Optimizing the Performance of NDP Operations by Retrieving File Semantics in Storage

Lin Li, Xianzhang Chen*, Jiali Li, Jiapin Wang, Duo Liu, Yujuan Tan, Ao Ren
College of Computer Science, Chongqing University, Chongqing, China
{linli265, xzchen, lijiali, wjp, liuduo, tanyujuan, ren.ao}@cqu.edu.cn

2023 60th ACM/IEEE Design Automation Conference (DAC) | 979-8-3503-2348-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/DAC56929.2023.10247986

***Abstract*—In-storage Near-Data Processing (NDP) architectures can reduce data movement between the host and the storage device by offloading computing tasks to the storage. This encourages many studies on building NDP applications, such as recommendation systems and databases, on computational SSDs. However, in the data path of existing NDP architectures, an NDP application has to find out the address of the requested file data by calling the I/O stacks of the kernel on the host, which incurs large overhead for transferring data between the host and the computational SSD. In this paper, we present File Semantics Retriever (FSR) to optimize the data path of NDP architectures by locating and fetching the requested file data directly in the computational SSD. The key idea is to recognize the file system layout and the metadata structures in the storage with the collaboration of a user-space library and a handler in the firmware of the computational SSD. We implement a prototype of FSR and evaluate it on the Cosmos plus OpenSSD, a widely-used computational SSD platform. The experimental results show that FSR outperforms existing NDP architectures in both benchmarks and real-world NDP applications.**

***Index Terms*—Near-data processing, computational storage**

I. Introduction

The rapidly growing volume of data on the embedded devices, such as autonomous vehicles and robotics, poses higher challenge on the efficiency and energy consumption of data processing. Recently, many studies propose in-storage Near-Data Processing (NDP) architectures [1]–[7] to mitigate the huge overhead of data movement between the host and the storage device. Generally, these works focus on designing a computational SSD that processes data inside the storage to avoid data movements for specific applications, such as recommendation systems [1], [2], deep learning [5], [6], and unstructured data processings [3], [4], using NDP accelerators inside the storage.

Nevertheless, from the perspective of the whole data path of an NDP operation, many existing NDP architectures [2]–[4] still have large overhead on data movement between the host and the computational SSD. In these NDP architectures, the data management of the computational SSD itself relies on a file system that runs on the host, while the in-storage NDP accelerators are unaware of the file system layout. Consequently, before executing the NDP tasks in the computational SSD, the NDP applications still need to locate the requested file data through the heavy I/O stacks in the kernel of the host, which may cost 84.6% execution time of the whole data path of an NDP operation.

In this paper, we aim to optimize the performance of NDP operations by mitigating the data movement overhead for locating the requested file data. We present *File Semantics Retriever (FSR)* to offload the whole data path of NDP operations by retrieving the file semantics, like file path and metadata, inside the storage. FSR consists of a user-space library called FSRLib and an in-storage module, FSR handler. Different from existing NDP architectures, an NDP application with FSR only needs to send the path of the requested file and the parameters of the NDP operation to FSR handler. FSR handler will locate and feed the requested file data to the NDP module directly in storage.

We implement a prototype of FSR based on Linux and Cosmos plus OpenSSD [8]. We evaluate the performance of FSR and compare it with the host-based data path of existing NDP architectures. The experimental results show that FSR can reduce the latency of locating the requested file data by 83.94% over existing NDP architecture. For real-world NDP applications, the FSR-based NDP operations outperform the host-based NDP operations on all workloads. Compared with the host-based NDP operations, the FSR-based NDP operations reduce the execution time by 51.96% on 16KB files.

In summary, the contributions of this paper include:

- The insight on the data path of NDP architecture through typical NDP operations.
- A new NDP data path, File Semantics Retriever (FSR) that supports direct data locating and fetching inside the computational SSDs.
- A prototype of FSR and a detailed comparative evaluation with the existing host-based NDP data path using micro-benchmarks and real-work NDP applications.

The rest of this paper is organized as follows. In Section II, we introduce the data path of existing NDP architectures and show a motivational example. Section III presents the design and implementation of FSR. Section IV evaluates the proposed FSR. Section V concludes the paper.

II. Background and Motivation

A. File Semantics in NDP Operation

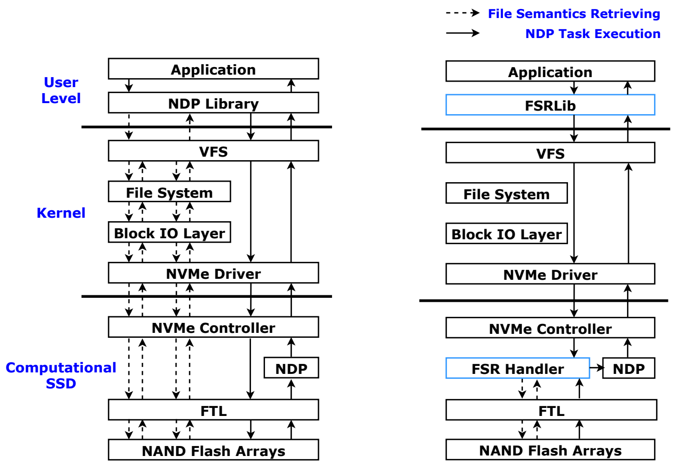
In the namespace of an inode-based file system, such as ReconFS [9] and F2FS [10], the data of a directory file is

This work is partially supported by National Natural Science Foundation of China (Project No. 62072059 and No. 62102051), Chongqing Post-doctoral Science Foundation (Project No. 2021LY75), and the Funds for Chongqing Distinguished Young Scholars (Project No. cstc2020jcyj-jqX0012).

*Xianzhang Chen is the corresponding author.

organized as a table. Each entry in the table points to a sub-directory or sub-file in the directory. Through this recursive organization, all the data files and directory files are indexed in a huge tree structure [9], [11]. To simplify the access to namespace for users and applications, each file in the tree can be mapped to a path in the form of a string. Each file has a unique inode to maintain its attributes (e.g., file size and modification time) and the indexes of all the data blocks in the storage device.

To optimize the data path of NDP operations, we use *file semantics* to represent the metadata related to a file, including the path in the namespace, the attributes and the indexes recorded in the inode. To accommodate the read/write granularity of storage devices, a file system encapsulates semantics and data of each file in fixed-size data blocks. The metadata of the file system, such as the layout descriptor and bitmap, is also encapsulated in blocks and maintained in fixed place of the storage [12].



(a) Existing data path (b) FSR-based data path

Fig. 1: Overview of data paths: (a) Existing computational SSDs; (b) The proposed FSR-based computational SSDs (Section III).

B. Data Path of Existing NDP Architectures

Generally, the computational SSD is managed by a host-side file system whereas the file semantics and the metadata of the file system are reserved in the computational SSD. In the data path of an NDP operation, it is necessary to retrieve the file semantics to find out the location of the requested data before executing the NDP task.

Fig. 1(a) shows the data path of an NDP operation in most of existing NDP architectures [2], [3]. The data path has two parts: *file semantics retrieving* and *NDP task execution*. In file semantics retrieving, the NDP library uses system calls to retrieve the file semantics of the requested files, and the kernel's Virtual File System (VFS) traverses the namespace tree following the file path. In NDP task execution, the NDP library packages it with the parameters of the NDP operation and then sends them to the computational SSD. However, this kind of data path may incur large overhead for repeatedly transferring the related directory files between the host and the computational SSD.

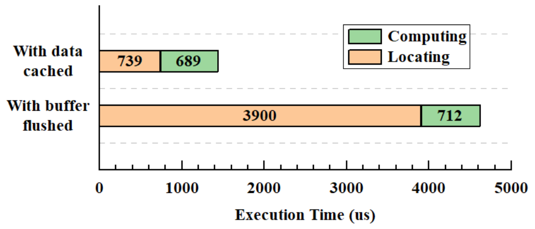


Fig. 2: The execution time of an NDP-based string search.

We give a motivational example to demonstrate the overhead of data path in existing NDP architectures. Fig. 2 shows the execution time of an NDP operation of string search [3]. In the NDP operation, locating the requested file data takes 739us (cached in the SSD DRAM buffer) or 3900us (needs to read data from the Flash chips). By contrast, the computing process just takes around 700us. Thus, if we can retrieve the file semantics of the requested file directly in the computational SSD, we may achieve more than 48% performance improvement for NDP operations.

III. Design and Implementation

A. Overview of File Semantics Retriever

In this paper, we present a File Semantics Retriever (FSR) for computational SSDs to realize and fetch the required data of the in-storage NDP operations. Fig. 1(b) shows the architecture of FSR. FSR is comprised of an in-storage module (FSR handler) and a host-side library (FSRLib).

The key to directly access the file data in the storage is to understand the semantics of the file system. In a computational SSD, the Flash Translation Layer (FTL) is responsible for handling the mapping from the logical address space used by the file system to the physical address space. Hence, to correctly retrieve the file semantics in the namespace of a file system, FSR handler needs to be in the same address space of the file system. On the other side, FSR handler needs to track and parse the I/O requests from the host and resolve the corresponding data blocks in the storage. Therefore, we put FSR handler in between the NVMe controller and FTL in the computational SSD, as shown in Fig. 1 (b).

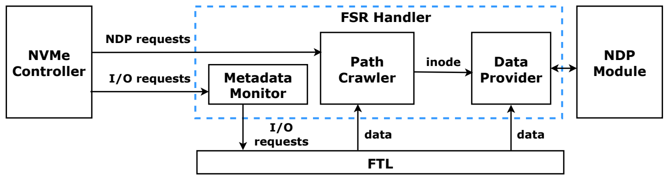


Fig. 3: The architecture of the FSR handler.

FSR handler is designed to be a lightweight plug-in component for the firmware of SSDs. Fig. 3 shows the architecture of FSR handler. FSR handler listens and handles the related requests with three major components, i.e., metadata monitor, path crawler, and data provider.

Basically, there are two types of requests, i.e, NDP requests and "normal" I/O requests (e.g., read and write), upon a computational SSD [13]. Given an NDP request, the *path crawler* transfers the file path in the hierarchical namespace of the corresponding file system to locate the inode of the targeting file (Section III-B). Then, the *data provider* extracts the required file data for the NDP operation according to the inode of the targeting file (Section III-C). For normal I/O requests, the *metadata monitor* captures the latest metadata written back by the file system and uses it to keep consistent with the host (Section III-D). FSR handler is embedded in the firmware of the computational SSD in the form of function calls.

The FSRLib on the host has three responsibilities, including packaging the parameters of NDP tasks, constructing the FSR-specific commands, and sending the commands to the computational SSD. With FSRLib, user processes can complete all the data fetching and near-data processing just in one command to the computational SSD. As a consequence, our solution avoids the overhead for transferring data between the host and the computational SSD in NDP tasks.

B. Path Crawler

The hierarchical file systems generally construct their namespaces as a tree of directory files. The data of a directory file mainly contains a "dentry table" where each entry in the table records the inode number and file name of a child file. To locate the file requested by an NDP operation, the path crawler of FSR handler needs to recursively traverse the tree of directory files following the corresponding file path of the requested file.

The path crawler first reads the inode of the directory from Flash chips, and then imitates the operation process of the file system to find the entry in the directory. For instance, suppose the computational SSD is formatted as F2FS [10], the path crawler first determines whether the inode carries inline data by comparing the relevant flags in the inode. If true, the path crawler treats the inline data as directory entries. Otherwise, the path crawler parses the addresses of the data blocks and reads them from Flash chips. Then, the path crawler uses the same hash function as F2FS [10] to calculate the hash value of the requested file's name, and compares it with the hash values in the dentry table to find the entry pointing to the requested file. The path crawler recursively repeats these procedures until it reaches the inode of the requested file or returns an error in case the path is invalid.

The path crawler is also responsible for controlling the access permission of the requested files. When a user process commits an NDP operation, FSRLib sends the *uid* and *gid* of the user process along with the requested file path to the FSR handler. When the path crawler accesses an inode, it first examines the class of the user process by comparing the *uid* and *gid* maintained in the inode with these delivered by FSRLib. Then, the path crawler determines the permissions of the user process to the file according to the information recorded in the inode.

C. Data Provider

The data provider is responsible for in-storage data provisioning for the NDP module. On one hand, the data provider abstracts the parsing process of the metadata of files and encapsulates them into a set of interfaces. With these interfaces, the NDP module can obtain the required file semantics, e.g., file size and the addresses of data blocks, without knowing the detailed structures of the file system. Thus, the NDP module is decoupled with the file system using the abstractions provided by the data provider. As a result, NDP operations are easy to deploy on different computational SSDs.

On the other hand, the data provider is responsible for the data provisioning of the NDP module. Since the inode records the indexes of data blocks of a file, the data provider can reach all addresses of the data blocks via inode. In order to meet the data requirements of NDP operations in different scenarios, data provider mainly provides data in two different ways.

First, for the scenarios where the NDP module does not have strict requirements on the processing orders of data blocks, the data provider parses out the addresses of the required blocks and sends requests to FTL in batches with a *NDP_requested* mark. Each time a marked data block is read into the buffer of the data provider, FTL informs the NDP module to process the data via a function pointer. Hence, the reading of data blocks performs in parallel to maximize the internal bandwidth of the computational SSD.

Second, when the NDP module only needs a certain piece of data of the file, the data provider calculates the corresponding data blocks according to the incoming logical address and length, and reads the required data from the Flash to a continuous memory space for the NDP module.

D. Metadata Monitor

Based on the modern memory hierarchy, the VFS generally maintains a set of run-time data structures for the files, such as the inode cache and the directory cache. Meanwhile, the FSR handler retrieves file semantics for NDP operations using the file system metadata persisted on the computational SSD. Basically, if the FSR handler fetches the related file system metadata from the Flash chips each time an NDP operation is executed on the NDP module, there will be no consistency issues. However, the price is the degradation of performance since reading data from the Flash chips is expensive.

To improve the performance of file semantics retrieving, FSR handler adopts the in-storage DRAM to cache the commonly-used metadata, such as the information of the root inode. In this case, FSR handler should ensure that the metadata in the DRAM is consistent with the one persistent on the Flash chips. However, the kernel on the host does not explicitly notify the computational SSD when the file system metadata is updated, resulting in risks of inconsistency for the FSR handler.

Hence, we present the metadata monitor to handle the consistency between the running file semantics on the host and the required file semantics in the computational SSD. Generally, the file systems pack the latest metadata and write it back to the corresponding region on the storage when certain conditions (e.g., *fsync()* system call) is triggered. The metadata monitor tracks such behaviors in the I/O path of the computational SSD. Fig. 4 shows the workflow of metadata monitor. When the kernel thread on the host initiates a *down_write* request to the computational SSD, it first goes through the metadata monitor before moving to FTL. If the block address carried in the request hits the region of the metadata, it means that the carried data is the latest file system metadata. Then, the metadata monitor updates the cached metadata with the carried data.

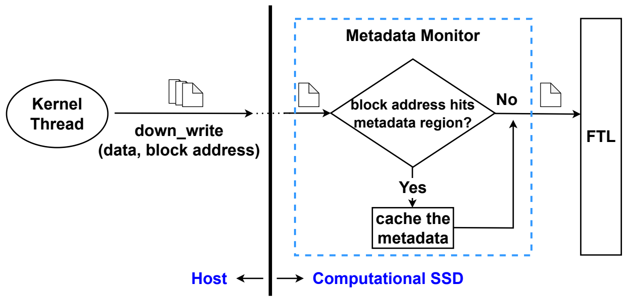


Fig. 4: The workflow of metadata monitor.

To correctly capture the regions of metadata, the FSR first locates the superblock of the file system, which is generally maintained in fixed locations of a file system. Then, the metadata monitor obtains the addresses of other metadata areas by parsing the superblock of the file system. The metadata monitor can address these information during launch time once for all, thereby will not affect the NDP operations. Moreover, the latency of metadata monitor is less than $1us$ (common case) or $60us$ (needs update of metadata cache) on the Cosmos platform, which is insignificant for most write I/Os (more than $610us$ [8]).

E. Limitation and Discussion

Although FSR tries to provide the same functionality as kernel-based NDP operations, there are limitations that FSR cannot bypass easily.

File path. The path crawler traverse the hierarchical namespace to locate the requested file of NDP operations. However, the run-time environment in computational SSD does not support the full run-time states as VFS does [14]. Thus, FSR only support absolute file paths that start from the root of the namespace.

Supported file systems. To fetch file data in storage, FSR handler should be aware of the data structures of the file system metadata since FSR needs to resolve the inodes and the file indexes of the requested file. However, different file systems may employ different metadata structures. FSR abstracts a set of common data structures and parsing routing to support commonly-used file systems on SSDs, such as EXT4 [15] and F2FS. The support of other file systems, such as FAT [16], needs further engineering works.

F. Implementation

We implement a prototype of FSR based on Linux and Cosmos plus OpenSSD [8]. FSRLib and FSR handler are written in *C* language with 100 lines and about 1K lines of code, respectively. FSRLib prepares the configurations of NDPs, constructs the FSR-specific command for FSR handler, and sends the command to the computational SSD via *ioctl()* interface. To avoid the scheduling overhead of the block layer in the kernel, we use the "Get Features" command in the NVMe standard version 1.2a [17] to implement the FSR-specific command. FSRLib expands a new "feature id" such that the computational SSD can recognize the command.

FSR handler is implemented as a set of function calls that can be embedded to the firmware of computational SSDs. When the computational SSD receives a FSR-specific command, it notifies the NVMe controller to fetch the corresponding configurations of the NDP operation from the host. The NDP module chooses the data supply mode of the data provider according to the NDP operations' demand and processes the operation.

In the current implementation, the path crawler works in blocking manner since the firmware we use runs in single-threaded mode on only one core of the processor, even though FSR handler supports multiple threads.

IV. Evaluation

A. Experimental Setups

To evaluate the effectiveness of FSR, we embed FSR handler into the firmware of Cosmos plus OpenSSD [8]. The OpenSSD is composed of a Xilinx Zynq-7000 AP SOC (XC7Z045), 1GB DRAM, 1TB eight-channel and eight-way NAND Flash array, and a PCIe Gen2 8-lane interface. The firmware of the OpenSSD runs on the Dual 1GHz ARM Cortex-A9 processor of XC7Z045. To complete the evaluation system, we use a PC equipped with Intel Core i5-4460 CPU@3.20GHz and 16GB DRAM as the host. We deploy FSRLib on the host along with Ubuntu 16.04 using Linux kernel 4.4.4. The OpenSSD storage is formatted by F2FS [10].

To comprehensively present the characteristics of FSR, we first evaluate and analyze the performance of FSR with micro-benchmarks. Then, we run real-work NDP applications on the FSR-based computational SSD to reveal the advantages and disadvantages of FSR. We compare the performance of FSR to the data path based on the kernel I/O stacks of the host, denoted by "Host-based", which is widely-used by existing NDP applications [2], [3]. Moreover, we also consider the impact of the FTL buffer on performance. We conduct all operations on two ways, i.e., buffer hit and buffer miss. For the buffer-hit operations, we repeat each one several times to ensure that the required file data is cached in the FTL buffer. For the buffer-miss operations, we flush the FTL buffer by filling it with unrelated data before committing the operation.

B. Latency Breakdown of Path Crawler

Among the components of FSR, the path crawler is the major one that affects the performance of NDP operations. Thus, we take a breakdown for the path crawler to reveal the key factors affecting the performance. Table I shows the latency breakdown of the path crawler when the FTL buffer is hit. The deeper the path is, the more directory files the path crawler needs to parse. Thus, the latency of retrieving *Node Address Table (NAT)* of F2FS (denoted by "Read NAT") and finding the related directory entry (denoted by "Find entry") grows linearly. When the FTL buffer is missed (Table II), all the data blocks of the directory files and NAT need to be load from the Flash chips (denoted by "Read directory"), resulting in rapid growth of latency. Note that the latency of Read NAT does not increase much. It is because only the first read of NAT occurs on the Flash while the rest of reads happens on the cached NAT.

TABLE I: Latency breakdown of path crawler when buffer hit.

Path depth	Read directory data	Read NAT	Finding dentry	Total latency
5	7us	9us	39us	130us
10	12us	15us	67us	227us
20	23us	39us	125us	419us

TABLE II: Latency breakdown of path crawler when buffer miss.

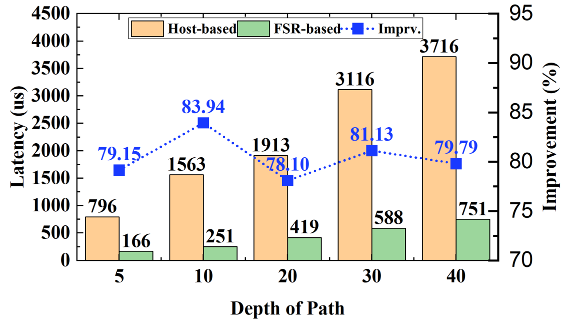
Path depth	Read directory data	Read NAT	Finding dentry	Total latency
5	2951us	255us	40us	3321us
10	4933us	262us	67us	5395us
20	8895us	287us	128us	9543us

C. Effect on Locating Files

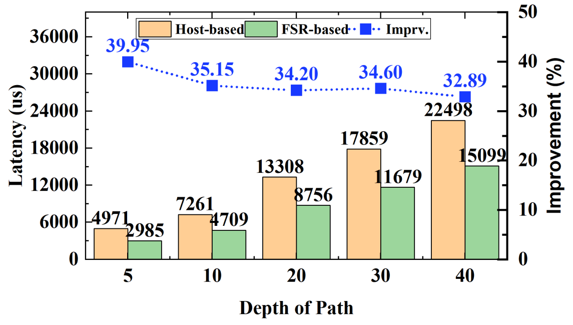
Here, we evaluate the performance of locating the requested file by resolving the file path. We use the widely-used benchmark, Filebench [18], to generate the dataset. In the dataset, the maximum directory depth is set to 40 and each directory contains 50 to 60 sub-directories or sub-files. Then, we devise a workload to access the dataset using FSR and the host-based approach with different configurations of directory depth.

Fig. 5 shows the execution time of locating the requested file data by the FSR-based and the Host-based approaches as the depth of file paths increases. The execution time of both the tested approaches scales with the increase of the path depth. The FSR-based approach outperforms the host-based approach on all the workloads, including the buffer-hit workloads (Fig. 5(a)) and the buffer-miss workloads (Fig. 5(b)) . This is because the FSR handler directly traverses the file path in the computational SSD and avoids the overhead for transferring data between the host and the computational SSD.

The experimental results show that the FTL buffer has a significant impact on performance. On the buffer-hit workloads, the tested approaches have almost zero overhead for reading data from the Flash chips. In this case, the FSR-based approach reaches 78.1% to 83.94% faster than the Host-based approach. On the contrary, Fig. 5(b) shows that the execution time of both



(a) With FTL data buffer hit



(b) With FTL data buffer miss

Fig. 5: Performance comparison of locating the requested file.

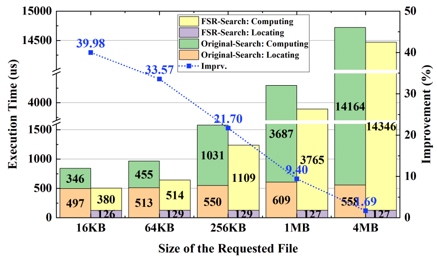
the FSR-based approach and the Host-based approach increase significantly for reading data from the Flash chips. However, the FSR-based approach still saves 32.89-39.95% execution time in comparison with the Host-based approach.

D. Effect on Real-world NDP Operations

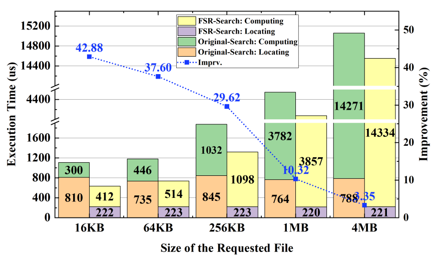
In this subsection, we evaluate the performance gain that FSR brings to real-world NDP applications. Since existing works [2]–[4], such as REGISTOR [3], are not open-sourced, we implement the NDP operation "string search" on Cosmos platform according their papers. The workflow of the string search operation has two major steps. The first step is to figure out where the data blocks of the requested file are distributed in the Flash chips (marked as "Locating"). The second step is to load those data blocks from the Flash chips and perform string matching (marked as "Computing"). We evaluate FSR-based string search and the Host-based string search with different sizes of requested files and path depth.

Fig. 6 shows the execution time of "Original-Search" and "FSR-Search" with different configurations of buffer hit/miss, path depth, and requested file size. Generally, FSR-based string search achieves better performance on all workloads. For example, Fig. 6(c) shows that FSR-Search outperforms Original-Search by 6.95%-51.96% when the buffer is hit and the path depth is 20.

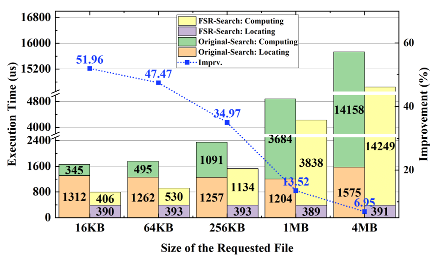
In the buffer-hit workloads (Fig. 6(a)(b)(c)), the performance gains of FSR-Search increases with the growth of path depth. It is because that the locating time of Original-



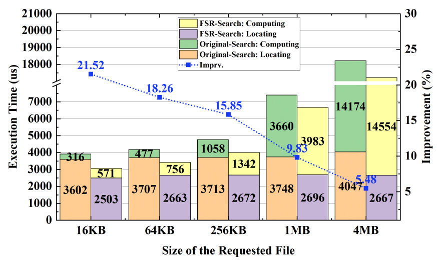
(a) With FTL buffer hit; path depth is 5.



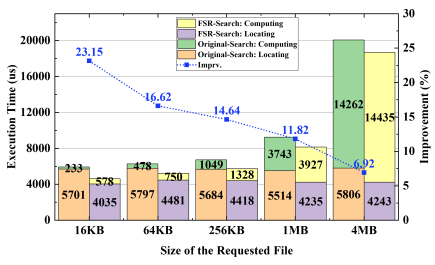
(b) With FTL buffer hit; path depth is 10.



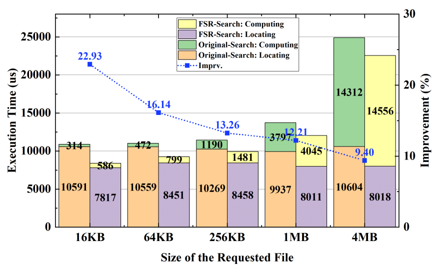
(c) With FTL buffer hit; path depth is 20.



(d) With FTL buffer hit miss; path depth is 5.



(e) With FTL buffer miss; path depth is 10.



(f) With FTL buffer miss; path depth is 20.

Fig. 6: Performance comparison on real-world NDP application workloads.

Search grows faster than that of FSR-Search with the increase of path depth. With the same path depth and requested file size, the performance gains of FSR-Search on the buffer-hit workloads (Fig. 6(a)(b)(c)) are higher than these on the buffer-miss workloads (Fig. 6(d)(e)(f)). It is because reading the directory files from the Flash chips dilutes the computing advantages of FSR. However, FSR-Search still gains 5.48% to 23.15% performance improvement over Original-Search on the buffer-miss workloads.

It is noteworthy that with the increase of the size of the requested file, the performance gains brought by FSR decreases. As Fig. 6 shows, the performance gains of FSR comes from the in-storage data locating. With the increase of requested file size, the computing step gradually dominates the execution time of the NDP operation. When the requested file size exceeds 4MB, the performance improvement of FSR over existing NDP architectures is less than 10%. Hence, it is better to use FSR for the NDP operations on small files.

V. Conclusion

This paper analyzes the overhead of the redundant file path in exiting NDP operations. To mitigate the overhead, the paper proposes FSR to offload the retrieve of file semantics into the computational SSD. We implement the user library and in-storage handler of FSR on a real platform to evaluate the effectiveness of FSR. Our evaluation shows that the proposed FSR outperforms existing solutions in both micro-benchmarks and real-world NDP applications.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments and improvements to this paper.

References

[1] M. Wilkening, U. Gupta, and S. e. a. Hsia, "Recssd: near data processing for solid state drive based recommendation inference," in *ASPLOS*, 2021, pp. 717–729.
[2] X. Sun, H. Wan, and Q. e. a. Li, "Rm-ssd: In-storage computing for large-scale recommendation inference," in *HPCA*, 2022, pp. 1056–1070.
[3] S. Pei, J. Yang, and Q. Yang, "Registor: A platform for unstructured data processing inside ssd storage," *ToS*, vol. 15, no. 1, pp. 1–24, 2019.
[4] I. F. Adams, J. Keys, and M. P. Mesnier, "Respecting the block interface–computational storage using virtual objects," in *HotStorage*, 2019.
[5] S. Liang, Y. Wang, and Y. L. et al, "Cognitive ssd: A deep learning engine for in-storage data retrieval," in *ATC*, Jul. 2019, pp. 395–410.
[6] S. Liang, Y. Wang, and C. e. a. Liu, "Ins-dla: An in-ssd deep learning accelerator for near-data processing," in *FPL*, 2019, pp. 173–179.
[7] J.-H. Park, S. Choi, and G. e. a. Oh, "Sas: Ssd as sql database system," *Proceedings of the VLDB Endowment*, vol. 14, no. 9, pp. 1481–1488, 2021.
[8] J. Kwak and L. et al, "Cosmos+ openssd: Rapid prototype for flash storage systems," *ToS*, vol. 16, no. 3, pp. 1–35, 2020.
[9] Y. Lu, J. Shu, and W. Wang, "Reconfs: A reconstructable file system on flash storage," in *FAST*, 2014, pp. 75–88.
[10] C. Lee, D. Sim, and J. e. a. Hwang, "F2fs: A new file system for flash storage," in *FAST*, 2015, pp. 273–286.
[11] M. Cai, J. Shen, and B. e. a. Tang, "Flatfs: Flatten hierarchical file system namespace on non-volatile memories," in *ATC*, 2022, pp. 899–914.
[12] J. Chen, Q. Wei, and C. e. a. Chen, "Fsmac: A file system metadata accelerator with non-volatile memory," in *MSST*, 2013, pp. 1–11.
[13] J. Li, X. Chen, and D. e. a. Liu, "Horae: A hybrid i/o request scheduling technique for near-data processing based ssd," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.
[14] J. Liu, A. C. Arpaci-Dusseau, and R. H. e. a. Arpaci-Dusseau, "File systems as processes," in *HotStorage*, 2019.
[15] A. Mathur, M. Cao, and S. e. a. Bhattacharya, "The new ext4 filesystem: current status and future plans," in *Proceedings of the Linux symposium*, vol. 2, 2007, pp. 21–33.
[16] S. Park and S.-Y. Ohm, "New techniques for real-time fat file system in mobile multimedia devices," *IEEE Transactions on Consumer Electronics*, vol. 52, no. 1, pp. 1–9, 2006.
[17] "Nvm express 1.2a," 2015. [Online]. Available: https://nvmexpress.org/wp-content/uploads/NVM-Express-1_2a.pdf
[18] "Filebench," 2014. [Online]. Available: https://filebench.sourceforge.net/wiki/index.php/Filebench