FDPFS: Leveraging File System Abstraction for FDP SSD Data Placement

Ping-Xiang Chen, *Graduate Student Member, IEEE*, Dongjoo Seo, and Nikil Dutt, *Life Fellow, IEEE*

***Abstract*—Flexible data placement (FDP) is an emerging interface within the NVM express (NVMe) storage standard, aiming to decrease write amplification factor (WAF) in solid state drives (SSDs) through explicit user-controlled data placement. Currently, the FDP ecosystem burdens embedded software programmers with low-level systems programming to efficiently deploy FDP SSDs. We propose FDPFS, a file system that elevates the abstraction to file systems by exposing FDP SSDs as directories to which programmers can easily group and direct semantically similar data for user-controlled data placement. Under the hood, FDPFS performs the tedious low-level tasks of interfacing and assigning these semantically grouped data to different SSD erase blocks to reduce WAF, and improve overall SSD performance and lifetime. Our case study on the filebench benchmark demonstrates that our FDPFS prototype not only eases explicit data placement, but also yields up to 34% reduction in the SSD WAF which promises improved overall performance and lifetime of the SSD.**

***Index Terms*—Data placement, endurance, file systems, performance, solid-state drives (SSDs).**

I. INTRODUCTION

THE POPULARITY of flash storage in embedded systems stems from its advantages: low cost, small size, faster access speed, and high-data density [1]. Nevertheless, solid-state drives (SSDs) suffer from unpredictable performance due to garbage collection (GC) [2], [3], which incurs extra writes and increases the write application factor (WAF), which in turn reduces the write-cycle-limited lifespan of SSDs.

Flexible data placement (FDP) [4] is a new feature within the NVM express (NVMe) standard for SSDs that facilitates reducing the GC overhead. Embedded software (ES) programmers can use the FDP interface to explicitly control data placement on FDP-enabled SSDs. As illustrated in Fig. 1, the FDP interface allows the programmer to exploit semantic knowledge of the application/data (e.g., streams with similar data rates or access patterns) and perform explicit data placement of erase block groups (lower half of Fig. 1) to minimize GC overhead, improve performance and extend the lifetime of SSDs. However, as shown in Path ② in Fig. 2, current FDP approaches place a huge burden on the ES programmer to understand and perform tedious low-level systems programming for explicit data placement (e.g., calls to io_uring and the emerging NVMe char interface [5]). Instead, our FDPFS approach (Path ① in Fig. 2) eases explicit data placement by elevating the placement abstraction to the file system abstraction via directories, leaving FDPFS to perform the low-level detailed placement.

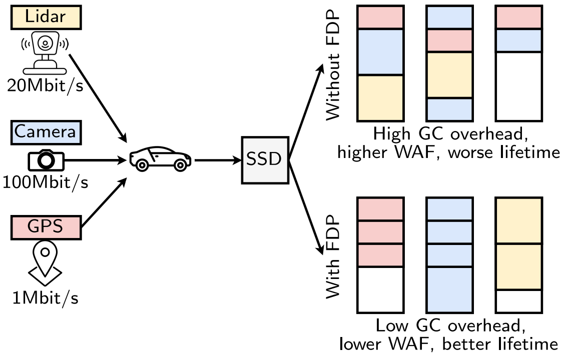


Fig. 1. Explicit data placement with FDP SSDs.

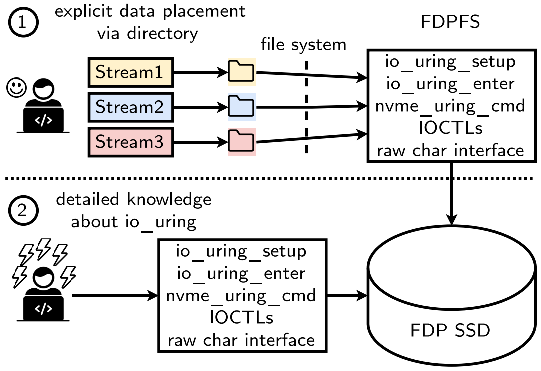


Fig. 2. FDPFS eases explicit data placement on FDP SSDs.

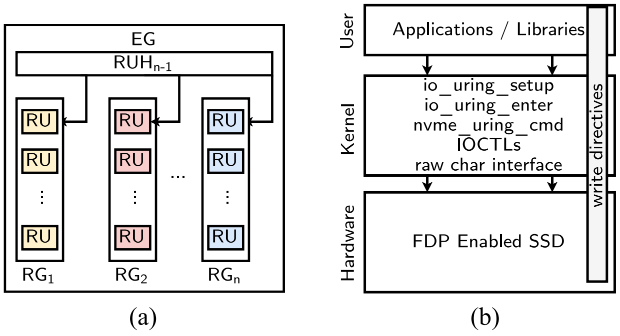
To this end, FDPFS makes the following contributions: 1) FDPFS elevates the data placement abstraction, allowing ES programmers to exploit data semantics for explicit placement via simple directory setting; 2) FDPFS remove the burden of low-level system programming for programmers to achieve optimized user-controlled placement on FDP SSDs; and 3) FDPFS achieves lower WAF with easy-to-use user-controlled data placement directives, demonstrating FDPFS's benefits the potential of applying FDP SSDs for embedded systems.

Manuscript received 28 July 2024; accepted 30 July 2024. This work was supported in part by J. Yang and Family Foundation Fellowship. This manuscript was recommended for publication by A. Shrivastava. *(Corresponding author: Ping-Xiang Chen.)*

The authors are with the Department of Computer Science, University of California at Irvine, Irvine, CA 92697 USA (e-mail: p.x.chen@uci.edu; dseo3@uci.edu; dutt@ics.uci.edu).

Digital Object Identifier 10.1109/LES.2024.3443205

1943-0671 © 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.



(a) (b)

Fig. 3. FDP. (a) FDP interface. (b) Open source ecosystem.

TABLE I
PLACEMENT HANDLE LIST

Placement Handle	Reclaim Unit Handle Identifier
0	0
1	1
n-1	n-1

II. BACKGROUND

A. *FDP SSDs and File System in Userspace*

FDP [6] is a recent addition to the NVMe standard that optimizes data storage on SSDs through explicit data placement directives. Fig. 3 shows the FDP SSD interface and the current ecosystem for FDP deployment. As shown in Fig. 3, each endurance group (EG) represents an FDP configuration comprising: 1) one or more reclaim units (RUs); 2) one or more reclaim groups (RGs); and 3) one or more RU handles (RUH) that reference to a RU in each RG. The ES programmer's directive to explicitly control data placement is achieved via a placement handle list, as shown in Table I. By specifying the placement handle identifier to the FDP SSD, the host directs the write data to the desired RG. ES programmers can therefore map data with different semantics (e.g., life times) to different erase blocks, reducing write amplification and improving overall SSD performance and lifetime (Recall Fig. 1).

The current open source ecosystem for leveraging FDP interface [5] (Fig. 3) is built on top io_uring with NVMe generic char interface. io_uring [7], [8] is a cutting-edge subsystem in Linux that supports efficient and scalable asynchronous I/O operations for storage and network tasks. It uses ring buffers to communicate between applications and the kernel, reducing system calls and improving performance. io_uring relies on shared ring buffers—a submission queue (SQ) for sending requests and a completion queue (CQ) for receiving results—to handle communication between user programs and the kernel. It prepares the I/O by extracting an entry from SQ called `SQE`, fills up the `SQE` and submits the I/O by calling `io_uring_enter` system call. This new FDP paradigm requires low-level systems programming to manage ring buffers and queues for submitting and tracking I/O operations, as shown in Algorithm 1. This places a huge burden on ES programmers, and requires a deep understanding of system internals (② in Fig. 2).

Algorithm 1 Setting Up io_uring With FDP Devices

```
io_uring_init()
Create SQE
Specify operation
SQE->opcode = IORING_OP_URING_CMD
SQE->cmd_op = NVME_URING_CMD_IO
Setup nvme_uring_cmd
Submit SQE to io_uring ring
io_uring_enter()
Wait for CQE
Process CQE
Mark CQE as completed
```

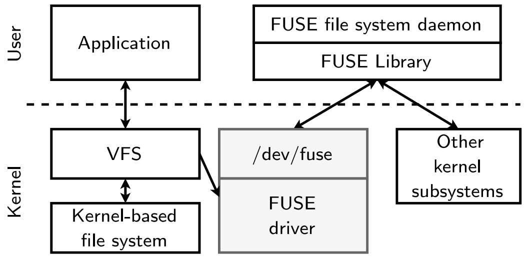


Fig. 4. FUSE high-level architecture.

File systems in user space (FUSE) [9], provides an alternative by allowing developers to write custom file systems as regular user programs [10]. Fig. 4 illustrates FUSE's core architecture. When a program interacts with the mounted FUSE file system, the operating system redirects the request to the FUSE kernel driver. This driver creates a request structure and stores it in a queue. The program might then wait for a response. Meanwhile, a separate user-level FUSE daemon awakens, reads the request from the kernel queue via a special device file (`/dev/fuse`), and processes it based on the file system's logic. The request might be forwarded to the underlying file system or other kernel subsystems. Once finished, the FUSE daemon writes the response back to `/dev/fuse`, notifying the kernel driver. Our proposed FDPFS approach exploits the FUSE interface to enable explicit data placement at the file system directory level. We begin in Section III-A by outlining the FDPFS approach. Section III-B then describes how ES programmers can intuitively exploit data semantics to achieve explicit data placement via the FUSE interface, and Section III-C describes how FDPFS orchestrates the FUSE file system with the current io_uring ecosystem for using FDP devices.

III. FDPFS

A. *Overview*

Fig. 5 shows an overview of FDPFS. First, when mounting FDPFS on a specific folder, FDPFS needs to get the placement identifier information from the FDP device. (① in Fig. 5). Second, after getting the device information from the FDP device, FDPFS exposes each placement identifier as a directory to the user space (② in Fig. 5). For each mounted directory, a worker thread is launched listening to

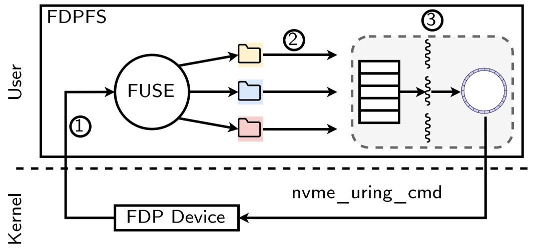
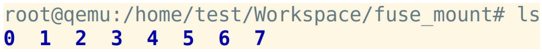


Fig. 5. Overview of FDPFS.



```
root@qemu:/home/test/Workspace/fuse_mount# ls
0  1  2  3  4  5  6  7
```

Fig. 6. FDPFS abstracts each placement identifier in FDP device as a directory to programmers.

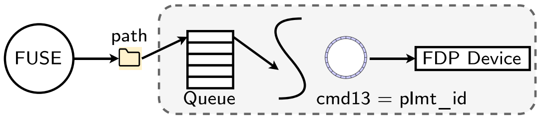


Fig. 7. Inter process communication in FDPFS.

the message queue created during the mounting of FUSE. Each worker thread also initializes a shared ring buffer with `io_uring_setup` for serving incoming write requests by sending `nvme_uring_cmd` to the device. As a result, when FDPFS captures a write request based on a directory name, it submits the I/O request corresponding to the placement identifier of FDP SSDs (③ in Fig. 5).

B. Abstraction Benefits for ES Programmers

Fig. 6 shows the file system abstraction provided by FDPFS with FDP SSDs. The placement handle list can be obtained from the FDP device through xNVMe [11] during FUSE initialization. Each placement identifier is represented by one directory with the directory name starting from 0 to number of placement identifiers$-$1. This placement identifier exposure enables ES programmers to exploit data semantics intuitively, such as by clustering and redirecting similar lifetime data streams to separate placement identifiers by merely assigning a directory name. For example, ES programmers can direct data with small and random updates by simply saving the data to the same directory name within FDPFS; and direct data streams with large and sequentially written data to another directory. By doing so, data segregation can be easily accomplished for the FDP device (Fig. 1). FDPFS's file-system directory abstraction relieves ES programmers from performing complex, low-level io_uring ring buffer management to achieve explicit data placement on FDP SSDs (Fig. 2).

C. Communication From FDPFS to FDP

Fig. 7 summarizes how FDPFS captures the write events and directs the data streams to FDP devices based on the directory name (③ in Fig. 5). During the initialization of FDPFS, a worker thread for each mounting directory is created with a dedicated io_uring ring buffer for setting `nvme_uring_cmd` and submitting I/O requests to the FDP device. The worker thread is listening to the message queue and processes the message generated by the FUSE daemon which captures the write events on specific directory. The worker thread will setup `nvme_uring_cmd` based on the directory name (path), submitting the I/O request with the placement directives to the FDP device. By setting `cmd13` in `nvme_uring_cmd` with the directory's placement identifier, FDPFS can write the incoming user-directed data streams to the target placement identifier set by ES programmers.

IV. CASE STUDY

In this section, we use filebench [12] to showcase how ES programmers can explicitly perform optimized data placement through FDPFS to FDP SSDs via file semantics.

A. Experimental Setup

We evaluate FDPFS functionality using QEMU [13] (v8.2.1) with FDP NVMe emulation on Ubuntu 22.04. We use libfuse 3.16.2 for building the FDPFS prototype and xNVME v0.7.4 library to obtain the placement handle list from the emulated FDP device. The Linux kernel version is 6.7.9 within QEMU. We set up a machine equipped with a single Intel Xeon Platinum 8321HC Processor (26 cores, 1.4 GHz) and 96 GB of memory (DDR4). We use FEMU [14] to emulate both normal and FDP SSD. First, we emulate a 16-GB normal SSD. Then, we emulate two 8-GB SSDs as a FDP device with two placement identifiers in a single namespace. To emulate the placement identifier exposed by FDPFS, we mount two 8-GB SSDs with the Ext4 file system. The 16-GB closed-box SSD is also mounted with the Ext4 file system.

B. Case Study—Filebench's `oltp` With FDPFS

1) Efficacy of FDPFS for Explicit Data Placement on the FDP SSD: We use the Online Transaction Processing (`oltp`) workload from filebench [12] as a demonstrator application to highlight the abstraction benefits provided by FDPFS. The `oltp` workload has two writer processes with differing write patterns: 1) the database writer process with a 2k size random write pattern and 2) the log files writer process with a 256k size random write pattern. We exploit the semantics of these different write patterns to explicitly place their data using the FDP SSD into different groups as illustrated in Fig. 1. Accordingly, using FDPFS we assign the database and log file writer processes to separate directories by simply setting the `path` argument for each of the two processes. FDPFS then greatly simplifies explicit data placement on FDP SSD at the directory level leaving the tedious management of io_uring (Algorithm 1) under the hood of FDPFS. Without detailed knowledge about complex ring buffer management in io_uring, ES programmers can now explicitly place data streams with similar lifetimes on the FDP SSD by assigning working directories exposed by FDPFS (Path ① in Fig. 2). As shown in Fig. 8, FDPFS is able to achieve an average 0.31 reduction in WAF (same as the native FDP SSD ecosystem using io_uring), but while greatly reducing the tedious low-level programming burden via the file system abstraction provided by FDPFS.

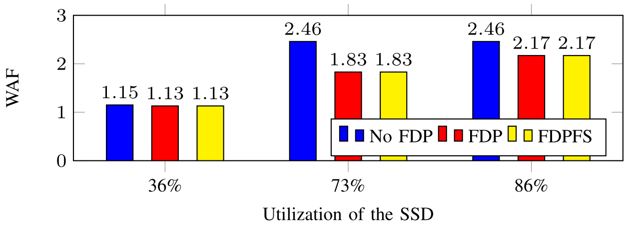


Fig. 8. WAF comparison between emulated normal and FDP SSDs with FDPFS.

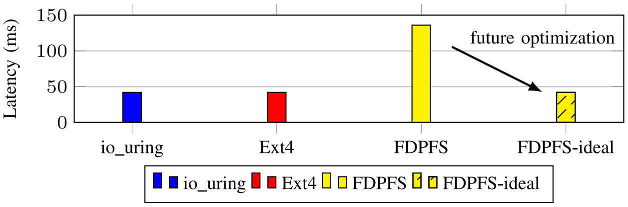


Fig. 9. Comparison of average I/O requests issue latency to the underlying FDP device.

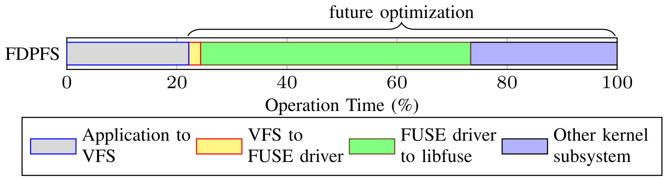


Fig. 10. Overhead breakdown of FDPFS issue I/O path to emulated FDP device.

2) FDPFS Overheads: Since FDPFS leverages FUSE to provide the directory abstraction, it incurs some overhead in capturing the write I/O requests and submitting the relevant `nvme_uring_cmd` based on the directory corresponding to the placement identifier. Fig. 9 shows this extra overhead. Although there is an average 94 ms latency increment per I/O requests compared to io_uring and in-kernel file system, this abstraction enables researchers to easily leverage the benefits provided by FDP SSDs (WAF reduction could be up to 0.63 for `oltp`). The overhead breakdown of the I/O issue path in FDPFS is shown in Fig. 10. The breakdown reveals that 49.05% of the total time is consumed by the FUSE driver interacting with the libfuse library for each write I/O requests. This highlights our ongoing work in relieving the overhead ("future optimization" arrow in Fig. 9) that could bring down the latency to be on par with native FDP.

V. CONCLUSION AND FUTURE WORKS

We presented FDPFS, an approach leveraging FUSE to elevate the programmer's abstraction for data placement to the file system level for explicit data placement in FDP devices. FDPFS exposes FDP SSDs as directories, enabling programmers to easily group and direct semantically similar data for user-controlled data placement, relieving them of the tedious low-level programming required for native FDP deployment. Our case study on the filebench benchmark demonstrates that our FDPFS prototype not only eases explicit data placement, but also yields up to 34% reduction in the SSD WAF which promises improved overall performance and lifetime of the SSD. Our ongoing work addresses reduction in the I/O system overhead incurred by FUSE by combining [15] with a user space page cache for batching multiple requests interfaced with io_uring. We also plan to integrate automatic stream separation for SSDs [16] to further ease the programmer's burden in explicit data placement. Finally, our ongoing work expands deployment of FDPFS to applications with semantically richer data streams (e.g., AV systems with diverse sensors) that can fully exploit the ease of using FDPFS.

REFERENCES

[1] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti, "Introduction to flash memory," *Proc. IEEE*, vol. 91, no. 4, pp. 489–502, Apr. 2003.

[2] M.-C. Yang, Y.-M. Chang, C.-W. Tsao, P.-C. Huang, Y.-H. Chang, and T.-W. Kuo, "Garbage collection and wear leveling for flash memory: Past and future," in *Proc. Int. Conf. Smart Comput.*, 2014, pp. 66–73.

[3] D. Seo, P.-X. Chen, H. Li, M. Bjørling, and N. Dutt, "Is garbage collection overhead gone? case study of F2FS on ZNS SSDs," in *Proc. 15th ACM Workshop Hot Topics Storage File Syst.*, 2023, pp. 102–108.

[4] C. Sabol and R. Stenfort. "Hyperscale innovation: Flexible data placement mode (FDP)." 2022. [Online]. Available: https://nvmexpress.org/wp-content/uploads/Hyperscale-Innovation-Flexible-Data-Placement-Mode-FDP.pdf

[5] A. Manzanares and J. Granados. "Flexible data placement open source ecosystem." 2023. [Online]. Available: https://www.snia.org/educational-library/flexible-data-placement-open-source-ecosystem-2023

[6] M. Allison and J. Rudelic. "What is the NVM express® flexible data placement (FDP)?" 2022. [Online]. Available: https://www.youtube.com/watch?v=c8Jw_WANn6A

[7] J. Axboe. "Efficient IO with io_uring." 2019. [Online]. Available: https://kernel.dk/io_uring.pdf

[8] K. Joshi et al., "I/O Passthru: Upstreaming a flexible and efficient I/O path in Linux," in *Proc. 22nd USENIX Conf. File Storage Technol. (FAST)*, 2024, pp. 107–121.

[9] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To FUSE or not to FUSE: Performance of user-space file systems," in *Proc. 15th USENIX Conf. File Storage Technol. (FAST)*, 2017, pp. 59–72.

[10] T. Yoshimura, T. Chiba, and H. Horii, "EvFS: User-level,event-driven file system for non-volatile memory," in *Proc. 11th USENIX Workshop Hot Topics Storage File Syst. (HotStorage)*, 2019, pp. 1–7.

[11] S. A. Lund, P. Bonnet, K. B. Jensen, and J. Gonzalez, "I/O interface independence with xNVMe," in *Proc. 15th ACM Int. Conf. Syst. Storage*, 2022, pp. 108–119.

[12] R. McDougall and J. Mauro. "FileBench." 2005. [Online]. Available: http://www.nfsv4bat.org/Documents/nasconf/2004/filebench.pdf

[13] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. USENIX Annu. Tech. Conf.*, vol. 41, 2005, pp. 1–6.

[14] H. Li, M. Hao, M. H. Tong, S. Sundararaman, M. Bjørling, and H. S. Gunawi, "The CASE of FEMU: Cheap, accurate, scalable and extensible flash emulator," in *Proc. 16th USENIX Conf. File Storage Technol. (FAST)*, 2018, pp. 83–90.

[15] K.-J. Cho, J. Choi, H. Kwon, and J.-S. Kim, "RFUSE: Modernizing Userspace filesystem framework through scalable kernel-userspace communication," in *Proc. 22nd USENIX Conf. File Storage Technol. (FAST)*, 2024, pp. 141–157.

[16] J. Yang, R. Pandurangan, C. Choi, and V. Balakrishnan, "AutoStream: Automatic stream management for multi-streamed SSDs," in *Proc. 10th ACM Int. Syst. Storage Conf.*, 2017, pp. 1–11.