InDeF: An Advanced Defragmenter Supporting Migration Offloading on ZNS SSD

Wenjie Qi, Zhipeng Tan*, Jicheng Shao, Lihua Yang, Yang Xiao
Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System
Engineering Research Center of Data Storage Systems and Technology, Ministry of Education of China
School of Computer Science & Technology, Huazhong University of Science & Technology
Email: {hustqwj, tanzhipeng, shaojicheng, lihuayang, menguozi}@hust.edu.cn, *corresponding author

***Abstract*—The Zoned Namespace (ZNS) emerges as a new storage interface with advantages such as low garbage collection overhead and low over-provisioning cost. However, it is vulnerable to fragmentation due to its out-of-place updates and the multi-threaded writing behaviors of applications. Fragmentation splits I/O requests and causes resource conflicts within the device, degrading I/O performance. Most defragmentation tools need to read the entire contents of the file from the SSD to the host memory and then rewrite the data to contiguous space in the SSD. The host-level migration operations prolong the elapsed time of defragmentation and excessive write operations for data migration even reduce the device lifetime.**

Our experiments discover that defragmenting data with a low degree of fragmentation or cold data provides little performance gain. With the observation, we propose a new defragmentation tool called InDeF to reduce the defragmentation overhead. InDeF combines the degree of logical and physical fragmentation and access hotness to filter out the fragments that have little impact on I/O performance for reducing the write traffic of the SSD. To take advantage of the internal flash chip parallelism, InDeF offloads the data migration on the SSD, decreasing the elapsed time of defragmentation. Evaluation results show that InDeF decreases the elapsed time of defragmentation by 91.6%∼94.2% and the amount of data migration by 38.1%∼66.7% compared with conventional defragmentation tools.

***Index Terms*—Fragmentation, filesystem, zoned namespaces, solid-state disks**

I. Introduction

Zoned Namespace (ZNS) SSDs have become a hot topic in the research community [1]–[3]. Since ZNS uses a sequential write-only zone, accessing it requires a log-structured file system (LFS) such as F2FS [4]. Due to the append-only write policy and the multi-threaded write operations, LFS is highly prone to fragmentation [5]–[8]. File fragmentation caused by discontinuous logical addresses is called logical fragmentation [5], [9]. Most file systems provide their own defragmentation tools to reduce the impact of fragmentation on I/O performance [10]–[12]. However, the conventional defragmentation tools have two main problems.

On the one hand, most of them are designed based on conventional SSDs, so they cannot perceive physical fragmentation. The I/O performance of a flash device is largely affected by the average I/O parallelism [9], [13], [14]. File fragmentation due to limited I/O parallelism in accessing files is called physical fragmentation [9]. In conventional SSDs, the flash translation layer (FTL) hides the management of the underlying media and the presence of garbage collection (GC) from the host. Logically adjacent logical pages may be physically far apart [2]. Conventional SSD is a black box for conventional defragmentation tools. Different from the data management of conventional SSDs, ZNS SSDs group the logical address space in zones. ZNS SSDs align the zone with the physical media boundary of the device, and the host manages the placement of data [1]. Therefore, we can take advantage of the characteristics of ZNS SSDs to get the distribution of data within the device, and to identify physical fragments at the host side.

On the other hand, conventional defragmentation tools have a high defragmentation overhead, e.g. `defrag.f2fs` [11]. It is provided by F2FS. `defrag.f2fs` reads the entire contents from the underlying device into the host memory and rewrites the data into the contiguous space of the device, which generates excessive I/O operations. Firstly, the use of `defrag.f2fs` is too difficult for novices. The user needs to determine the range of data to be defragmented and the input parameters include file system block-level addresses. These can be challenges for users due to the lack of sufficient system information. Secondly, since flash devices have a limited number of program/erase cycles [15], excessive data migration significantly increases the program/erase operations of ZNS SSDs and shortens the lifetime of ZNS SSDs. For this reason, many computer experts usually advise not to defrag SSDs frequently [16]. Finally, the defragmentation process consumes a long time and substantially degrades the performance of foreground applications due to the excessive I/O operations.

In this paper, we first define the degree of logical and physical fragmentation of an I/O request and experimentally investigate the characteristics of ZNS SSD performance degradation caused by fragmentation. We find that defragmenting data with a low degree of fragmentation or cold data that is rarely accessed provides little performance gain. With the observation, we propose a new data selection method to decrease the amount of data migration for defragmentation.

Although ZNS SSD shifts the responsibility of data management to the host, it requires a detailed design to identify physical fragments at the host side. We introduce an SSD-internal zone mapping policy suitable for ZNS SSDs, where the host can get the physical location of the logical block inside the device by the segment number and in-segment offset of the logical block. This gives the host the opportunity to get the

2576-6996/22/$31.00 ©2022 IEEE
DOI 10.1109/ICCD56317.2022.00052

physical distribution of data on the flash device by logical address and thus can identify the physical fragments at the host side.

The whole process of conventional defragmentation tools requires a lot of I/O operations and consumes host resources. We can offload the data migration from the host to the SSD by adding a command to the ZNS SSD command set. Migration offloading can make full use of the parallel units inside the device to improve migration efficiency and reduce the host memory consumption during migration.

We propose **In**-device **def**ragmenter (**InDeF**), a new ZNS SSD-based defragmenter. InDeF first analyzes the I/O characteristics of the application. Then InDeF combines the degree of fragmentation and access hotness to filter out fragments that have little impact on I/O performance for minimizing the amount of data migration for defragmentation. Finally, by adding a command to the ZNS SSD command set, InDeF offloads the data migration from the host to the SSD to decrease the elapsed time of defragmentation. Compared to `defrag.f2fs`, in particular for fileserver workload, InDeF decreases the amount of data migration by 44.8% and the elapsed time of defragmentation by 94.2%. InDeF increases the chip utilization by 8.9× by reducing the idle intervals invoked during the host-level copy operations and has a negligible memory consumption.

Overall, this paper makes the following contributions:

- Define the degree of logical and physical fragmentation of an I/O and analyze the characteristics of ZNS SSD performance degradation caused by fragmentation.
- Propose a new data selection method to minimize the amount of data migration for defragmentation.
- Identify physical fragments at the host side by introducing an SSD-internal zone mapping policy suitable for ZNS SSDs.
- Decrease elapsed time of defragmentation by expanding the ZNS command set to offload data migration from the host to the SSD.

The remainder of this paper is organized as follows. Section II reports our motivation experiments. A detailed description of InDeF is given in Section III. Experimental results follow in Section IV, and related work is summarized in Section V. Finally, Section VI concludes with a summary.

II. Motivation

A. The Performance Analysis on Fragmentation

(1) Fragmentation accumulation under the workloads. The application usually performs tasks in a multi-threaded mode, which may cause fragmentation. We measure the sequential read bandwidth of files before and after the workloads

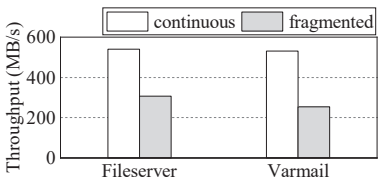


Fig. 1. The sequential read performance of different files.

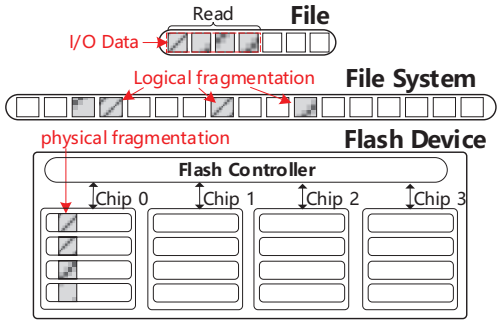


Fig. 2. The logical and physical fragmentation for the I/O data.

are running. The results are shown in Figure 1. We run Filebench [17] fileserver and varmail separately, and the details of the experimental setup are described in Section 5. Due to the large number of fragments accumulated by running multi-threaded applications, the sequential read bandwidth of files is reduced by 42.2% (fileserver) and 51.7% (varmail) compared to before the workloads run.

(2) Definition of the fragmentation of an I/O request. Due to file system fragmentation and internal operations of flash SSDs (such as GC and wear-leveling), the distribution of a portion of the accessed data becomes more random, leading to internal resource conflicts. We define the logical and physical fragmentation of an I/O request prior to our study, as shown in Figure 2. The current Linux kernel uses the bio and request structures to handle I/O operations, but they both represent only contiguous file system blocks. Since the logical addresses of the fragmented data are not contiguous, the large sequential I/O issued to the fragmented data is split into multiple small random I/O. This is called **request splitting** [18]. The excessive I/O can increase the management overhead of the kernel and decrease the read bandwidth [19]. We define the degree of logical fragmentation (**DoLF**) of an I/O request as **the number of logical fragments in an I/O range**.

Since SSDs consist of multiple chips[1] that can be accessed in parallel, the definition of physical fragmentation based on physical data sequentiality on disks makes little sense for SSDs. Due to the I/O performance of a flash device being largely affected by the average I/O parallelism [9], [13], [14], we measure **the degree of physical fragmentation of an I/O request by how evenly the data in the I/O range are distributed among the flash parallel units.** The degree of physical fragmentation (**DoPF**) can be expressed as follows:

$$DoPF = \frac{\sum_{i=1}^{L} (N_i - \frac{M}{L})^2}{L} \quad (1)$$

M represents the number of data blocks accessed by an I/O, L is the number of parallel units of flash storage, and N_i indicates the number of data blocks located in the i-th parallel unit ($\sum_{i=1}^{L} N_i = M$). A larger DoPF indicates a higher degree of physical fragmentation and a lower degree of I/O parallelism.

(3) The impact of logical and physical fragmentation on

[1]A flash chip consists of multiple flash dies and each flash die has multiple flash planes. In this paper, we assume a flash chip has one flash die and one flash plane structure for simplicity.

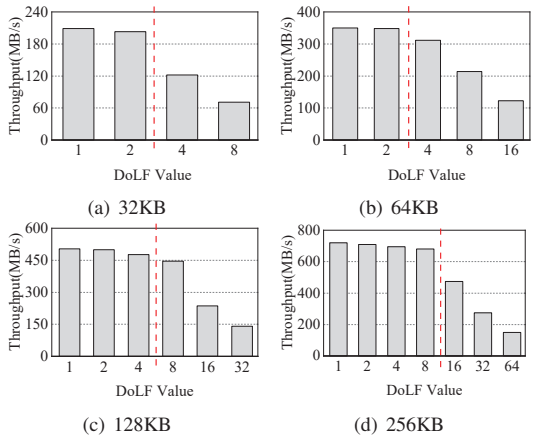


Fig. 3. The sequential read performance with various DoLF value. A larger DoLP value means more fragmentation.

I/O performance. Firstly, we measure the I/O performance with different degrees of logical fragmentation, as shown in Figure 3. We select four common I/O sizes in Linux: 32KB, 64KB, 128KB, and 256KB. The horizontal axis is the DoLF value and the vertical axis is the read throughput. Even though a high DoLF value has a significant impact on I/O performance, a low DoLF value has a small impact on I/O performance. For example, in Figure 3(d), when the DoLF value is less than 8, the logical fragmentation has a negligible impact on I/O performance. This result is due to the fact that when the number of fragments in the I/O range is less, the kernel overhead is also relatively low. However, we find that most conventional defragmentation tools migrate all the contents of a data range even when there are few fragments [11], [12], which may cause unnecessary data reads and writes.

Secondly, we measure the I/O performance with different degrees of physical fragmentation, as shown in Figure 4. We change the layout of the data in the flash device and then measure the sequential read bandwidth with an I/O size of 256KB. Each value on the horizontal axis represents the number of chips used for an I/O request. The left axis is the read throughput and the right axis is the corresponding DoPF value. When the number of chips is less than 6, the I/O parallelism has a significant impact on I/O performance, the DoPF value increases significantly, while the I/O performance decreases greatly. However, when the DoPF value is small, e.g., less than 40, physical fragmentation has a negligible impact on I/O performance. This is due to the fact that the software overhead of I/O dominates the total I/O latency when the I/O physical parallelism is greater than a certain degree.

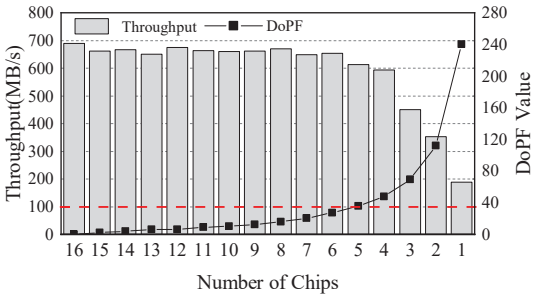


Fig. 4. The sequential read performance with various DoPF value. A smaller number of chips means lower I/O parallelism.

(4) How to select the appropriate fragments for defragmentation. In addition, modern storage systems typically perform data access in non-uniform distribution [20], [21]. When the read count of the data is higher, the data are likely to be more hotly accessed. Multiple reads to fragmented data accumulate the access latency caused by fragmentation. Therefore, the performance gain of defragmenting fragmented data is proportional to the read count of the data.

With the aforementioned observations, we find that **defragmenting data with a low degree of fragmentation or cold data that is rarely accessed provides little performance gain**. It is necessary to consider the degree of logical and physical fragmentation and the access hotness of data in order to select the most suitable data for defragmentation. For the data in I/O range, we define the I/O data defragmentation priority (**IODP**):

$$IODP = (\alpha \cdot DoLF + \beta \cdot DoPF) \times readcount \quad (2)$$

$DoLF$ is the degree of logical fragmentation, $DoPF$ is the degree of physical fragmentation, α and β are used to normalize the $DoLF$ and $DoPF$, and $readcount$ is the read count of the I/O data. The larger the IODP is, the higher the defragmentation priority of I/O data. We can sort the I/O data based on IODP and filter out the fragments that have little impact on I/O performance to reduce the amount of data migration for defragmentation.

B. The Overhead Analysis on Conventional Defragmenter

In this section, we analyze the overhead of `defrag.f2fs`. As shown in Figure 5, `defrag.f2fs` contains five phases: memory page allocation, data read, contiguous space allocation, data write, and metadata update.

If the data to be migrated is not cached in the page cache, the host will read the relevant data from the device. **Since the data to be migrated is usually fragmented, this may cause the host to send a large number of read requests.** The corresponding space must be allocated in the host page cache before sending read requests. A large number of reads into memory may cause a significant increase in the host memory usage and invoke page frame reclamation (see Section IV-E). Secondly, until all the target data has been read into the page cache, defrag.f2fs allocates a contiguous free space and writes all the data sequentially to the SSD, thus **resulting in a large chip idle interval in the SSD**. As a result, these operations introduce a large amount of overhead in the kernel

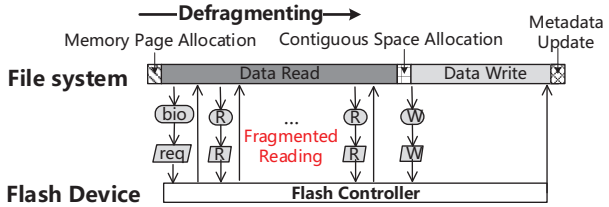


Fig. 5. The overall process of conventional defragmenter.

I/O stack, prolonging the elapsed time of defragmentation and degrading the performance of co-running applications (see Section IV-D). Therefore, these observations motivate us to offload data migration from the host to the SSD. The SSD-internal controller can fully use the parallel units of SSD to reduce the elapsed time of defragmentation.

III. The Design of InDeF

InDeF is designed to achieve two main goals: one is to minimize the amount of migration data for defragmentation to reduce the write traffic of the underlying device, and the other is to take advantage of the internal parallelism of ZNS SSDs to decrease elapsed time of defragmentation. Figure 6 shows the overview of InDeF, which consists of three modules. The fagmentation information management module collects I/O information from the filesystem and manages I/O fragmentation information. The fragmentation calculation module calculates the DoLF value and DoLP value based on the collected I/O information. The defragmentation module filters the fragments based on the I/O fragmentation information and offloads the data migration from the host to the SSD.

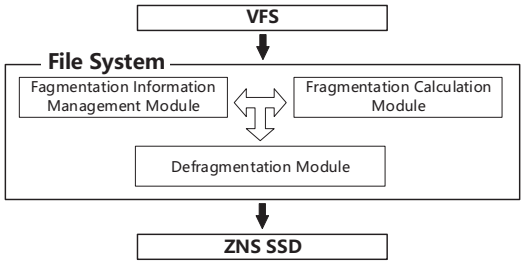


Fig. 6. The overview of InDeF.

A. Fagmentation Information Management Module

(1) I/O Monitor. To get the access hotness of data, InDeF collects the I/O access pattern of the applications. Some I/O may be served by the page cache and not reach the storage device and the Virtual File System may change the I/O range of the application based on the read-ahead policy adopted by the system. Therefore, monitoring I/O at the system call layer may be inaccurate. InDeF monitors I/O activity at the file system layer to truly and accurately capture I/O requests arriving at the ZNS SSD. The I/O information contains the inode number, the I/O size, and the start index number of the I/O. We can find out the corresponding files with the inode numbers. The start index number and the I/O size are used to determine the range of file blocks that the I/O tries to access.

(2) I/O Information Management. After monitoring I/O information, InDeF creates two separate ordered lists for file and I/O request information. The former is sorted by the file inode number and the latter is sorted by the start index. As shown in Figure 7, each file entry of the file list points to an I/O information linked list. For each I/O information entry, InDeF first finds the corresponding file entry in the file list according to the inode number and then inserts the I/O into the I/O information linked list according to the start index. If there is already an entry of this I/O information in the I/O information chain list, the read count of the original entry is

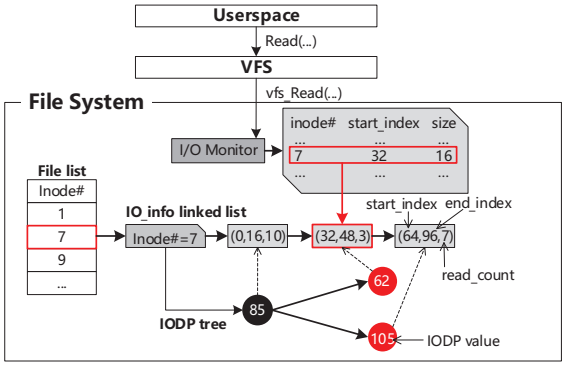


Fig. 7. The fagmentation information management module of InDeF.

increased. Since I/O requests may have overlapping addresses, InDeF performs a merge operation on these I/O requests and preserves the largest address range.

(3) Fagmentation Information Management. After calculating the DoLF and DoPF (see Section III-B), InDeF calculates the IODP value of each I/O information entry by Equation 2. As shown in Figure 7, InDeF inserts the IODP of each I/O information into a red-black tree (called IODP tree) in order. The nodes in the IODP tree are sorted based on the IODP values. Each node in the IODP tree also points to an entry of the I/O information linked list that represents its I/O range. The root node of the IODP tree for each file is stored in the I/O information linked list.

B. Fragmentation Calculation Module

After getting the I/O information, the fragmentation calculation module calculates the degree of logical and physical fragmentation. In addition, we introduce an SSD-internal zone mapping policy in order to get the physical layout of the data by the logical block addresses in the I/O range.

(1) Zone Mapping in ZNS SSD. There are no zone mapping constraints in the ZNS specification [22]. We introduce a general and efficient SSD-internal zone mapping policy. F2FS typically uses a 2MB size segment to manage the logical address space [4]. Figure 8 presents an example of the zone and flash blocks mapping, where the ZNS SSD has four parallel chips, a logical block corresponds to a physical flash page, and a zone is aligned with two consecutive segments of F2FS. Depending on the zone size, a physical zone consists of a set of blocks distributed in parallel chips, and pages in a physical zone are placed in different chips in an interleaved manner. This can take full advantage of the flash operation parallelism. The flash blocks corresponding to each zone are dynamically allocated, and the flash blocks not mapped to a zone are freely available for replacement operations such as wear leveling within the ZNS SSD. The i-th logical block of a segment is stored on the j-th flash chip that $j = i \ \% \ (the\ number\ of\ parallel\ flash\ chips)$. With this mapping that we have introduced, the fragmentation calculation module can calculate the degree of physical fragmentation by getting the physical distribution of data on the ZNS SSD.

(2) Calculating the degree of Fragmentation. The frag-

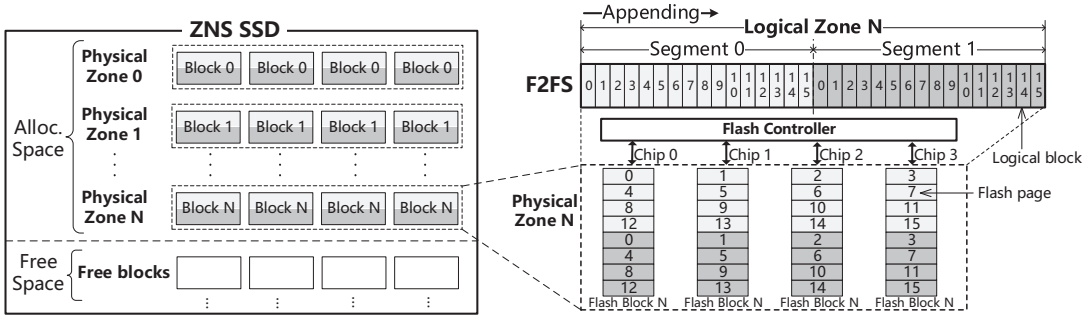


Fig. 8. An example of zone and flash blocks mapping.

mentation calculation module first converts the index number of each file block in the I/O range to a file system logical block address. The degree of logical fragmentation (DoLF) can be calculated by the number of logical fragments in the I/O range. The degree of physical fragmentation (DoPF) can be calculated by equation 1 after getting the distribution of data on the ZNS SSD.

An example of the calculation is depicted in Figure 9. The fragmentation calculation module gets the I/O information and then translates the index number to the file system logical block address by File-to-Storage Mapping. For logical fragmentation, we can know that there are 6 discontinuous fragments, so the DoLF is 6. For physical fragmentation, we first need to get the location of each logical block in the flash device by segment number and in-segment offset. By analyzing the number of data blocks on each chip, the physical layout of I/O can be derived. Finally, the DoPF can be calculated by Equation 1 as 2.5.

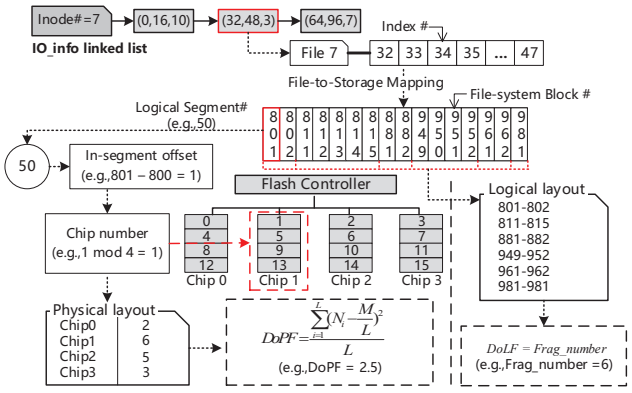


Fig. 9. An example of calculating the DoLF and the DoPF.

C. Defragmentation Module

The defragmentation module of InDeF consists of four phases, as shown in Figure 10. Firstly, the fragments filter selects the data to be migrated and filters out the fragments that have little impact on I/O performance. Secondly, the contiguous space allocation allocates contiguous free space from the zone. Thirdly, the data migration task of the host is offloaded to SSD. Finally, the file system persists metadata blocks of migrated data.

(1) Fragments filter.

According to Equation 2, migrating I/O data with a low IODP value may not contribute to a large performance gain. Based on the ordered IODP tree, the defragmentation module can easily truncate the I/O information entries that have a low IODP value. The system administrator can determine the percentage of entries that are truncated. After filtering out the fragments that have little impact on I/O performance in each file information linked list, InDeF defragments the data of the remaining entries. In addition, the defragmentation module allocates a contiguous space of the same size based on the size of the I/O data range. To ensure that the allocated space is contiguous, the process of the space allocation is protected by segment locking.

(2) Migration offloading.

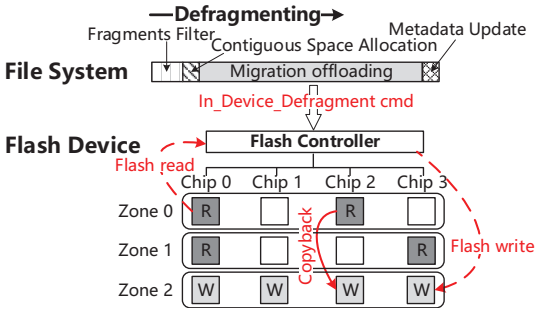


Fig. 10. The overall process of The Defragmentation Module.

Command generating. For each block in the I/O range, it is first checked whether it is cached in the page cache. If it is already cached and the page is dirty, the block needs to be excluded from the migration offloading and written from memory to the device. If the cached page is clean or not cached, it can be copied internally through migration offloading. InDeF divide the source data into two groups, one of which needs to be transferred from host memory to the device, and the other is processed by migration offloading.

Command sending. Secondly, InDeF sends the command to offload the data migration from the host to the SSD. We design the `In_Device_Defragment` command to implement the migration offloading. It contains a set of source LBAs and a set of destination LBAs. The command is highly efficient since each 4KB logical block is replaced by two 4-byte block addresses.

Command processing. Finally, The SSD-internal controller handles migration offloading and host requests according to the zone's write policy. As shown in Figure 10, the flash controller analyzes the physical flash chip number where the

original address and the destination address are located. If they belong to the same chip, then use the copyback command to perform data migration within the chip without off-chip data transfer, otherwise, read the data block to the DRAM on the SSD first, and then write to the destination address.

Even for read requests to the data of migration offloading, they can be handled normally through DRAM or flash chip on the SSD. Due to the out-of-place update policy, write requests to the data of migration offloading are not affected. Finally, the file system persists metadata blocks of migrated data to the device. F2FS implements checkpointing to provide a consistent recovery point.

TABLE I
THE SOFTWARE AND HARDWARE PARAMETERS

Host configuration	
CPU	Intel(R) Xeon(R)Silver 4208 CPU@2.10GHz
Memory	128 GB DRAM
OS info	Ubuntu 18.04 (kernel version 5.4.0)
Guest OS configuration	
CPU	4 vCPU
Memory	8 GB
OS info	Ubuntu 16.04 (kernel version 4.10.17)
File system	F2FS
NVMe SSD	16 GB
Flash memory configuration	
Flash chip type	TLC
Flash page	16 KB
Pages per flash block	128
Flash chips	16
Flash page read latency	80 μs
Flash page program latency	700 μs
DMA from/to flash controller	24 μs

IV. EVALUATION

In this section, we evaluate the performance of InDeF using FEMU, a QEMU-base NVMe SSD emulator [23]. The basic parameters of our experiments are summarized in Table I. The emulated NVMe SSD consists of sixteen parallel flash chips by default, which can be accessed in parallel via eight channels and two ways per channel. A zone consists of sixteen flash blocks in sixteen flash chips, and its size is 32MB. Each segment in F2FS contains 512 logical blocks of 4KB, and each zone is aligned with sixteen segments. Since the range of DoLF is (0, 64) and the range of DoPF is (0, 240), α is set to 4 and β is set to 1 for normalization in equation 2. For comparison with InDeF, we provide two advanced defragmentation tool: `defrag.f2fs` [11] and FragPicker [18]. FragPicker analyzes the I/O characteristics of the application and defragments only the file fragments that are hot to access. `defrag.f2fs` migrates the entire contents of a target file to a new contiguous space, which we call Conv.f2fs. The synthetic and macro benchmarks are used for evaluation. In all experiments, we fix the size of read requests as 256KB.

A. Synthetic Workloads

To evaluate the optimization of InDeF on logical and physical fragments separately, we create two fragment files (file A and file B) in synthetic workloads. File A consists of a series of sixty-four 4KB blocks, two 128KB blocks, and one 256KB block. Different from file A, file B consists of a series of 16KB logical blocks while controlling the physical distribution of each 16KB block in the SSD. We divide all 16KB blocks into three parts, the first part is distributed on two chips, the second part is distributed on eight chips, and the third part is distributed on all chips. File A has three types of degrees of logical fragmentation and file B has three types of degrees of physical fragmentation. We measure the sequential read performance of the files and the amount of data migration for defragmentation. We set the percentage of truncated entries as 60% in this experiment.

The normalized sequential read bandwidth before and after defragmentation is shown in Figure 11a. InDeF shows a similar level of performance improvement as FragPicker and Conv.f2fs. The normalized amount of data migration for defragmentation is shown in Figure 11b. In file A, FragPicker has 33.3% less data migration than Conv.f2fs, due to FragPicker filtering out the unfragmented I/O data. In file B, since FragPicker cannot identify physical fragments, FragPicker needs to defragment all the data in order to achieve similar performance improvement as Conv.f2fs. Since InDeF first sorts the data based on IODF and then filters out data that has little impact on I/O performance, it decreases the amount of data migration by about 60%.

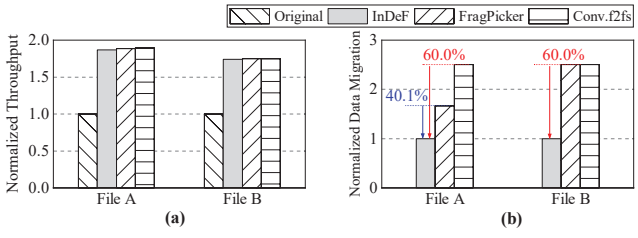


Fig. 11. The experimental results of synthetic workloads.

B. Macro Workloads

(1) Filebench Fileserver and Varmail. Since the recursive grep sequentially reads the entire files under a certain directory, the grep test is often adopted by research papers on fragmentation to measure the sequential read performance [7], [24]. We measure the elapsed time of the recursive grep after running Filebench fileserver and varmail workloads and calculated the grep throughput. For the fileserver, the average file size is approximately 4MB and the number of files is 3072. For the varmail, the average file size is approximately 2MB and the number of files is 6496. They both have 50 threads. We set the percentage of truncated entries as 30% (fileserver) and 20% (varmail) in this experiment.

Figure 12a shows the grep throughput improvement for fileserver and varmail. InDeF improve the throughput by about 72% (fileserver) and 112% (varmail), compared with that before defragmentation. This is due to the fact that defragmentation reduces the overhead caused by I/O fragmentation and device resource conflicts. Figure 12b shows the amount of data migration for fileserver and varmail. InDeF decreases the amount of data migration by 31.2% (fileserver)

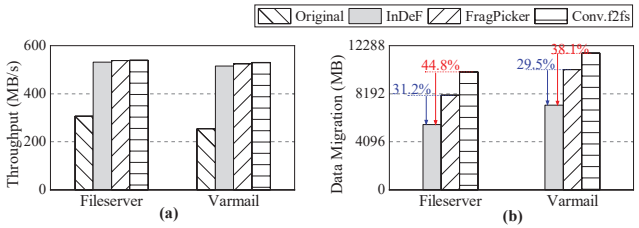


Fig. 12. The experimental results of the fileserver workload and varmail workload.

and 29.5% (varmail) compared with FragPicker. This is because FragPicker does not consider the degree of logical and physical fragmentation, and migrates some data that has little impact on I/O performance ineffectively. Compared with Conv.f2fs, InDeF decreases the amount of data migration by 44.8% (fileserver) and 38.1% (varmail).

To further verify the effectiveness of IODP, we rerun the

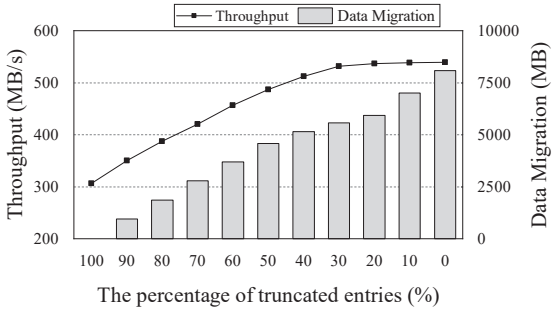


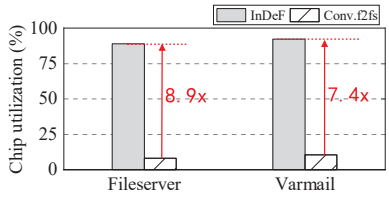
Fig. 13. Throughput and data migration under the different percentages of truncated entries. As the defragmentation percentage of truncated entries decreases, the grep throughput and the amount of data migration increase.

above experiment by changing the percentage of truncated entries of InDeF, as shown in Figure 13. When the percentage is less than 30%, there is no significant change in reading performance, but migration increases significantly. In other words, migrating about 30% of the I/O data with a low IODP value does not have a significant performance improvement. Therefore, only the I/O data with a high IODP value needs to be migrated to achieve high performance.

(2) RocksDB. To evaluate InDeF with database workloads, we run YCSB workload-C using RocksDB on ZNS SSD. We create 4,000,000 records and configure the block size of RocksDB as 128KB. Before running, we load the workload data and age the filesystem with dummy files. We set the percentage of truncated entries as 20% in this experiment. The evaluation results are not included in the form of figures due to the limited space of the paper. The Conv.f2fs migrates the entire database files, while InDeF only considers the data that has been accessed. InDeF decreases the amount of data migration by 66.7% compared with Conv.f2fs while achieving a similar level of I/O performance improvement.

C. Flash Chip Utilization and The Elapsed Time of Defragmentation

InDeF can increase chip utilization by reducing the idle intervals invoked during the host-level migration operations. We measure the flash chip utilization and the elapsed time of



Elapsed time (s)	InDeF	Conv.f2fs
Fileserver	21	360
Varmail	35	417

Fig. 14. Flash chip utilization and the elapsed time of defragmentation with different defragmenters.

defragmentation when InDeF and Conv.f2fs are run separately, as shown in Figure 14. For InDeF, the chip utilization during migration is 88.7% (fileserver) and 92.2% (varmail). For Conv.f2fs, the chip utilization during migration is only 8.9% (fileserver) and 10.9% (varmail). This is because Conv.f2fs needs to read the fragments into the host memory before writing them to the device, which can lead to a lot of fragmented I/O. Compared to Conv.f2fs, InDeF decreases the elapsed time of defragmentation time by 94.2% (fileserver) and 91.6% (varmail) due to the higher chip utilization.

D. The Impact on Co-running Applications Performance

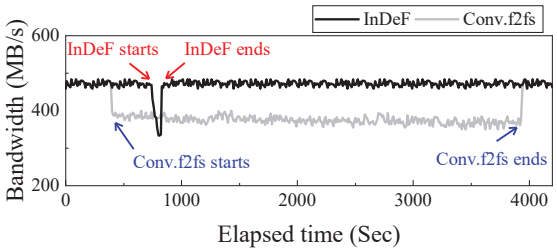


Fig. 15. The performance of fileserver with different background defragmenters.

To evaluate the impact of InDeF on co-running application performance during defragmentation, we measure the bandwidth of fileserver while running InDeF and Conv.f2fs separately. We set the amount of data migration for both InDeF and Conv.f2fs to 4GB, as shown in Figure 15. The elapsed time of defragmentation will be longer than running the defragmentation tool alone due to interference from foreground GC. Conv.f2fs takes 3599 seconds to perform defragmentation and decreases foreground application bandwidth by about 20%. This is because of Conv.f2fs issue excessive migration operations for defragmentation. Moreover, we observe that, unlike InDeF, Conv.f2fs inefficiently issues fragmented read I/Os for reading fragmented data in host-level migration, which further exacerbates the overhead. InDeF offloads data migration from the host to the device, sending only a few migration offloading commands. Although InDeF decreases workload bandwidth by about 23% during defragmentation, InDeF takes only 81 seconds, which is 97.7% less than Conv.f2fs.

E. Memory Consumption

In this section, we measure the host memory consumption when InDeF and Conv.f2fs are run separately. The results

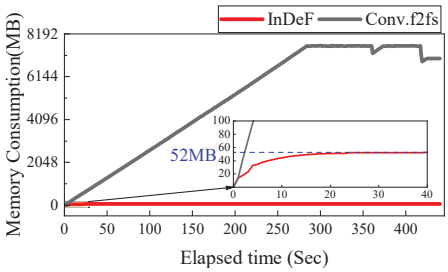


Fig. 16. The Memory usage curve with different defragmenters.

are shown in Figure 16. For Conv.f2fs, fragmented data is read into the host page cache, and memory usage grows linearly with elapsed time. After the page cache usage reaches the maximum, the host will perform memory reclamation to allocate space for subsequent read operations. InDeF avoids excessive read operations and unnecessary page cache usage by offloading data migration on the SSD, therefore the memory usage is only 52MB.

V. Related Works

Janusd [9] eliminates fragmentation by modifying the mapping table inside the SSD instead of copying the physical data. However, Janusd is not suitable for ZNS SSD because it requires a specialized remapping operation, while ZNS SSD only does zone-level coarse-grained mapping. FragPicker [18] analyzes the I/O characteristics of the application and defragments only the file fragments with high access counts. Similar to traditional tools, FragPicker still uses a host-level data migration approach and does not regard physical fragmentation.

VI. Conclusion

In this paper, we experimentally find that defragmenting data with a low degree of fragmentation or cold data provides little performance gain. With the observation, we propose InDeF. It combines the degree of fragmentation and access hotness to find out the most suitable data set to migrate in each file. By the `In_Device_Defragment` command, InDeF offloads the data migration from the host to the SSD to improve the efficiency of defragmentation. Experimental results show that InDeF decreases the elapsed time of defragmentation significantly while minimizing the amount of data migration for defragmentation. Since InDeF greatly decreases the time consumed by defragmentation, the host can schedule defragmentation around I/O flexibly to eliminate existing fragmentation and increase performance predictability.

VII. Acknowledgements

We thank the anonymous referees of ICCD 2022 for their valuable feedback and comments. Special thanks are extended to Ying Yuan and Xinyan Zhang for their contributions to this project.

References

[1] M. Bjørling, A. Aghayev, H. Holmberg *et al.*, "Zns: Avoiding the block interface tax for flash-based ssds," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 689–703.
[2] T. Stavrinos, D. S. Berger *et al.*, "Don't be a blockhead: zoned namespaces make work on conventional ssds obsolete," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2021, pp. 144–151.
[3] K. Han, H. Gwak, D. Shin, and J. Hwang, "Zns+: Advanced zoned namespace interface for supporting in-storage zone compaction," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021, pp. 147–162.
[4] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2fs: A new file system for flash storage," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015, pp. 273–286.
[5] L. Yang, Z. Tan, F. Wang *et al.*, "Improving f2fs performance in mobile devices with adaptive reserved space based on traceback," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 1, pp. 169–182, 2021.
[6] S. Kadekodi, V. Nagarajan, and G. R. Ganger, "Geriatrix: Aging what you see and what you don't see. a file system aging approach for modern storage systems," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 691–704.
[7] A. Conway, A. Bakshi, Y. Jiao *et al.*, "File systems fated for senescence? nonsense, says science!" in *15th USENIX Conference on File and Storage Technologies (FAST 17)*, 2017, pp. 45–58.
[8] Y. Liang, C. Fu, Y. Du, A. Deng, M. Zhao, L. Shi, and C. J. Xue, "An empirical study of f2fs on mobile devices," in *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2017, pp. 1–9.
[9] S. S. Hahn, S. Lee, C. Ji, L.-P. Chang, I. Yee, L. Shi, C. J. Xue, and J. Kim, "Improving file system performance of mobile storage systems using a decoupled defragmenter," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 759–771.
[10] "btrfs-filesystem," 2022, https://btrfs.readthedocs.io/en/latest/btrfs-filesystem.html.
[11] "defrag.f2fs," 2022, https://manpages.debian.org/testing/f2fs-tools/defrag.f2fs.8.en.html.
[12] "e4defrag," 2022, http://manpages.ubuntu.com/manpages/bionic/man8/e4defrag.8.html.
[13] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for ssd performance," in *2008 USENIX Annual Technical Conference (USENIX ATC 08)*, 2008.
[14] M. Jung, E. H. Wilson III *et al.*, "Physically addressed queueing (paq) improving parallelism in solid state disks," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 404–415, 2012.
[15] B. Schroeder, R. Lagisetty, and A. Merchant, "Flash reliability in production: The expected and the unexpected," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016, pp. 67–80.
[16] "How to defrag your drives the right way." 2022, https://www.auslogics.com/en/articles/how-to-defrag/.
[17] "Filebench," 2022, https://github.com/filebench/filebench/wiki.
[18] J. Park and Y. I. Eom, "Fragpicker: A new defragmentation tool for modern storage devices," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 280–294.
[19] ——, "Anti-aging lfs: Self-defragmentation with fragmentation-aware cleaning," *IEEE Access*, vol. 8, pp. 151 474–151 486, 2020.
[20] Q. Wang, J. Li, P. P. Lee, T. Ouyang, C. Shi, and L. Huang, "Separating data via block invalidation time inference for write amplification reduction in log-structured storage," in *Proc. of USENIX FAST*, 2022.
[21] Y. Lv, L. Shi *et al.*, "Access characteristic guided partition for read performance improvement on solid state drives," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.
[22] "Nvm express base specification," 2022, https://nvmexpress.org/developers/nvme-specification/.
[23] H. Li, M. Hao, M. H. Tong *et al.*, "The case of femu: Cheap, accurate, scalable and extensible flash emulator," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018, pp. 83–90.
[24] A. Conway, E. Knorr, Y. Jiao *et al.*, "Filesystem aging:it's more usage than fullness," in *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.