig. 4. Suppose a file contains blocks $(d1, d2, d3)$, and the host sends an NDP request to process this file in storage. If the file is written before or during the NDP operation, block $(d1, d2)$ is modified to block $(d1', d2')$ in the host. But the write order and time of $(d1', d2')$ is not determined, so the data read by NDP request may be an inconsistent state, such as $(d1', d2, d3)$ and $(d1, d2', d3)$, resulting in an error result. Some CSDs prevent this problem by locking the file [10], [11], preventing the NDP request and write request from executing simultaneously. However, file lock does not eliminate this problem because the updated file may delay writes, the data still may be written during the execution of the NDP request, thus the NDP request still will read the inconsistent data. Some CSDs force file synchronize [15] after each file write to ensure that the file data is written to storage, but this will cause task blocking and reduce the performance of the CSD. To address this problem, LightFS takes a snapshot of written files without any data copy by recording the physical address of files and prohibits garbage collection of the block. And LightFS accurately triggers the metadata updates by monitoring the written blocks.

III. LightFS Design

In this section, we will introduce the design details of LightFS, including architecture, workflow, and synchronization mechanism.

A. *Overview of LightFS*

LightFS aims to enable CSDs to directly retrieve file layouts in storage and minimize the number of required I/O operations to read the file layouts. LightFS also needs to ensure the consistency of retrieved files, i.e., preventing access to files that are updating. To achieve this, LightFS constructs an index file called *.ndpmeta* which is recognizable by both the host and CSD. This file stores the metadata required for the offloading NDP tasks. LightFS consists of two components: 1) LightFS-Host and 2) LightFS-CSD, which seamlessly collaborate to efficiently offload NDP tasks and update the metadata.

The structure of LightFS is shown in Fig. 5. LightFS-Host is an user-space process that receives the user requests and handles the metadata updates in the background. The main functions of LightFS-Host include as follows.

1) Checking permissions for NDP requests, converting paths into ino list, and sending them to LightFS-CSD.
2) Sending write notification requests to LightFS-CSD before write requests.
3) Recording written files and periodically executing synchronization then sending the updated file layouts to LightFS-CSD.

The primary functions of LightFS-CSD include as follows.

1) The NDP task dispatcher retrieves the file layouts from the metadata cache in CSD, and then reads them for process.
2) The metadata locker records the physical addresses of files in write notifications requests and forbids garbage collection at those addresses.
3) The update synchronizer monitors written block numbers and triggers metadata updates after blocks are written. Both host and CSD have a metadata cache, each caching a portion of the *.ndpmeta* file. Detailed data layouts and workflows are presented below.

B. *Layout of LightFS*

To keep compatibility with the existing I/O stacks and file systems, LightFS stores the index file *.ndpmeta* on the backend file system. Upon initialization, *.ndpmeta* is generated in each directory that needs to be processed by CSD and stores the metadata of every file and subdirectory within that directory. The primary fields of *.ndpmeta* are illustrated in Table I.

1) Data Structure: The *.ndpmeta* file comprises two parts: 1) a header and 2) a body with multiple entries, which, respectively, store the metadata about each file and *.ndpmeta* in subdirectories. The header primarily contains the number of files/subdirectories in this directory and permissions information about them. The aggregated permissions are used to efficiently check permissions. When sending NDP requests

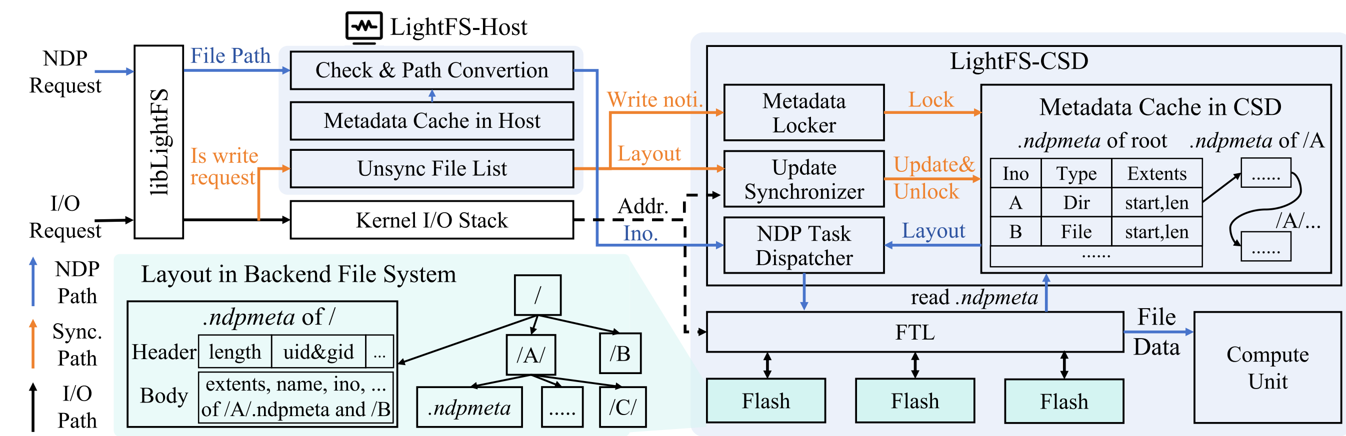


Fig. 5. Architecture of LightFS.

TABLE I
MAIN FIELDS OF *.ndpmeta*

	Field	Description	Size(B)
Header	uid, gid, mode	Aggregated permissions of files	10
	size	Size of *.ndpmeta*	4
	file count	Number of store files	4
Body Entry	name	Name of dir/file	32
	extents	Layout of file	64
	ino	Inode number	8
	length	File length	8
	uid, gid, mode	File/dir permissions	8
	flags	Indicate status, e.g. is file or dir	2

for a large number of small files (e.g., "scan directory /A/"), reading the inode of each file to check permissions can be time consuming. To mitigate this, we summarize the permissions field (i.e., uid, gid, and mode) of all the files within the header if their permissions field is the same. This design is predicated on the assumption that the files in a directory typically have the same permissions. Thus, if files in a directory share the same permissions, they are checked once in the header to avoid redundant I/O before NDP task sending. However, if there are files with different permissions, the permissions field will not be set, and LightFS needs to check permissions file-by-file.

The body of *.ndpmeta* mainly consists of 1) the extent (i.e., file layout) of each file or subdirectory, to retrieve the file in storage directly; 2) the name of the file, to efficiently resolve the file path; and 3) the ino of the file, to send requests to CSD. Notably, the extents field of a subdirectory's entry stores the layout of the *.ndpmeta* file within it, enabling CSD to directly obtain the layout of subdirectories. The name and extents field may not be long enough to store the complete file name and layouts of a file. When the file name is too long, we use the space of the file's next entry in the body and mark it in the flag to avoid space waste in short file names. If there are too many extents, it means that this is not a small file and LightFS will give up on storing the layout of this file. At this time LightFS takes the traditional method, to get the file layout through the *filefrag* and send it to the CSD.

2) Metadata Cache: Host and CSD load *.ndpmeta* into the metadata cache upon the system startup and cache misses. However, the memory of CSD is limited [24]. Fortunately, the LightFS-Host and LightFS-CSD only require certain fields of *.ndpmeta*, thus the metadata caches of host and CSD, respectively, cache specific fields of *.ndpmeta*.

The main function of LightFS-Host is to check requests and convert path names to ino. Therefore, its metadata cache mainly consists of aggregated uid, gid, and mode for quick permission comparison, as well as the names and ino of files/subdirectories for searching the file names and converting them into the ino lists. The main function of LightFS-CSD is to retrieve the required file layout based on the ino lists. Therefore, its metadata cache mainly contains extents for reading the files, ino for retrieving the files corresponding to the ino lists in requests, and flags for determining whether it is a file or a subdirectory.

3) Space Overhead: LightFS requires additional storage and memory space to store and cache *.ndpmeta* as follows.

1) *Storage Space Overhead:* Typically, LightFS only needs an additional 128B of space for each file and directory. When the average file size is 16 KB, the additional space overhead is only 0.78%;
2) *Memory Footprint:* when caching the metadata of 10 000 files, the memory footprint for the host and CSD is 625 and 723 KB, respectively (since neither the host nor the CSD needs to cache the entire index file). This overhead is much smaller than the memory size of mainstream CSDs [9], [24].

C. Workflow of LightFS

We will introduce the workflow of LightFS from the perspective of different request execution processes.

1) Initialize: Before using LightFS, initialization is required, similar to formatting in other file systems. The purpose of initialization is to generate a *.ndpmeta* file in the directory specified by the user. LightFS iteratively fetches metadata for each file or directory to build the *.ndpmeta* file. Since the parent directory needs to store the metadata of the *.ndpmeta* in subdirectories, LightFS uses a depth-first approach to build from the bottom directory upward. After initialization, *fsync* is used to ensure consistency and allow LightFS-CSD to read out the root directory's *.ndpmeta*. The initialization only needs to be executed once for each directory.

In subsequent system startups, LightFS read the *.ndpmeta* file from the underlying file system to build the metadata cache.

2) I/O Request: Applications use libLightFS for POSIX-like file read and write operations, interacting with the LightFS-Host. Only write operations may cause inconsistencies, so libLightFS checks for write flags in open requests, sending a notification to lock the file layout if needed. Afterward, write requests are sent directly to the kernel I/O stack. The written file path is added to an unsync list, and LightFS-Host periodically executes *sync* to allocate blocks for these files. Unlike *fsync*, *sync* avoids blocking by pushing I/O requests to the kernel, reducing overhead. After a file is closed, its layout is retrieved by *filefrag* and updated in metadata, with low overhead (about $5us$) since the file is still in memory. Detailed descriptions of the metadata locker and update synchronizer are in Sections III-D and III-E.

3) NDP Request: Applications can send NDP requests to LightFS-Host through libLightFS. LightFS-Host first checks whether the permissions of the application process match those requested for the file or directory. If the permission check passes, LightFS-Host resolves the path in the metadata cache to locate the corresponding entry. During locating, LightFS-Host records the inos of each level of the path, forming an ino list, which is then sent to the NDP task dispatcher in LightFS-CSD. The reason for sending the ino list instead of the raw path string includes as follows.

1) Reducing DMA data transfer volume.
2) Decreasing CSD memory usage, as only inos need to be cached in CSD for the file indexing rather than the strings.
3) Speeding up CSD matching, as matching each directory or file requires only integer calculations without time-consuming string parsing and matching.

After receiving the ino list, the NDP task dispatcher obtains the target file's extents (if the inode list points to a file), or all the file's extents (if the inode list points to a directory) inside the target directory from the metadata cache. Then read data based on these extents and processes by the computing unit. In particular, if the file is being locked by a metadata locker, then the physical address will be fetched directly. If the target inode is not cached in the metadata cache, the *.ndpmeta* file will be read level by level to build the cache.

D. Metadata Locker

When an NDP request processes a written but not fully flushed file, the partially written file data will affect the consistency of the data read by the NDP request. Therefore, the file must be locked before the file is written to CSD completely. In order to lock files and avoid request blocking as much as possible, we propose the metadata locker. The key idea of the metadata locker is to leverage SSD's out-of-place update mechanism as mentioned in Section II-A to lock the previous version of the file. Thus, before modifying a file, the metadata locker locks the file by recording the file's physical addresses and forbids garbage collection on these addresses, equivalent to saving a snapshot of this file but without any copy of the data. NDP requests retrieve the required data through

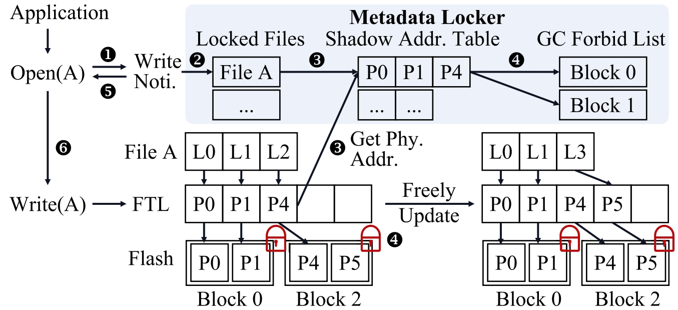


Fig. 6. Workflow of metadata locker.

these physical addresses, thus avoiding NDP request blocking caused by the file writes. The workflow of metadata locker is shown in Fig. 6.

Specifically, upon libLightFS receiving an open request with a write flag, LightFS-Host will ❶ send a write notification request to the metadata locker of LightFS-CSD. The meaning of the write notification request is to notify CSD that the file is about to be written, but it is unpredictable when the data will be written to the storage. The application cannot write data to the backend file system until the write notification request returns because if the file system writes the new data to the original logical address, the logical address recorded in LightFS-CSD will point to the partially updated data. Although the metadata locker leads to task blocking on the subsequent write requests (locking each file takes about $20us$ by our evaluation), it is still better than the file lock in that the metadata locker only needs to block the requests for a period of time when opening, without impacting the subsequent write requests.

When the metadata locker receives a write notification request, it ❷ adds the ino from the request to the locked file list and ❸ creates a shadow address table for the file. The shadow address table records the physical addresses of the locked file. The metadata locker first locates the logical address of the file in the metadata cache and finds the corresponding physical address in the FTL, then ❸ adds these physical addresses to the shadow address table. Subsequently, the metadata locker ❹ adds the flash blocks corresponding to these physical addresses to the GC forbid list, indicating that garbage collection operations cannot be performed on these blocks. Once the above steps are completed, the signal ❺ returns to the open functions, allowing applications to ❻ freely write. When an NDP request needs to retrieve the layout of the file, it will get the physical addresses of the file from the metadata locker. Therefore, the metadata locker ensures the consistency of files and avoids the task blocking to improve the request concurrency.

E. Update Synchronizer

The metadata locker effectively locks the file, but determining the time of unlocking the file and updating the metadata poses a challenge. As discussed in Section II-D, data written by the host's traditional I/O stack may not be immediately written to storage, leading to unpredictable write

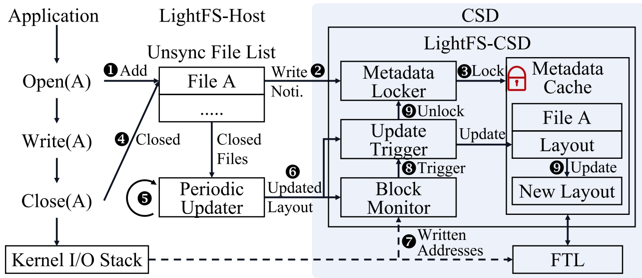


Fig. 7. Workflow of update synchronizer.

times. For instance, if LightFS-CSD unlocks the file and updates metadata as soon as the file is closed, the file data may not yet be written to the CSD, resulting in NDP requests reading invalid the data. To address this issue, we propose the update synchronizer, which unlocks and updates LightFS when all the file data has been written to the CSD. The key idea of the update synchronizer is to monitor written blocks in the CSD and trigger metadata updates when all the blocks of a file data are received, to update the metadata as promptly as possible while maintaining the metadata consistency.

The workflow of the update synchronizer is depicted in Fig. 7. It comprises a block monitor and update trigger in LightFS-CSD along with a periodic updater in LightFS-Host. When an application opens a file, it performs several steps: ❶ adding the file to the unsync file list, ❷ sending a write notification to CSD, and ❸ locking the file using the metadata locker and writing the file to the kernel I/O stack as described in Section III-D. Upon completing the file write and closing it, libLightFS ❹ notifies LightFS-Host to mark the file as closed. To prevent excessive *sync* calls after each file write, LightFS employs the periodic updater to synchronize files at regular intervals. This periodic updater ❺ calls *sync* for each closed file in the unsync file list and retrieves the updated layout. The updated layout is then ❻ sent to both the update trigger and block monitor. The block monitor ❼ monitors the written addresses and compares them with the sent layout. If all addresses are received, it will ❽ trigger the update trigger. Subsequently, the update trigger ❾ updates the file layout in the metadata cache and notifies the metadata locker to unlock the file.

Update of .ndpmeta: The update synchronizer only updates the metadata cache in LightFS-CSD. The updated *.ndpmeta* file is then written back by the host because it requires modification on the backend file system, only the host can update it. However, *.ndpmeta* needs to record the layout of *.ndpmeta* in subdirectories. Consequently, when a file is written in a directory, the *.ndpmeta* files in both the directory and its parent directory require updating. In that case, the issue of delayed writes and out-of-order writes of *.ndpmeta* files persists. If the cache of a *.ndpmeta* file is evicted and subsequently needs to be read again, there may still exist partially updated data. Therefore, updating *.ndpmeta* by the update synchronizer is still necessary to ensure consistency when the NDP requests retrieve the subdirectories layouts.

Specifically, after LightFS-Host constructs the *.ndpmeta* file for each directory and calls *fsync* to ensure the metadata file is written to the CSD, LightFS-CSD reads the root directory's *.ndpmeta* to initialize the metadata cache, ensuring initial consistency for all files and sub-directories. Before a file write, LightFS caches the metadata of each level in the file path for iterative updates. If files are modified, LightFS periodically writes the updated metadata to the *.ndpmeta* file in the backend file system. During updates, the metadata locker locks the parent directory's *.ndpmeta* to maintain consistent sub-directory layouts for NDP requests. The update synchronizer then updates the parent directory's layout, and this iterative update continues up the directory tree until reaching the root directory or when a *.ndpmeta* is updated in its original location. Hence, to maintain consistency, the root directory's metadata cache must be pinned, preventing eviction and ensuring that consistent data is always retrieved.

F. Limitation of LightFS

LightFS does not make intrusive modifications to the kernel and file system, its efficiency is subject to certain limitations. First, LightFS still relies on *filefrag* to fetch the extent of a file. However, unlike the traditional methods, LightFS retrieves the layout in memory without redundant storage I/O. Moreover, this layout retrieval occurs in the background, thus not impeding the critical path of NDP task offloading. Second, LightFS requires periodic calls of *sync* to ensure the file system has allocated the data blocks. To mitigate this overhead in the future, the block address allocation can be captured in the kernel using eBPF. Third, modifications to a file entail iteratively updating metadata in its parent directory, resulting in issues of wandering trees [32]. Nevertheless, since CSD is typically designed toward read-intensive applications [4], [33], such issues are nonexistent in this scenario. This issue can be solved by adopting an indirect index table similar to F2FS [32], which is our future work.

IV. Evaluation

In this section, we will introduce the experimental configuration, followed by presenting and analysing the performance improvement of LightFS compared to other methods. Our evaluation of LightFS aims to address the following questions.

1) How does LightFS perform under different workloads (e.g., different file sizes and counts)? (Section IV-B).
2) How does LightFS achieve its improvements? (Sections IV-C and IV-D).
3) How does LightFS perform under the concurrent hybrid requests? (Section IV-F).
4) How does LightFS perform under the real-world datasets and operations? (Section IV-E).
5) How is the performance of LightFS on different backend file systems? (Section IV-G).

A. Experiment Setup

1) Platforms: The host and CSD configurations are detailed in Table II. We implement the LightFS-CSD prototype on the cosmos plus OpenSSD platform [24]. It is an open-source programmable SSD equipped with a 1 GB DRAM and a Xilinx ZYNQ XC7Z045 controller with an ARM

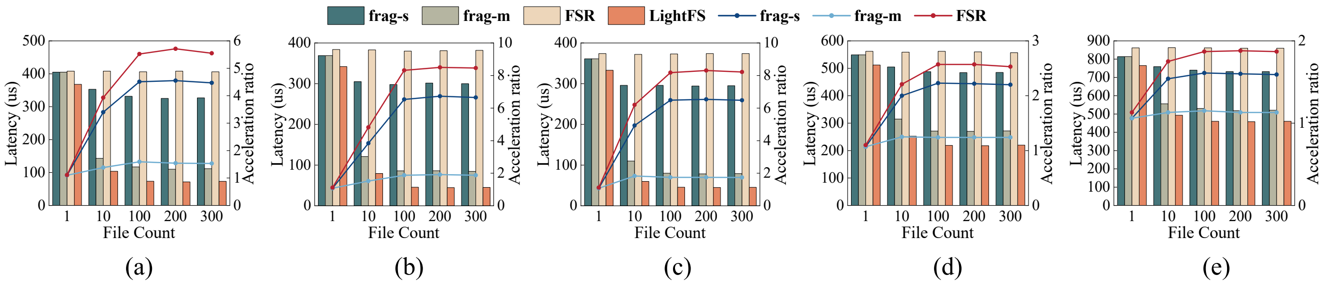


Fig. 8. Average latency under different file count. (a) STATS32. (b) STATS64. (c) KNN. (d) Grep-ACC. (e) Grep-ARM.

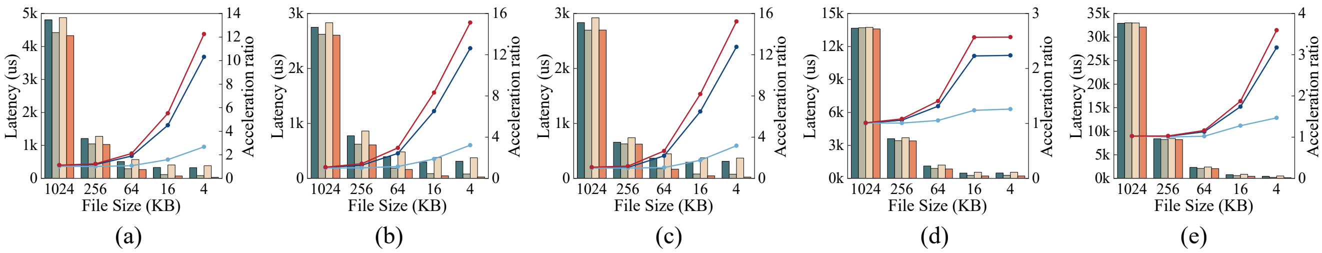


Fig. 9. Average latency under different file size. (a) STATS32. (b) STATS64. (c) KNN. (d) Grep-ACC. (e) Grep-ARM.

TABLE II
HOST CONFIGURATIONS

	Name	Configuration
Host	CPU	E3-1230V2, 8 Cores, 3.3GHz
	Memory	12G, DDR3 1600MHz
	OS	Ubuntu16.04.6 (Kernel version 4.4.4)
CSD	Controller	Xilinx ZYNQ XC7Z045
	Memory	1GB, DDR3
	Flash	1TB, Page Size 16KB

TABLE III
EVALUATION APPLICATIONS

Application	Description	Time (us)
STATS32/64 [10]	Read file data as 64/32 bit integers and calculate the sum.	50/25
KNN [11]	Read file data as 128 dimensions vectors and calculate the L2 distances between a given vector.	17
Grep-ACC/ARM [36]	Read file data as a string, and find a given string by a hardware accelerator (ACC) or the embedded processor(ARM).	203/500

Cortex-A9 processor. We equip cosmos plus OpenSSD with the two flash modules, each with 500 GB capacity and four flash channels, and are connected to the host via an eight-lane PCIe Gen2 interface. We implement the LightFS-CSD based on the firmware Greedy-FTL 2.7.0d [34]. About 1.5K LoC are added to the firmware.

It should be noted that due to inherent compatibility constraints within the OpenSSD [34], the host is unable to utilize cutting-edge CPUs. Nevertheless, NDP tasks usually are I/O-intensive tasks, and the host only needs to retrieve the file layout and offloading tasks, thus the experimental results and conclusion will not be significantly impacted by the performance of the host's CPU. We develop LightFS-Host atop an user-space nonvolatile memory express (NVMe) driver UNVMe [35], which ensures its seamless operability across a wide range of hosts.

2) Comparisons: We compare LightFS with three NDP task offloading methods as follows.

1) *Filefrag-Single(frag-s):* The most widely used task offloading method [10], [11], [15], [16] in computational storage that using the host-centric file systems. We obtain the block address of files through the *fiemap* system call and send them to the CSD.
2) *Filefrag-Multi(frag-m):* The multithreaded Filefrag-Single, i.e., multiple Filefrag-Single threads are created, each of which retrieves the block address and sends NDP request independently. In our evaluations, frag-m usually gets its best performance with eight threads, the same number of I/O command queues as OpenSSD [24]. So we use eight threads in all the afterward experiments.
3) *FSR:* An in-storage file layout retrieve method that directly resolves the path of the file and reads the file in CSD [12].

Except for FSR, the backend file system of all the other methods is Ext4. FSR only supports retrieving the data layout in F2FS currently. Out of fairness, we will show the performance of other methods on different file systems in Section IV-G.

3) Applications: We evaluate five common in-storage computing applications, detailed in Table III, with computation times ranging from 17 to 500 us, noting that due to the weaker performance of Cosmos Plus OpenSSD compared to commercial CSDs [6], [9], execution times are generally longer, but our results show that shorter computation times lead to greater performance improvements with LightFS; all results are averaged over five consecutive measurements, with variance analysis discussed in Section IV-E.

B. Effect on Different Workload

We show LightFS's performance compared to the other methods across different applications, file counts, and file sizes to analyse its performance improvements in various workloads.

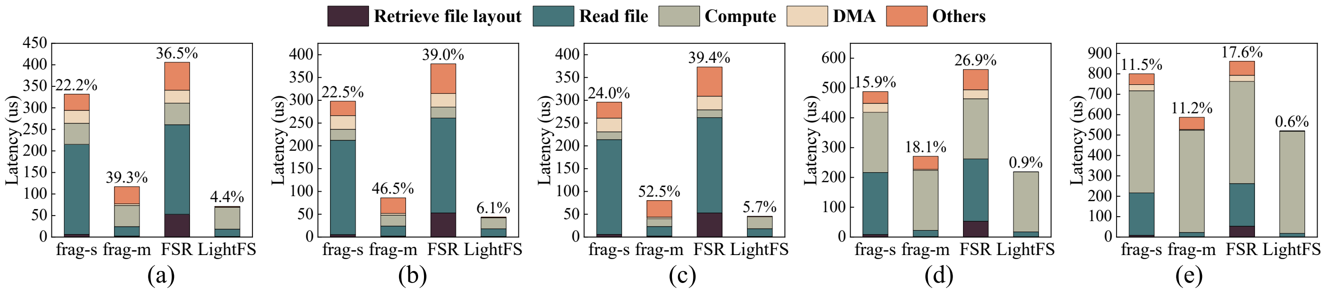


Fig. 10. Latency breakdown under different applications (with warmup). (a) STATS32. (b) STATS64. (c) KNN. (d) Grep-ACC. (e) Grep-ARM.

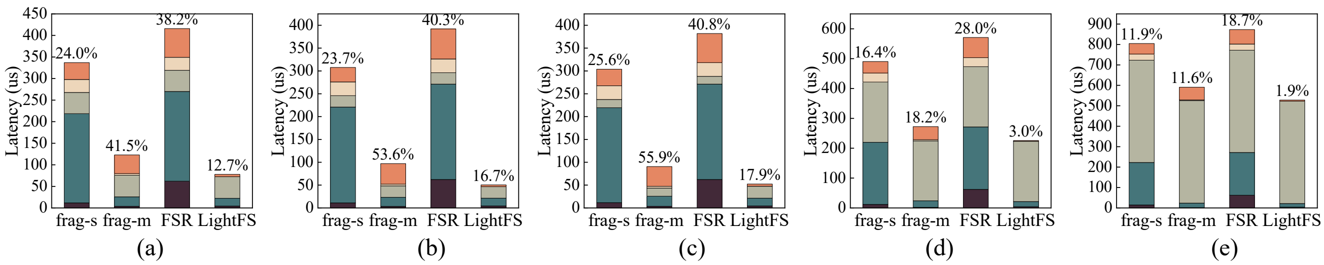


Fig. 11. Latency breakdown under different applications (without warmup). (a) STATS32. (b) STATS64. (c) KNN. (d) Grep-ACC. (e) Grep-ARM.

For performance under different file counts, we fix the file size as 16 KB and gradually increase the number of files as shown in Fig. 8. The column depicts the average latency of processing each file under different methods, while the line depicts the acceleration ratio of LightFS compared to the other methods. LightFS demonstrates performance improvements across all the workloads. As the number of files increases, LightFS's performance rapidly improves. It reaches its peak at 100 files and remains stable as the number of files continues to increase. This is because LightFS's performance enhancement comes from batch-fetching file layouts to rapidly send the flash read operations in CSD, thereby improving the flash concurrency and avoiding the overhead of individually fetching metadata and sending requests for each file. Hence, as the number of files increases, LightFS's advantage in batch-fetching metadata becomes more significant.

For performance under different file sizes, we fix the number of files at 100 and gradually increase the file sizes as shown in Fig. 9. LightFS also presents performance improvements in all the cases with the performance improvement being more significant as the file size decreases. This is because when the file size is smaller, the time taken for data retrieval and computation is shorter. As a result, the PAO that can be avoided by LightFS becomes higher, leading to more significant performance gains. It is worth noting that in Grep-ACC, the performance improvements for 16 and 4 KB are nearly the same. This is because both the Grep-ACC's hardware accelerator and flash read granularity are 16 KB, resulting in the same latency for them.

Even in scenarios with fewer or larger files, LightFS consistently outperforms other methods. As the number of files decreases or file size increases, the speedup approaches but does not drop below 1, as shown in Figs. 8 and 9. This is because LightFS's advantage in batch reading metadata diminishes with fewer files, and larger file sizes increase the proportion of necessary operations, reducing overhead. Nonetheless, LightFS still performs as well as traditional methods in less favorable workloads. Performance improvements are most significant with shorter computation times, such as in KNN, where LightFS achieves gains of up to 6.48×, 1.74×, and 8.22× over frag-s, frag-m, and FSR. In longer computation tasks like Grep-ARM, improvements stabilize at 1.59×, 1.13×, and 1.87×. LightFS excels when files are smaller and more numerous, and computation times are shorter.

C. Latency Breakdown

To reveal the source of LightFS's performance improvement, we conducted a breakdown of latency, measuring the latency of each stage separately. We perform NDP operations on 100 files of 16 KB each, and the experimental results are shown in Figs. 10 and 11.

1) "Retrieve file layout" in frag-s and frag-m refers to the time taken for the host to retrieve layout through *filefrag*, while in FSR and LightFS, it refers to the time taken to read the file layout in storage.
2) "DMA" refers to the time taken by CSD to receive the file layouts or paths from the host. Due to the limited command length of NVMe [37], the host cannot just use reserved fields to transmit request parameters. Thus, DMA is needed for CSD to read the path or file layout from the host memory.
3) "Read file" time refers to the average time taken to read the data for each file, indicating the average interval of a file is read. Although the read latency of the flash page is long, concurrent reads can significantly reduce the average latency.
4) "Others" include task scheduling time, NVMe requests sending time, etc. The number on each bar represents

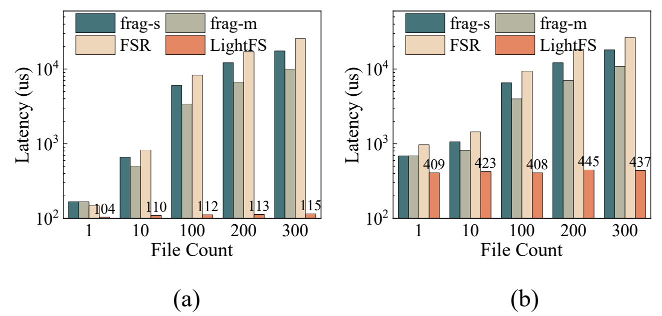


Fig. 12. Task offloading overhead under different file count (file size 16 KB). (a) With warmup. (b) Without warmup.

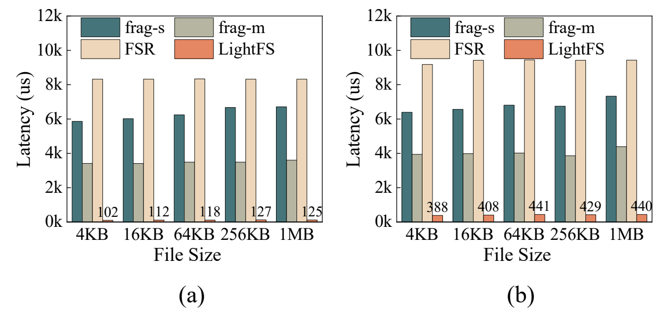


Fig. 13. Task offloading overhead under different file size (100 files). (a) With warmup. (b) Without warmup.

the PAO (i.e., the PAO in Section II-C), the lower the better.

First, LightFS almost only spends time on reading and computation, with PAO significantly lower than the other methods. This is because LightFS first can batch retrieve file layouts, thus the average time taken for retrieving the file layouts is very short. Second, because the CSD only needs one DMA to obtain the requested directory, the average DMA time for each file is very short. Additionally, LightFS does not require much host involvement, avoiding redundant data transfers and host task scheduling, resulting in others' time being short as well. When the system is not warmed up, compared to the warmed-up result, PAO for each method is increased. The main reason for the increment is primarily because the time to retrieve the file layout increases as it requires reading metadata from the flash.

LightFS improves performance by both reducing overhead and enhancing flash read speeds. Unlike frag-m, which improves read performance by using multiple processes but still faces delays before sending read requests, LightFS quickly sends many requests after batch retrieving layouts, making full use of flash bandwidth. As a result, LightFS reads a file in an average of 17us compared to frag-m's 22us. Applications with shorter computation times generally experience higher PAO, as fixed non-preemptive computation leaves more time wasted on unnecessary operations. For example, frag-m's PAO for KNN reaches 52.5%, while LightFS achieves a 1.75× acceleration by reducing this overhead.

D. Effect on Task Offloading

We evaluated LightFS performance improvements for the NDP task offloading. NDP task offloading refers to the latency of CSD in obtaining the layout of all the requested files. Hence, for the filefrag method, offloading overhead refers to the time taken for the host to retrieve the layout then send to CSD, and CSD to receive the data layout via DMA. For LightFS and FSR, offloading overhead refers to the time taken for the host to send the file path, and for CSD to directly read the file layout inside CSD.

Fig. 12 demonstrates the impact of file count on NDP task offloading overhead, showing that while LightFS's overhead remains nearly constant as it batch-fetches file layouts, the overhead for other methods grows linearly due to fetching and sending layouts individually; this is because LightFS only incurs additional time for reading layouts from the metadata cache after initially reading metadata from flash, with minimal impact on initialization time as the file count grows. Fig. 13 further shows that file size has little effect on the offloading overhead for LightFS, as its batch retrieval process minimizes the impact, leading to greater performance improvements over other methods, particularly with smaller files where layout retrieval overhead becomes more pronounced.

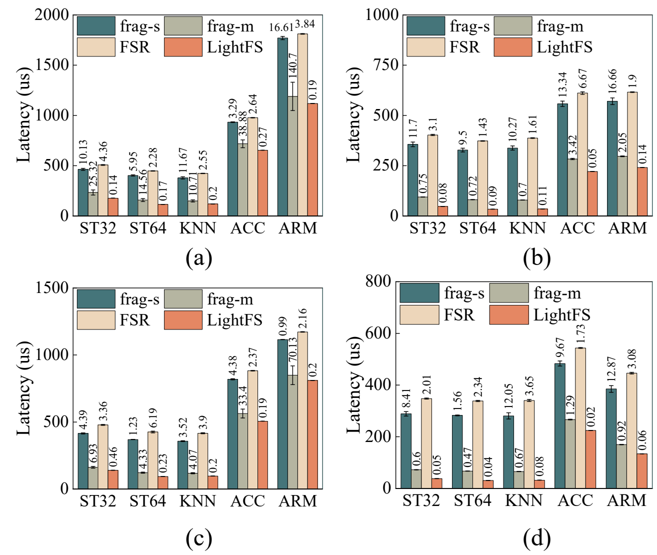
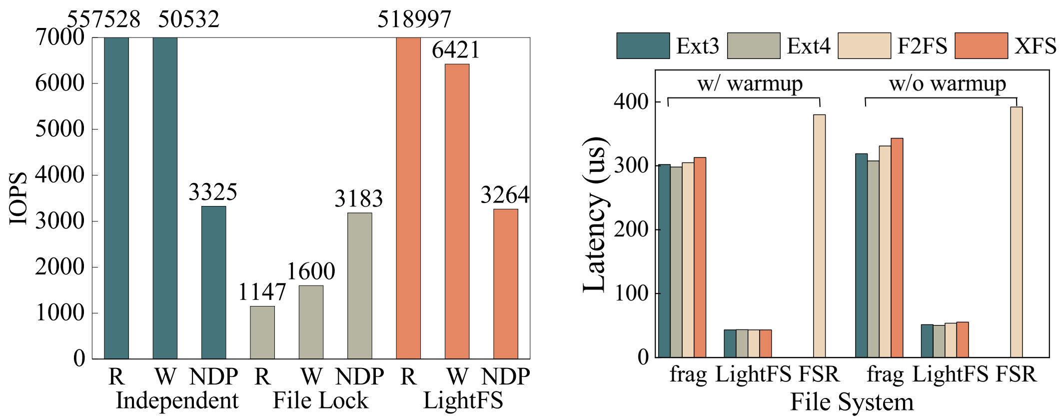


Fig. 14. Average latency comparisons on real datasets. (a) Hadoop. (b) Doppler. (c) Weather. (d) CIFAR-10.

E. Effect on Real-World Dataset

With the development of AI and IoT, the data is typically gathered and processed in the form of numerous small files. We deploy several real-world datasets in CSD, including images, system logs, and sensor data to show LightFS's performance in real-world scenarios. Information about these datasets is provided in Table IV. We perform the real-world operations outlined in Table III on these datasets, the results are shown in Fig. 14. The height of each column represents the average latency in processing each file, and the error bars indicate the standard deviation of multiple measurements. For clarity, we have labeled the standard deviations at the top of the bars.



(a) (b)

Fig. 15. (a) IOPS comparison between file locking and LightFS. (b) Performance comparison under different backend file systems.

TABLE IV
INFORMATION OF REAL-WORLD DATASETS

Dataset	Description	Average Size(B)
hadoop [21]	979 logs from a Hadoop cluster	49640
doppler [20]	17485 radar data captured with doppler radar	5287
weather [29]	1530 images of 5 kinds of weather conditions.	250977
cifar-10 [30]	50000 images of 10 classes	924

TABLE V
LATENCY OF FILEFRAG UNDER DIFFERENT BACKEND FILE SYSTEM

	With Warmup	Without Warmup
Ext3	$73us$	$960us$
Ext4	$50us$	$848us$
F2FS	$71us$	$859us$
XFS	$73us$	$729us$

LightFS significantly outperforms other methods, achieving up to 11.07× faster performance with an average boost of 3.66×. The largest gains are in STATS64 and KNN (5.35× and 5.13×), while Grep-ACC and Grep-ARM show smaller improvements (1.73× and 1.81×). This is because Grep's longer data processing time reduces the impact of NDP offloading, whereas shorter tasks benefit more from LightFS. Since CSDs are often used for simple data tasks, LightFS typically shows strong performance gains, especially with smaller files. For example, in the cifar-10 dataset, LightFS is 6.14×, 1.72×, and 7.33× faster than frag-s, frag-m, and FSR, respectively. In the weather dataset, with larger files, the gains are 2.73×, 1.17×, and 3.09×. LightFS performs better with smaller files because it eliminates more redundancy overhead.

LightFS is also more consistent, with the lowest standard deviation of 0.15us, compared to 8.41us, 18.03us, and 3.06us for frag-s, frag-m, and FSR. This stability comes from running LightFS-Host in user space and LightFS-CSD in the CSD's embedded environment, avoiding disruptions from kernel file systems and scheduling. Frag-m is less stable due to multi-threading, which causes kernel preemption and PCIe contention. LightFS avoids these issues by handling concurrency without multi-threading and reduces overhead by not sending NDP requests for each file.

F. Effect on Hybrid Request

We evaluated the impact of file lock on the performance of CSD. We use three processes to read, write, and send STATS32 NDP requests to a 16 KB file at the same time, and statistics the I/O per second (IOPS) of each operation. We lock the file by *flock* function during the file operation, to achieve the multithreading situations in the other methods [10], [11]. We also show the performance of each process executing independently to demonstrate the performance upper bounds.

The experimental results in Fig. 15 show IOPS improvements for read (452.5×), write (4×), and NDP (1.03×) requests. LightFS achieves high read IOPS by avoiding request blocking, thus nearly approaching performance limits. Although write requests incur some performance loss due to the metadata locker notifications, they still surpass the file locking methods. NDP request performance is similar across methods, as NDP requests take much longer time to execute, and holding the lock longer leads to minimal impact on the overall I/O performance. Thus, the main performance loss with file locking is for the I/O requests, with minimal impact on the NDP requests.

G. Effect on Different Backend File Systems

To demonstrate LightFS's adaptability to various backend file systems, we evaluate the performance of task offloading methods on widely used kernel file systems, including Ext3, Ext4, F2FS, and XFS [10], [11], [12], [38]. Given FSR's exclusive compatibility with F2FS, we report its results only for that file system. We measure the average latency for STATS64 tasks performed on 100 16KB files under each system, as shown in Fig. 15(b).

The performance disparities across file systems are due to different overheads in retrieving file layouts. Ext4 exhibits the lowest overhead, as shown in Table V, leading to the best performance. If FSR supported Ext4, its performance could improve, though it remains inferior to LightFS due to its traditional architecture. Once metadata is cached, LightFS's performance stabilizes across file systems, unaffected by backend differences during task offloading. However, filefrag-based methods reveal variations, with XFS increasing latency by 5% compared to Ext4. Performance disparities arise during initialization when LightFS retrieves the layout of *.ndpmeta*.

V. Conclusion

This article reveals the task offloading overhead and inconsistency problems in CSDs. To solve these problems, we propose LightFS, a lightweight user-space file system for CSDs. To reduce offloading overhead, LightFS retrieves the file layouts in storage to avoid redundant I/Os. To ensure consistency, LightFS employs a metadata locker and an update synchronizer to prevent the partial file updates. By evaluating LightFS on a real-world testbed, LightFS shows significant performance improvements in real-world applications.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments and improvements to this article.

References

[1] D. R.-J. G.-J. Rydning, *The Digitization of the World from Edge to Core*, Int. Data Corp., Framingham, MA, USA, 2018, p. 16.

[2] C. Zou and A. A. Chien, "ASSASIN: Architecture support for stream computing to accelerate computational storage," in *Proc. 55th IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2022, pp. 354–368.

[3] A. Barbalace and J. Do, "Computational storage: Where are we today?" in *Proc. CIDR*, 2021, pp. 1–7.

[4] W. Cao et al., "POLARDB meets computational storage: Efficiently support analytical workloads in cloud-native relational database," in *Proc. 18th USENIX conf. FAST*, 2020, pp. 29–41.

[5] J. Do, Y. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt, "Query processing on smart SSDs: Opportunities and challenges," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2013, pp. 1221–1230.

[6] K. Huang, Z. Shen, Z. Shao, T. Zhang, and F. Chen, "Breathing new life into an old tree: Resolving logging dilemma of B+-tree on modern computational storage drives," *Proc. VLDB Endowm.*, vol. 17, no. 2, pp. 134–147, 2023.

[7] M. Wilkening et al., "RecSSD: Near data processing for solid state drive based recommendation inference," in *Proc. 26th ACM Int. Conf. ASPLOS*, 2021, pp. 717–729.

[8] C. Li, Y. Wang, C. Liu, S. Liang, H. Li, and X. Li, "GLIST: Towards in-storage graph learning," in *Proc. USENIX ATC*, 2021, pp. 225–238.

[9] A. HeydariGorji, M. Torabzadehkashi, S. Rezaei, H. Bobarshad, V. Alves, and P. H. Chou, "Stannis: Low-power acceleration of DNN training using computational storage devices," in *Proc. 57th ACM/IEEE DAC*, 2020, pp. 1–6.

[10] Z. Yang et al., "lambda-IO: A unified IO stack for computational storage," in *Proc. 21st USENIX Conf. FAST*, 2023, pp. 347–362.

[11] Z. Ruan, T. He, and J. Cong, "INSIDER: Designing in-storage computing system for emerging high-performance drive," in *Proc. USENIX ATC*, 2019, pp. 379–394.

[12] L. Li et al., "Optimizing the performance of NDP operations by retrieving file semantics in storage," in *Proc. ACM/IEEE DAC*, 2023, pp. 1–6.

[13] J. Zhang, Y. Ren, and S. Kannan, "FusionFS: Fusing I/O operations using CISCOps in firmware file systems," in *Proc. 20th USENIX Conf. FAST*, 2022, pp. 297–312.

[14] Z. Shen, F. Chen, G. Yadgar, Z. Jia, and Z. Shao, "Prism-SSD: A flexible storage interface for SSDs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 4, pp. 882–896, Apr. 2022.

[15] I. F. Adams, J. Keys, and M. P. Mesnier, "Respecting the block interface–computational storage using virtual objects," in *Proc. 11th USENIX Workshop Hot Topics Storage File Syst. (HotStorage)*, 2019, pp. 1–7.

[16] A. Barbalace, M. Decky, J. Picorel, and P. Bhatotia, "BlockNDP: Block-storage near data processing," in *Proc. 21st Int. Middleware Conf. Ind. Track (Middleware)*, 2020, pp. 8–15.

[17] J. Zhang, Y. Ren, M. Nguyen, C. Min, and S. Kannan, "OmniCache: Collaborative caching for near-storage accelerators," in *Proc. 22nd USENIX Conf. FAST*, 2024, pp. 35–50.

[18] S. Kannan, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Y. Wang, J. Xu, and G. Palani, "Designing a true direct-access file system with DevFS," in *Proc. 16th USENIX Conf. FAST*, 2018, pp. 241–256.

[19] Z. J. Wang, E. Montoya, D. Munechika, H. Yang, B. Hoover, and D. H. Chau, "DiffusionDB: A large-scale prompt gallery dataset for text-to-image generative models," 2023, *arXiv:2210.14896*.

[20] I. Roldan et al., "DopplerNet: A convolutional neural network for recognising targets in real scenarios using a persistent range–doppler radar," *IET Radar, Sonar Navig.*, vol. 14, no. 4, pp. 593–600, 2020.

[21] J. Zhu, S. He, P. He, J. Liu, and M. R. Lyu, "Loghub: A large collection of system log datasets for ai-driven log analytics," in *Proc. IEEE 34th ISSRE*, 2023, pp. 355–366.

[22] X. Liao, Y. Lu, E. Xu, and J. Shu, "Write dependency disentanglement with HORAE," in *Proc. 14th USENIX Symp. OSDI*, 2020, pp. 549–565.

[23] C. Whitaker, S. Sundar, B. Harris, and N. Altiparmak, "Do we still need IO schedulers for low-latency disks?" in *Proc. 15th ACM Workshop Hot Topics Storage File Syst. (HotStorage)*, 2023, pp. 44–50.

[24] J. Kwak, S. Lee, K. Park, J. Jeong, and Y. H. Song, "Cosmos+ OpenSSD: Rapid prototype for flash storage systems," *ACM Trans. Storage*, vol. 16, no. 3, pp. 1–35, 2020.

[25] J. Li et al., "Horae: A hybrid I/O request scheduling technique for near-data processing-based SSD," *IEEE Trans. Comput.-Aided Design Integrated Circuits Syst.*, vol. 41, no. 11, pp. 3803–3813, Nov. 2022.

[26] S. Liang, Y. Wang, Y. Lu, Z. Yang, H. Li, and X. Li, "Cognitive SSD: A deep learning engine for in-storage data retrieval," in *Proc. USENIX ATC*, 2019, pp. 395–410.

[27] C. Ma et al., "Rebirth-FTL: Lifetime optimization via approximate storage for NAND flash memory," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 10, pp. 3276–3289, Oct. 2022.

[28] J. Sun, S. Li, Y. Sun, C. Sun, D. Vucinic, and J. Huang, "LeaFTL: A learning-based flash translation layer for solid-state drives," in *Proc. 28th ACM ASPLOS*, 2023, pp. 442–456.

[29] V. Gupta, "Weather classification," Dataset. Accessed: Mar. 28, 2024. [Online]. Available: https://www.kaggle.com/datasets/vijaygiitk/multiclass-weather-dataset

[30] A. Krizhevsky, *Learning Multiple Layers of Features From Tiny Images*, Univ. Toronto, Toronto, ON, Canada, 2009.

[31] L. Li. "Repo of FSR." 2023. [Online]. Available: https://github.com/ll26571/FSR

[32] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *Proc. 13th USENIX Conf. FAST*, 2015, pp. 273–286.

[33] Z. Ruan, T. He, and J. Cong, "Analyzing and modeling in-storage computing workloads on EISC—An FPGA-based system-level emulation platform," in *Proc. IEEE/ACM ICCAD*, 2019, pp. 1–8.

[34] "Cosmos-plus-openssd." GitHub. Accessed: Aug. 23, 2024. [Online]. Available: https://github.com/Cosmos-OpenSSD/Cosmos-plus-OpenSSD

[35] "UNVMe—A user space NVMe driver project." unvme. 2016. [Online]. Available: https://github.com/zenglg/unvme

[36] S. Pei, J. Yang, and Q. Yang, "REGISTOR: A platform for unstructured data processing inside SSD storage," *ACM Trans. Storage*, vol. 15, no. 1, pp. 1–24, 2019.

[37] (NVM Express Inc., Beaverton, OR, USA). *NVM Express Specifications*. Accessed: Mar. 24, 2024. [Online]. Available: https://nvmexpress.org/specifications/

[38] M. Kwon, D. Gouk, S. Lee, and M. Jung, "Hardware/software co-programmable framework for computational SSDs to accelerate deep learning service on large-scale graphs," in *Proc. 20th USENIX Conf. FAST*, 2022, pp. 147–164.