We Ain't Afraid of No File Fragmentation: Causes and Prevention of Its Performance Impact on Modern Flash SSDs

Yuhun Jun, *Sungkyunkwan University and Samsung Electronics Co., Ltd.;*
Shinhyun Park, *Sungkyunkwan University;* Jeong-Uk Kang, *Samsung Electronics Co., Ltd.;*
Sang-Hoon Kim, *Ajou University;* Euiseong Seo, *Sungkyunkwan University*

https://www.usenix.org/conference/fast24/presentation/jun

This paper is included in the Proceedings of the
22nd USENIX Conference on File and Storage Technologies.

February 27–29, 2024 • Santa Clara, CA, USA

978-1-939133-38-0

Open access to the Proceedings
of the 22nd USENIX Conference on
File and Storage Technologies
is sponsored by

We Ain't Afraid of No File Fragmentation: Causes and Prevention of Its Performance Impact on Modern Flash SSDs

Yuhun Jun[1,2], Shinhyun Park[3], Jeong-Uk Kang[2], Sang-Hoon Kim[4] and Euiseong Seo[*3]
[1]*Department of Semiconductor and Display Engineering, Sungkyunkwan University*
[2]*Memory Business Unit, Samsung Electronics Co. Ltd.*
[3]*Department of Computer Science and Engineering, Sungkyunkwan University*
[4]*Department of Software and Computer Engineering, Ajou University*
[*]*Corresponding Author: Euiseong Seo (euiseong@skku.edu)*

Abstract

A few studies reported that fragmentation still adversely affects the performance of flash solid-state disks (SSDs) particularly through request splitting. This research investigates the fragmentation-induced performance degradation across three levels: kernel I/O path, host-storage interface, and flash memory accesses in SSDs. Our analysis reveals that, contrary to assertions in existing literature, the primary cause of the degraded performance is not due to request splitting but stems from a significant increase in die-level collisions. In SSDs, when other writes come between writes of neighboring file blocks, the file blocks are not placed on consecutive dies, resulting in random die allocation. This randomness escalates the chances of die-level collisions, causing deteriorated read performance later. We also reveal that this may happen when a file is overwritten. To counteract this, we propose an NVMe command extension combined with a page-to-die allocation algorithm designed to ensure that contiguous blocks always land on successive dies, even in the face of file fragmentation or overwrites. Evaluations with commercial SSDs and an SSD emulator indicate that our approach effectively curtails the read performance drop arising from both fragmentation and overwrites, all without the need for defragmentation. Representatively, when a 162 MB SQLite database was fragmented into 10,011 pieces, our approach limited the performance drop to 3.5%, while the conventional system experienced a 40% decline.

1 Introduction

File system fragmentation, in which discontinuities exist between data blocks belonging to a single file, transforms sequential access to the file into a series of random accesses to scattered chunks at the storage level [35, 37]. In the era of hard disks (HDDs), which suffer from considerably long seek delays for random accesses, this resulted in additional seek operations and ended up with significantly impaired read performance [7].

To prevent performance degradation caused by fragmentation, file systems utilize various techniques [35], such as delayed allocation [23] and preallocation of data blocks [2], to maintain continuity among data blocks. Nonetheless, it is inherently challenging to avoid situations where the file system cannot locate free data blocks immediately adjacent to a file's data blocks, either due to the simultaneous writing of multiple files or appending to a file after a significant amount of time has passed since its last write.

In contrast to HDDs, flash-based solid-state disks (SSDs) eliminate mechanical movements, significantly reducing the performance gap between random and sequential accesses. However, recent studies have revealed that SSDs also experience a two to five times slower read performance when accessing fragmented files [4], prompting the development of several defragmentation schemes to address this performance decline [13, 31, 42]. However, these studies only superficially observed the performance degradation based on the fragmentation patterns and hypothesized that its primary cause is *request splitting* in the kernel I/O path due to fragmentation [13, 31].

In this paper, through a series of experiments, we reveal that the previous claim suggesting file fragmentation adversely impacts sequential read performance also in flash SSDs due to request splitting is based on inaccurate experiment settings and analyses. Moreover, we demonstrate that during file fragmentation, the page-to-die mappings within the SSD deviate from the ideal state, leading to a substantially increased number of *die-level collisions* [18] compared to the cases without file fragmentation. This increase in die-level collisions, which leads to the degradation of SSD's internal parallelism, is the primary contributing factor to the observed deterioration in read performance in an SSD with file fragmentation.

An SSD's firmware allocates its flash memory pages in a round-robin manner across the flash memory dies based on the order in which they are written. Consequently, in situations where file fragmentation occurs, the pages storing contiguous file blocks cannot be placed on contiguous dies but are instead allocated to arbitrary dies. To prevent such improper

page-to-die mapping patterns arising from file fragmentation, we propose a simple extension to the NVMe protocol that provides hints for page-to-die mapping in conjunction with a write command. With these hints, the page for an appending write is mapped to the die following the die where the previous file block's page was assigned to. In addition, the page for an overwrite operation to an existing file block, which also disrupts the page-to-die mapping pattern, is mapped to the same die where the original page was located. Through these simple hints and mapping rules, it is possible to avoid performance degradation in read operations even in situations with file fragmentation or overwrites to existing files. We evaluate the proposed approach using two configurations: first, through emulation with commercial SSDs, and second, by implementing it in the Linux kernel and NVMeVirt [22], an SSD emulator.

To the best of our knowledge, this research is the first to experimentally demonstrate that the primary cause of file fragmentation-induced performance degradation in an SSD is the deterioration of its internal parallelism. Moreover, we show that this performance degradation is not an inevitable consequence of fragmentation and can be easily avoided while keeping the fragmentation state unchanged.

The rest of this paper is organized as follows. After introducing the background and related work on the file system fragmentation in Section 2, Section 3 analyzes its impact on performance when using flash SSDs. Section 4 proposes our approach to avoid performance degradation from file fragmentation and overwrite operations, and Section 5 evaluates the proposed approach. Finally, Section 6 concludes the research.

2 Background and Motivation

2.1 Old Wisdom on File Fragmentation

In the HDD era, the primary and direct cause of performance degradation from file fragmentation was the seek time between dispersed sectors of the file [7]. File fragmentation has a more pronounced negative impact on read operations, which must wait for the completion, compared to writes that can be buffered by the storage. The long seek time of HDDs overshadowed other factors that negatively impacted performance due to file fragmentation. However, file fragmentation adversely affects performance at three levels: kernel I/O path, storage device interface, and storage media access.

As shown in Fig. 1, to the file system, a file is an array of file blocks, which are logically contiguous. However, the file system data blocks where these file blocks are actually stored may not be contiguous. Naturally, file systems strive to store contiguous file's logical blocks in contiguous file system data blocks. However, it is difficult to achieve a completely fragmentation-free data block allocation, especially when a file grows incrementally over time, as other files may be written behind the last written data block. Therefore, the data blocks of a file can be allocated in separate locations.

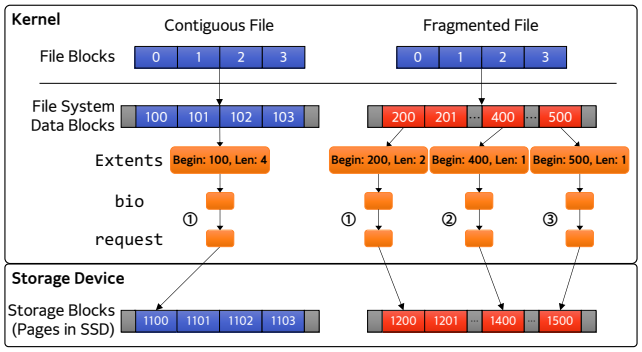


Figure 1: A sequential access to a contiguous file is translated to a single device command while that to a fragmented file ends up with multiple requests.

Only a single command is required for the host to instruct the storage device to perform read or write operations on contiguous storage space. Thus, when a sequential read occurs for a file, the Linux kernel reads the data block mapping in the file's inode, and for each contiguous data block region, it creates a `bio` (block I/O) data structure. This data structure is used to create the corresponding `request` data structure to be passed to the device driver, which then issues the command for the request to the device. Through this process, a single sequential file access may be split into multiple `bios` and corresponding requests to the storage device, depending on the degree of file fragmentation.

This request splitting is known to increase I/O execution time, as it increases the number of data structure creations and calls to underlying functions, including the device driver code [13, 16, 17, 31, 32]. Naturally, the increased number of device commands leads to time delays at the SATA [34] or NVMe [9] interface level. The increased number of storage device commands leads to an increased time for the storage device's firmware to process them. Specifically, the frequency of fetching, decoding, translating commands into storage media operations, and queuing media access operations increases. Therefore, file fragmentation also delays the processing time of the storage device controller.

Finally, file fragmentation extends the time to access storage media in the storage device. As mentioned earlier, in the case of HDDs, seek time is required for the disk head to move to the track where the requested sector is located, and a disk rotation delay occurs to locate it on the track [7]. However, unlike performance degradation caused by the kernel I/O path and storage device interface, SSDs are expected to not experience an increase in the storage medium access time due to fragmentation, as SSDs do not have seek time and rotational delay [10, 12, 39, 41].

To address the fragmentation-induced performance decline, two types of studies have been conducted: one aims to prevent fragmentation from occurring, and the other aims to recover

file access performance by transforming fragmented files into contiguous ones.

The delayed allocation technique used in the ext4 file system performs data block allocation not at the write system call handling but at the time of page flush [23]. In cases where small write operations are interleaved with writes to other files, delayed allocation increases the likelihood that write operations for a file are allocated to contiguous data blocks.

In addition, ext4 reserves a predefined window of free data blocks for each file's inode. These reserved free blocks will be actually allocated to the file for its successive append writes. This significantly reduces the occurrences of fragmentation especially when multiple files in the same directory are simultaneously written [2, 35].

While these techniques can reduce the frequency of file fragmentation, research has shown that it is an inevitable result of file system aging [5, 37]. Therefore, various defragmentation tools have been proposed to rewrite scattered file data blocks to contiguous ones to recover the I/O performance [8, 25, 27, 30, 35].

Sato proposed an online defragmentation tool for the Linux ext4 file system [35]. The proposed scheme allocates contiguous free blocks to a temporary inode, copies the fragmented file data to the temporary inode, deletes the original file, and renames the temporary inode to the original's.

Various techniques have been proposed to mitigate the overhead caused by defragmentation, as copying all fragmented files can take a significant amount of time. For example, F2FS's defragmentation tool, `defrag.f2fs`, allows users to selectively migrate only the user-selected area by manually inputting the starting block address, length, and target location as parameters [30]. XFS's `xfs_fsr` sorts files by their number of extents and groups the top 10% of files into a unit called a *pass*, performing defragmentation for each pass [27]. Btrfs's built-in defragmentation tool defragments only extents smaller than the target extent size specified as a parameter [33]. However, ultimately, defragmentation consumes a significant amount of time and energy as it induces a large number of read and write operations on relatively slow storage devices [13, 31].

2.2 File Fragmentation in SSD-Era

Most researchers and SSD manufacturers initially claimed that SSD performance is not affected by file fragmentation, and that defragmentation is unnecessary and may even be harmful due to the write operations involved in the defragmentation process, which can reduce the lifespan of the flash memory [10, 12, 39, 41]. However, contrary to initial claims that SSDs do not have fragmentation issues, some researchers observed performance degradation due to file system aging and resulting fragmentation.

SSDs offer significantly higher performance than a single flash memory die (chip) because they operate multiple flash dies in parallel [21]. Specifically, NVMe SSDs offer 65,535 command queues, each capable of queueing 65,536 commands. Even when fragmentation leads to smaller request sizes that cannot fully utilize die-level parallelism, smaller flash operations in the command queues can still be processed out-of-order, allowing most dies to be fully utilized. This enables SSDs to achieve performance close to their maximum potential even when accessing small fragments. Consequently, some researchers speculated that the kernel I/O path and interface overhead due to request splitting have a greater impact on fragmentation-induced performance degradation than flash memory access time [13, 16, 17, 31, 32].

Conway et al. empirically observed performance degradation in various workloads due to file system aging on SSDs [4,5]. They discovered that file fragmentation frequently occurs on SSDs as the file system ages. In scenarios where the `git` pull commands are executed repeatedly, they observed that read performance can be degraded by up to five times. Geriatrix is a tool capable of effectively emulating file system and storage aging [20]. Using Geriatrix, the authors demonstrated a performance degradation of up to 78% due to file system aging on SSDs. While both studies observed changes in file system layout and performance degradation due to file system aging in various circumstances, they did not conduct an in-depth analysis of the underlying causes for this performance decline.

Park and Eom argued that the main cause of performance degradation due to fragmentation on SSDs is request splitting, and thus the distance between data blocks does not significantly affect read performance, while the degree of fragmentation does [31]. In a subsequent paper, they made a contradictory claim, stating that the distance between fragmented blocks also significantly affects performance on SSDs [32]. In addition, they proposed FragPicker, an efficient defragmentation approach that carries out online migration of fragments only that have been accessed [31].

Zhu et al. proposed a scheme that can simultaneously issue parallel I/O requests for defragmentation in ext4, minimizing defragmentation time and maximizing SSD internal parallelism. This approach improved defragmentation time by three times compared to the traditional `e4defrag` [42].

Regarding these conflicting claims, we clarify in Section 3 that this arises because previous studies' experimental setups fail to distinguish between performance degradation directly caused by fragmentation and that indirectly caused by the influence of fragmentation on SSD's internal data placement.

2.3 Internals of Modern Flash SSDs

As previously mentioned, a modern flash SSD is equipped with multiple flash dies that can operate in parallel. To maximize the bandwidth and throughput of an SSD, it is crucial to maintain a high degree of die-level parallelism [3, 21]. Conversely, since a die can only process one request at a

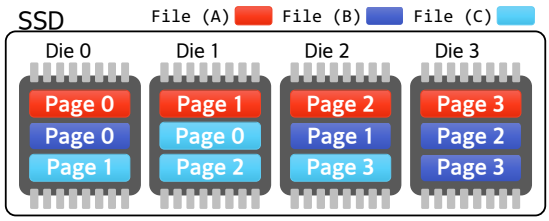


Figure 2: Data placement of three files in a flash SSD where one is contiguous and the other two are fragmented.

time [14, 40], if the pages to be read are stored on a single die, the read requests for these pages must be serialized within that die. This die-level read collision significantly degrades read performance [18].

To prevent die-level collisions for read operations, the flash translation layer (FTL) of an SSD's firmware must perform physical page allocation in a manner that distributes the physical pages storing contiguous logical pages across as many dies as possible. For this purpose, the FTL of most modern SSDs selects a die in a round-robin manner when allocating a flash page for processing an incoming page write request [3, 19]. Additionally, modern FTLs perform the valid page copy within the die where the page resides during the garbage collection (GC) process if the die has a sufficient number of free pages [11]. This allows for the maintenance of die parallelism. However, since GC occurs in parallel across all dies, this strategy does not significantly impact the performance of GC.

For example, in Fig. 2, File A is evenly distributed across four dies since its four pages were written without interference. Thus, a sequential read of File A will be performed simultaneously on these four dies, resulting in a bandwidth of up to four times the flash die performance. In contrast, assume that the writes to File B and File C were interleaved. As the die for storing a logical page is assigned in a round-robin manner according to the order of writes performed within the SSD, both the third and last pages of File B ended up being allocated to Die 3. As a result, the time to read File B is twice as long as that for reading an ideally-placed file of the same size, such as File A.

File fragmentation occurs in most cases when multiple files are simultaneously written [4, 31]. Therefore, when file fragmentation occurs, the die allocation of flash pages associated with a file might not be evenly distributed, leading to the pages of a single file being consolidated on certain dies. This phenomenon arises because the FTL allocates dies for pages solely based on their incoming order. However, the presence of file fragmentation does not inevitably result in uneven page distribution over dies, just as a contiguous file does not guarantee that its pages will always be evenly and sequentially allocated on consecutive dies.

Through ext4's preallocation, data blocks can be allocated contiguously in the file system, even if writes from other files occur in between. However, since the die mapping of flash pages takes place *at the actual moment of their writing inside the SSD*, even files that are contiguous at the file system level can exhibit uneven page distribution in the SSD. Conversely, if the data blocks of a fragmented file are written at the appropriate timing, it is possible for the file's pages to be distributed evenly across all dies.

In addition to the fragmentation cases, irregular die allocation may occur in cases of file overwrites. Assume a file stored in contiguous file system blocks has its pages sequentially allocated to dies on the SSD. In this ideal situation, if an overwrite is performed on a middle block of the file, the SSD must allocate a new page for that block and invalidate the page currently mapped to the block due to the nature of flash memory, which does not allow in-place updates. At this point, the new page will be allocated from the die next to the one that last allocated a page, according to the round-robin policy. As a result, there is a high likelihood that the new page will not be located on the same die as the original page, leading to a considerable decline in performance due to die-level collisions when conducting a sequential read on the file.

Table 1: System configurations for experiments.

Processor	Intel Xeon Gold 6138 2.0 GHz, 160-Core
Chipset	Intel C621
Memory	DDR4 2666 MHz, 32 GB x16
OS	Ubuntu 20.04 Server (kernel v5.15.0)
Interface	PCIe Gen 3 x4 and SATA 3.0
Storage	NVMe-A: Samsung 980 PRO 1 TB NVMe-B: WD Black SN850 1 TB NVMe-C: SK Hynix Platinum P41 1 TB NVMe-D: Crucial P5 Plus 1 TB
	SATA-A: Samsung 870 EVO 500 GB SATA-B: WD Blue SA510 500 GB

3 Analysis of File Fragmentation

This section explores the cause behind fragmentation-induced performance degradation through a series of experiments. The configuration of the experimental system for our analysis is described in Table 1. We used the ext4 file system [28]. To minimize the influence of the kernel's page cache and extent cache on the experimental results, we performed a cache drop before each experiment run. In addition, to adjust the block I/O request queue depth for our experiments, we used the kernel's `nr_requests` parameter. All experimental results are an average of 10 repetitions.

To begin, we examined the performance drop in ext4 on NVMe SSDs based on the degree of fragmentation (DoF), which is the ratio of the actual number of extents to the ideal number of extents [17]. For this, we created a set of files that have various DoF. Each fragmented file, with a size of 8 MB, is created by interleaving the writes to the target file and that to a dummy file as many times as the desired DoF. The size of

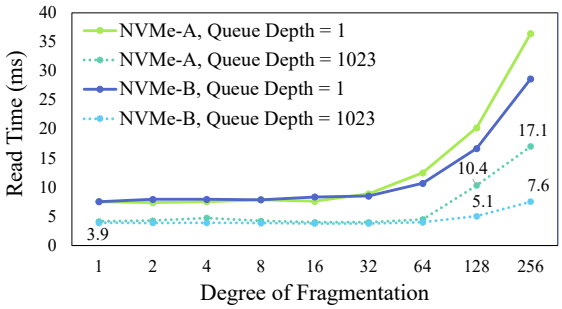


Figure 3: Time taken for reading an 8 MB file stored on NVMe SSDs while varying its DoF.

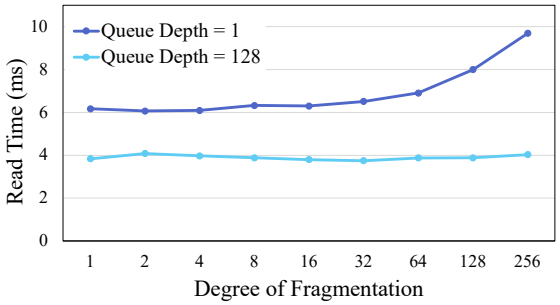


Figure 4: Time for sequentially reading an 8 MB file stored on `ramdisk` depending on its DoF.

the write to the dummy file between the writes to the target file was determined so that the offset between the two fragments of the target file becomes 8 MB. For example, if the target DoF is 4, four fragments or extents must compose the 8 MB target file. We wrote the first quarter of the target file first and then wrote 6 MB for the dummy file. By repeating this four times, we can obtain an 8MB file with a DoF of 4.

We varied the DoF in our analysis from 1, representing contiguous files, to 256, unlike previous studies that went beyond this range. A fragment size when the DoF is 256 in our analysis is 32 KB. Due to the aforementioned delayed allocation and block reservation techniques, which are used by ext4 to suppress fragmentation, it is highly unlikely for a fragment to have a smaller size than that.

In order to create a file exactly with the desired DoF using this method, it is necessary to disable delayed allocation and block reservation. To disable delayed allocation, we used the `direct` mode when writing files and provided the `nodelalloc` mount option. Block reservation was neutralized by setting both the `reserved_clusters` runtime parameter and the percentage of reserved block parameter, which is `mkfs`'s, to 0. We also disabled the per-inode preallocation feature. Every fragmented file was verified whether it has the desired DoF by using the `filefrag` tool [26]. A single extent in ext4 can represent 2^{15} contiguous blocks, or 128 MB, with a 4 KB file system block size [28]. Since the fragments were all smaller than 128 MB, the desired DoF value precisely matched the number of extents constituting the file.

As shown in Fig. 3, the read performance of both NVMe SSDs decreased from the point where the DoF exceeded 64, regardless of the request queue depth. Since the default maximum request size for the Linux kernel is 1 MB, no performance difference was observed when the fragment size exceeded 1 MB, corresponding to situations where the DoF is 8 or lower.

Notably, both SSDs showed a more drastic performance change when the queue depth was set to 1. When the I/O queue depth was set to 1023, which is the Linux default value for an NVMe SSD, the execution time was shorter, and the performance degradation due to fragmentation was less pronounced compared to when the queue depth was set to 1. However, even at a queue depth of 1023, the execution time increased significantly with the increase in DoF. NVMe-A exhibited 2.7 times and 4.4 times longer execution times at DoF 128 and 256, respectively, compared to DoF 1. Similarly, NVMe-B demonstrated 1.3 times and 1.9 times longer execution times at DoF 128 and 256, respectively.

In this experiment, we have confirmed that file fragmentation indeed causes performance degradation in SSDs. To further elucidate the specific causes of this performance degradation, subsequent experiments were conducted.

3.1 Impact Caused by Request Splitting

As previously mentioned, file fragmentation results in request splitting, where a single I/O operation is translated into multiple device commands. The impact of increased processing time at the host side due to request splitting on performance degradation will be more evident when the storage device's processing time is shorter. Therefore, we measured the delay occurring in the kernel I/O path due to request splitting by using a *ramdisk* as the storage device, which has an extremely short host-to-storage interface and storage media access times.

Fig. 4 shows the sequential read performance of files stored on the ramdisk according to their DoF. Since the ramdisk has extremely fast access speed and there is negligible difference between random and sequential access times, the performance changes observed in Fig. 4 can be attributed primarily to the difference in time consumed by the kernel I/O path rather than the storage media. Our experiments revealed that as the DoF increased with the request queue depth set to 1, read performance decreased, resulting in a 1.5-fold increase in read time when the DoF was 256.

As stated, one request to the storage device is generated for each contiguous storage address range. Consequently, the numbers of `iomap` structures, which are required for direct I/O, `bio` structures, and `request` structures increased with the DoF. Additionally, the number of function invocations for their creation also rose as the DoF grew. When the queue depth was set to 1, these procedures were performed syn-

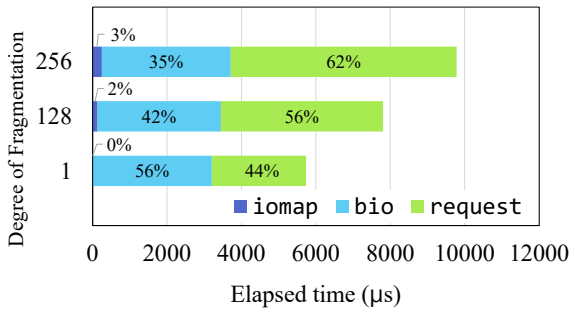


Figure 5: Time composition for creating request data structures in the kernel I/O path depending on File's DoF.

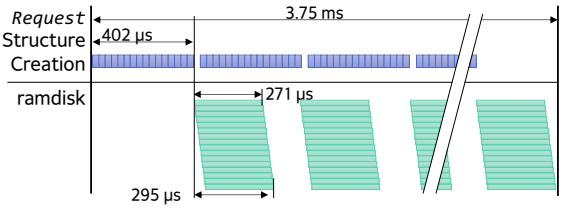


Figure 6: Reduction of read time due to the overlap of storage operations and `request` creation when File's DoF is 128.

chronously. As a result, the processing time for the kernel I/O path increased with the rise of DoF, and this increase became more pronounced when the DoF exceeded 8.

To identify the cause of these delays, we measured the time taken for the `__x64_sys_read` function, which processes the read system call, to create `iomap`, `bio`, and `request` structures separately, as the DoF changes. As seen in Fig. 5, the `iomap` creation time slightly increased in proportion to the DoF, as one `iomap` is created per extent. The change in time spent on `bio` creation was smaller compared to that of `iomap` creation. This is because most of the `bio` creation time was spent allocating buffer pages, and the number of buffer pages to be allocated remains constant at 2048, regardless of the file's DoF. During the `request` creation process, if there are consecutive `bio` addresses, they will be merged into a single `request`. However, the fragmented file prevented it from being merged. Thus, the time spent on creating `request`s increased proportionally with the increase in the DoF.

Note that even under the extreme case where the DoF was 256, the kernel I/O path only took approximately 9.7 ms. In addition, when the I/O queue allows queueing of multiple outstanding commands, this I/O path delay can be mostly overlapped by the consecutive read operations to the following fragments. As shown in Fig. 4, if the `request` I/O queue depth was set to 128, which is the default value for a ramdisk, the file's DoF barely affected the time for the read operation.

We closely observed the kernel I/O path delay, which can be masked by I/O queueing. Fig. 6 shows the time spent on the `request` data structure creation and ramdisk access measured with `blktrace` [24] when reading a file with a DoF of 128 and a queue depth of 128. The kernel performs a *plug* process to merge `request`s for contiguous blocks, reducing the number of commands issued to the storage. The plugged `request`s are unplugged and sent to the device driver if the number of `request`s exceeds the predefined maximum pluggable `request`s, or if the size of an individual `request` surpasses the predefined plug flush size. The default values of these parameters are 16 `request`s and 128 KB, respectively.

Thus, during the experiment, 16 `request`s were plugged and then separately issued to the device driver since they all accessed separate blocks. As a result, the time spent creating the following 16 `request`s in the kernel I/O path is mostly masked by the time it takes the ramdisk to process the previous 16 `request`s. Additionally, by issuing multiple `request`s simultaneously, the processing time of the ramdisk is significantly reduced due to the operation overlap.

As a result of these experiments, we found that the delay occurring in the kernel I/O path due to request splitting is at the level of a few milliseconds, even in the extreme cases. Furthermore, we confirmed that its impact on actual execution time is negligible due to I/O operation queueing.

Next, we analyzed the execution time delay caused by request splitting in both the host-to-storage interface and the SSD inside. Since the implementation within the SSD is a black box, and it is impossible to accurately distinguish between the time consumed by the interface and the flash memory access time, we analyzed their combined execution time. For this analysis, we used two types of SATA SSDs and two types of NVMe SSDs, as shown in Table 1.

For this analysis, we performed a task to read 8 MB of contiguous data from the storage device by accessing the raw device file of the SSD to exclude the influence of the file system and kernel I/O path. In this process, we measured performance while increasing the unit read size from 32 KB to 8192 KB, doubling it each time. To minimize the impact of the SSD's state on the results, such as the ratio of invalid pages and the number of free blocks, we used the trim command for the entire area after each experiment to restore the SSD to the fresh-out-of-the-box (FOB) state. Then, we performed a sequential write on a 1 GB area to be read.

Fig. 7 shows the time taken to read 8 MB of data from each of the four SSDs, depending on the unit size of the read operation. Similar to Fig. 4, when the device's queue depth is set to 1, the elapsed time for reading 8 MB of data increases as the size of the read unit decreases. According to the regression analysis of the results, for NVMe SSDs, the elapsed time increased by 85 µs for each additional request, while for SATA SSDs, the time increased by 136 µs per request. These results encompass the impact of request splitting on the host-to-device interface, SSD firmware, and flash memory access time. Among these factors, the flash memory access time is expected to decrease as the unit read operation size increases influenced by the aforementioned internal parallelism.

However, the read time delay caused by request splitting

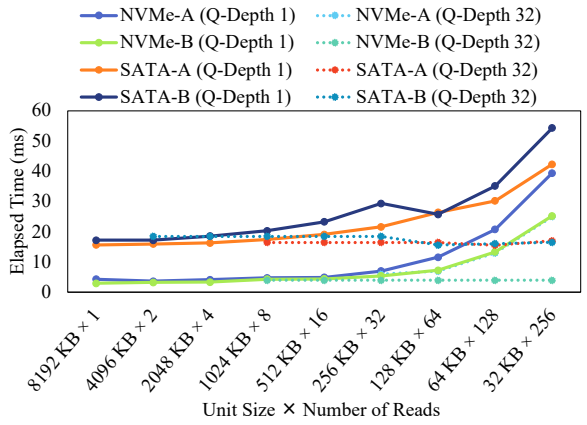


Figure 7: Time for reading 8 MB of data through raw device I/O operations with varying unit sizes.

disappeared when multiple outstanding I/O operations were queued, similar to the effect in the kernel I/O path. The native command queue (NCQ) of the SATA standard can queue up to 32 outstanding commands, while the NVMe standard also supports 65,536 queues with a queue depth of 65,535. Therefore, when request splitting occurs, the SSD can simultaneously place the split requests into the command queue and process them out of order. This reduces the number of interactions at the host-to-device interface and can further increase the die-level parallelism of the SSD. The dashed lines in Fig. 7 show the results when the command queue depth was set to 32. As expected, despite reducing the read unit size and, consequently, increasing the number of read commands in all SSDs, the difference in the total execution time was observed to be within a few milliseconds.

Based on our analyses, we confirmed that the request splitting overhead in the kernel I/O path is negligible compared to the increased operation time due to fragmentation. Furthermore, its impact is largely mitigated when issuing I/O operations asynchronously through command queueing. Additionally, we verified that even when request splitting occurs, the increase in processing time both at the host-to-device interface and within the storage device itself is extremely minimal, again thanks to command queueing, which most modern SSDs support.

3.2 Page Misalignment from Fragmentation

As explained with Fig. 2, when a file is written sequentially and there are no interrupting writes between the sequential write operations, the SSD evenly distributes the pages of the file across all dies in a round-robin manner. However, in cases of file fragmentation, such an ideal page allocation becomes impossible because writes to other files have occurred in between the writes to the fragmented file.

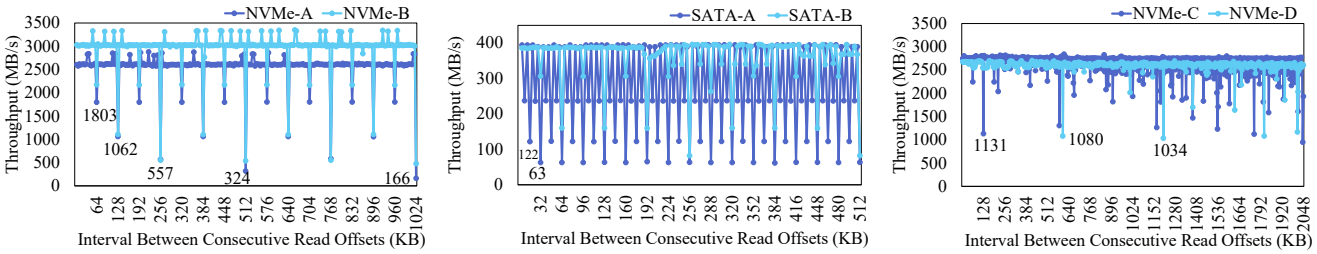
In cases of fragmentation, the page after the discontinuity point will be placed on a random die, regardless of the die where its semantically preceding page is located. In modern SSDs, because they have several tens of dies, the likelihood of a page containing a fragmented block being placed on a die immediately adjacent to the die where the previous file block's page is located is significantly low. As a result, when performing a sequential read access on a fragmented file, it causes significantly more die-level collisions compared to an ideal page placement scenario. The experiments in Fig. 3 created fragmented files and read them in such a way. Most previous research also fragmented files in the similar way. As a result, the significant performance degradation observed in fragmented file accesses in the experiments was very likely due to die-level collisions.

The read patterns observed at the die-level in these experiments can be emulated in actual SSDs by reading consecutively written file blocks at specific intervals. For instance, consider an SSD that assigns 4 KB pages to its dies in a round-robin manner and suppose this SSD has 16 dies. If we were to write 1 MB of data, that is, 256 pages consecutively, and then read every second page, resulting in reading 128 pages, this situation would produce a die-level read pattern similar to our experimental setup for sequentially accessing a 512 KB file with a DoF of 128. In this situation, compared to reading 128 consecutive pages without any interval, read operations would only take place on a half of the die set. This would inevitably lead to double the die-level collisions, making the time to read the 128 pages nearly twice as long. For the same reasons, reading every fourth page, amounting to 64 pages in total, would result in a read duration nearly four times longer than reading 64 consecutive pages.

To examine how read performance changes in such patterns, we conducted the following experiments. After initializing an SSD to its FOB state, we sequentially wrote 1 GB data to the area designated for reading. Subsequently, we configured fio to sequentially read 4 KB chunks at consistent intervals. For instance, if the interval of the read starting point were set to 16 KB, it would be set up to read 4 KB, skip a gap of 12 KB, and then read another 4 KB. To accomplish this, we modified the `blockalign` parameter of fio, incrementing it in 4 KB steps, ranging from a minimum of 4 KB to a maximum of 1024 KB in multiples. In these experiments, for NVMe SSDs, we set the *iodepth* parameter of fio to 512, and for SATA SSDs, we set it to the maximum supported value of 32. Furthermore, to exclude the effects of the file system and kernel I/O path, we configured fio to perform direct access on the raw device file.

Fig. 8a displays the throughput measurements for two NVMe SSDs when varying the read interval. When internal parallelism was adequately utilized, the SSDs achieved throughputs of 2600 MB/s and 3020 MB/s, respectively. However, as the interval between read operations expanded, the observed sustained throughput decreased to 166 MB/s and 480 MB/s for each SSD, respectively.

In both NVMe SSDs, the first significant performance drop was observed when the interval reached 64 KB. This indicates



(a) NVMe-A/B (b) SATA (c) NVMe-C/D

Figure 8: Throughput while varying the interval between starting points of consecutive read operations.

that the two NVMe SSDs allocate pages of a 32 KB size to a die before proceeding with allocation on the subsequent die, even though the actual number of pages allocated to a die at once might vary based on the device's page size. We refer to the size of the pages allocated to a single die at a given instance as the *die allocation granularity*. Both NVMe SSDs use a 16 KB page size [15,36], and a die allocation granularity of 32 KB means that they allocate two pages per die. This suggests that both SSDs store two pages per die using the two-plane program method [40].

The alignment size exhibited a significant performance drop starting at the 64 KB interval, and as it doubled each time, the decrease in performance became even more noticeable. This was likely due to the number of dies used for reads being halved every time the interval size doubled, as previously mentioned. In fact, for both SSDs, when the alignment size doubled from 64 KB to 1024 KB, the throughput decreased by 41 to 49% each time.

The lowest performance for both products was observed when the alignment size was 1024 KB for NVMe-A, dropping to approximately 6.5% of its typical value, and at 256 KB for NVMe-B, decreasing to 18.5%. These observations suggest that the *stripe size*, which represents the volume of data written across all dies before the allocation process restarts with the first die, differs among SSDs. Our experiments infer that the stripe size for NVMe-A is 1 MB, while for NVMe-B, it stands at 256 KB. For alignment sizes that exceed the stripe size, performance will mirror that of an alignment size equal to *(alignment size % stripe size)*.

This phenomenon was also observed in the SATA SSDs, as illustrated in Fig. 8b. While both products exhibited a throughput of 400 MB/s when all pages were accessible, the throughput decreased with the variation in the alignment size of accessible pages: dropping to 62 MB/s for SATA-A and 82 MB/s for SATA-B.

The performance degradation points of SATA SSDs showed a significant difference compared to those of NVMe SSDs, with SATA-A exhibiting its first performance drop at an alignment size of 8 KB. This indicates that its die allocation granularity is 4 KB. The most significant performance drop occurred at 32 KB, pointing to a stripe size of 32 KB. For SATA-B, the first performance drop was observed when the interval reached 32 KB, and, at 256 KB, it showed only 20.7% of its normal throughput. Therefore, SATA-B has a die allocation granularity of 16 KB, and the stripe size is estimated to be 256 KB.

When accounting for file fragmentation, the spacing between two accessed blocks typically aligns with multiples of the file block size, which is usually 4 or 8 KB. As evidenced in SATA SSDs, there's a marked performance dip when reading with intervals that are multiples of 4 KB. Consequently, the uptick in die-level collisions due to file fragmentation and the subsequent performance reduction are unavoidable. While NVMe SSDs generally have larger die allocation granularity and stripe sizes compared to SATA SSDs, leading to less pronounced performance drops with small read intervals, they are not exempt from the heightened die-level collisions brought about by the gap-reading patterns.

However, not all SSDs exhibited performance degradation at consistent intervals, as observed with the previous four products. Fig. 8c shows the performance drop in relation to the read offset intervals for NVMe-C and NVMe-D, respectively. Unlike the previous SSDs, the intervals at which these two products showed a decline were not necessarily powers of two. For NVMe-C, performance dips were noted at 64 KB and 128 KB intervals while the subsequent drops were found at multiples of 584 KB. In the case of NVMe-D, the drop was observed at intervals that are multiples of 604 KB. The die allocation policy of an SSD varies across manufacturers. However, the experimental results confirmed that non-sequential page access eventually leads to significant performance reduction due to high die-level collisions.

Unlike I/O path overhead or interface overhead that can be hidden by increasing the I/O queue depth, the read performance degradation due to die-level collisions was shown to persist even when the I/O queue depth was large. Therefore, for SATA SSDs with Linux kernel's default queue depth of 64 and NVMe SSDs with a queue depth of 1023, we can conclude that the main cause of performance loss due to file fragmentation is not the delay in the kernel I/O path or interface overhead but rather die-level collisions inside SSDs. In other words, while file fragmentation in HDDs causes additional seek time and rotational delay, in SSDs, it leads to additional die-level collisions.

4 Our Approach

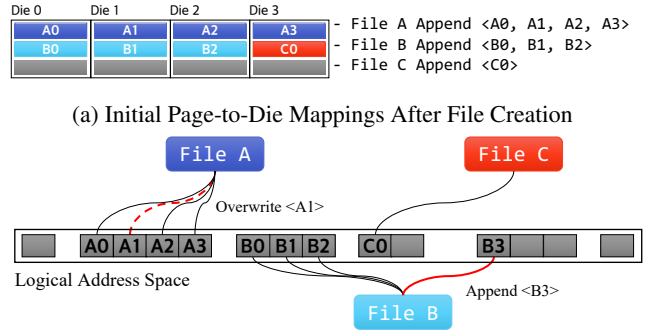
As previously analyzed, performance degradation of read operations to fragmented files mainly results from an increase in die-level collisions. However, the irregular page-to-die mapping observed in fragmented files is merely a consequence derived from the situations that cause fragmentation, rather than being a necessary condition for fragmentation to occur.

For example, let's assume that the three blocks of File B are written as in Fig. 9a and File C and A write a single block, respectively. From this initial state, if the application appends B3 to File B, B3 will be stored on the same die as B1, as shown in Fig. 9c. Consequently, a sequential read of File B will cause a die-level collision at Die 1. However, if File A overwrites not only A1 but also A2 and A3 in the situation shown in Fig. 9a, the position of B3 will shift by two dies and be located in Die 3. In this case, File B is stored in non-contiguous blocks on the file system, as shown in Fig. 9b, and sequential access to this file will cause request splitting. However, due to command queueing inside the SSD, all dies simultaneously process the same number of operations, enabling maximum performance. As previously mentioned, the time delay in the kernel I/O path and host-to-storage interface resulting from request splitting is minimal; consequently, despite File B being fragmented, its read performance remains barely affected.

Conversely, irregular page-to-die mapping may occur even without file fragmentation. Typically, overwriting an existing file block likely breaks the sequentiality of the page-to-die mapping. For instance, consider overwriting A1 in the situation shown in Fig. 9a. The file system supports in-place updates of blocks, so the position of A1 on the file system remains unchanged. Thus, File A maintains its contiguous state even after overwriting A1. However, since in-place updates of flash pages are impossible, the original page storing the A1 block becomes invalidated, and as shown in Fig. 9c, a new page for the updated A1 is assigned to Die 0, which follows Die 3. Consequently, although File A is contiguous at the file system level, a sequential read of File A will be significantly slowed down due to the die-level collision at Die 0.

Examining the two cases of fragmentation and overwriting that cause the irregular page-to-die mapping mentioned above, the fundamental reason is that SSD firmware cannot discern the file-level relationship between flash pages, and conversely, the file system cannot specify the position of the flash page storing the file block. To address this mismatch between page and file block placement, we propose an NVMe command extension and corresponding page-to-die mapping policy.

In our approach, the file system regards a write operation requiring a new data block allocation as an append write and the one to be performed on a data block already allocated to a file as an overwrite. If the Kernel I/O stack identifies the write being issued to the NVMe SSD as an append write or overwrite, it conveys additional information to the NVMe, on top of the existing NVMe write command, to perform



(a) Initial Page-to-Die Mappings After File Creation

(b) Files' Data Block Placement in File System

(c) Page-to-Die Mappings After Write Operations

Figure 9: File system-level block placement and storage-level page allocation of three files before and after write operations.

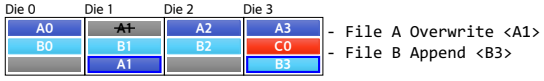


Figure 10: Page-to-die mappings after overwriting A1 and appending B3 blocks under our approach.

appropriate page-to-die mapping.

For append writes, the host provides the NVMe with the logical block address (LBA) of the file block immediately preceding the one being written, in addition to the write command. For example, when appending B3 to File B in Fig. 9a, the host sends the LBA of B2 along with the write command for B3 to the NVMe. In this case, the NVMe firmware deviates from the conventional round-robin algorithm for determining B3's die placement. Instead, as illustrated in Fig. 10, it assigns B3 to Die 3, which is the subsequent die after the one where B2 was stored. If the size of the write operation surpasses the die allocation granularity, the placement of additional pages adheres to the conventional round-robin approach; for instance, in this example, the second page is assigned to Die 0 after the first page is placed in Die 3.

For overwrites, the host sets a flag in the write command to indicate that the write operation is for overwriting an existing file block. For a write command with its overwrite flag set, the SSD firmware invalidates the existing flash page corresponding to the given LBA and allocates a new page. By assigning the new page to the same die where the original flash page was located, the die-level contiguity of the file blocks can be preserved. For example, when overwriting the A1 block of File A in Fig. 9a, a new flash page is allocated to Die 1, where the flash page storing A1 was originally located, as shown in Fig. 10, ensuring that sequential reads of File A maintain maximum internal parallelism. The die-level contiguity can also be preserved for overwrites exceeding the die allocation granularity by assigning new pages to the same dies where

the existing logical pages are located.

To implement the proposed approach, the host needs to provide additional information to the SSD when issuing a write operation. We can implement this without additional protocol overhead by utilizing unused bits in the NVMe protocol's write command. For example, the 24th and 25th bits of Command Dword (CDW) 12, which are currently unused and reserved, can be utilized to distinguish append writes and overwrites from conventional writes. Additionally, for append writes, the reserved CDW 2 and CDW 3 can be used to convey the LBA of the preceding file block.

Our approach specifically determines only the starting die for append writes, with subsequent writes following the existing mapping policy and being distributed across dies in a round-robin fashion. Consequently, it does not impact the performance of append writes. While it might be assumed that repetitive small-sized overwrites to the same file block could lead to write die collisions, these are typically merged in the host buffer and infrequently flushed to the SSD. Thus, even in these extremely rare cases, our approach does not adversely affect write performance.

Yet, continual overwrites on a small number of file blocks can quickly deplete free pages in certain dies, triggering GC earlier in these dies. Simultaneously, these overwrites invalidate the overwritten pages, reducing the number of valid pages. This decrease in valid pages necessitates fewer valid page copies during GC of those dies, which not only lowers the write amplification factor but also shortens the duration of the GC process.

However, despite these conditions being rare, they are not ideal, as they can lead to more frequent GCs in specific dies and cause uneven wear across the dies. This uneven wear might result in some dies wearing out prematurely, ultimately shortening the lifespan of the SSD. The LBAs that are targets of the frequent overwrites are likely to be evenly distributed and allocated across multiple dies, minimizing the occurrence of uneven wear. Nonetheless, should wear disparity become significant, a mechanism to reallocate the page for that specific LBA to a different die for wear leveling would mitigate the situation, albeit potentially at the expense of performance.

5 Evaluation

To assess the validity and efficacy of the proposed approach, we carried out two evaluations. First, to validate the proposed scheme, we emulated the write patterns as if the proposed approach were applied in commodity SSDs, and measured the read performance. Second, to examine the performance benefits that applications can gain through the proposed scheme, we implemented it in the ext4 file system and the NVMeVirt SSD emulator.

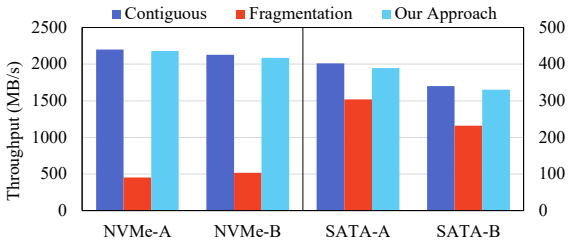
5.1 Validation of Our Approach

To implement the proposed scheme, the SSD's NVMe protocol stack must be modified to process the NVMe command extension, and its page-to-die mapping policy should also be adjusted to utilize the hints provided by the host through the command extension. However, modifying actual SSD firmware is not feasible. Therefore, to verify the validity of our approach, we created write patterns that would result in the same page-to-die mapping as the proposed approach under file fragmentation and partial file overwrite situations. We then measured the read performance of the files written in this manner. For these experiments, we deferred file system metadata writes to prevent them from interfering with die allocation control and configured journaling to be performed on a separate storage device. In these experiments, we used NVMe-A, NVMe-B, SATA-A, and SATA-B, all of which have regular die allocation granularity and stripe size, as depicted in Fig. 8.

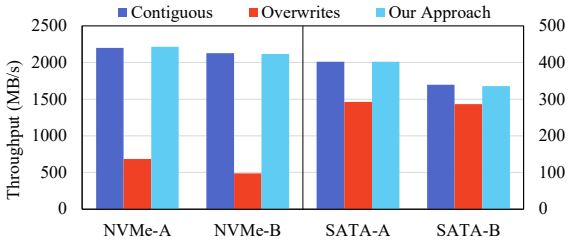
To evaluate the effectiveness of our proposed method under fragmentation, we generated a fragmented file on an FOB-state SSD and measured its read time. This file was formed by appending 256 fragments, each sized according to the SSD's die allocation granularity, cumulating to an 8 MB file. After writing each fragment of the die granularity size, if we write enough data to the dummy file to fill the remaining space of the SSD's stripe before writing the next fragment, all the fragments of the target file will be assigned to one die, resulting in significant performance degradation, which is denoted as *Fragmentation* in Fig. 11. Due to the smaller die allocation granularities of the SATA SSDs, 256 appends were insufficient to reach the desired 8 MB file size. To compensate, we adjusted the final append's size to ensure the total file size was 8 MB. Note that, as a result, only the initial segment of the file became fragmented in the SATA SSDs.

In order to emulate the proposed approach, we first wrote a single fragment and then wrote an amount of garbage data equal to the SSD's stripe size to a dummy file. Subsequently, we wrote the next fragment to the target file. Repeating this process, as shown in Fig. 10, each fragment of the target file would be located in the die immediately following the die where the previous fragment was located. Thus, while fragmentation occurs at the file system level, within the SSD, flash pages would be sequentially assigned to consecutive dies. We repeatedly read the written fragmented file while measuring the throughput.

As illustrated in Fig. 11a, the read performance of fragmented files on NVMe SSDs degraded by 79% for NVMe-A and 76% for NVMe-B, in comparison to that of contiguous files. Since the file was appended 256 times in 32 KB sizes on the NVMe SSDs, it was stored entirely on a single die. This led to die-level collisions during most read operations. We believe that NVMe-A's larger performance decrease was attributed to it having more dies than NVMe-B. Our approach



(a) Append Write



(b) Overwrite

Figure 11: Read performance of four kinds of SSDs for contiguous files, fragmented files, and fragmented files under our proposed approach, respectively.

mitigated this, with performance experiencing only a 2% decline from that of the contiguous files.

In the case of the SATA SSDs, the performance degradation due to fragmentation was less severe than that of NVMe because only the frontal part of the file was fragmented due to their smaller die allocation granularity size. Although the portions of the target files that were fragmented amounted to 12.5% for SATA-A and 50% for SATA-B, the performance experienced a degradation of 27% and 16%, respectively, when compared to the contiguous cases. In both products, our approach reduced the performance degradation, achieving nearly the same performance as accessing contiguous files, with only a 1.2% difference.

To understand the misalignment in page-to-die mapping caused by overwrites on a file, and the resultant performance degradation from die-level collisions, we conducted experiments on the four types of SSDs. First, we created a file. Then, we performed 256 overwrites, each of 32 KB, from the beginning to the end of the file. Finally, we read the file. The size of the target file was again set to 8 MB. Between consecutive overwrite operations, we wrote random data as large as (stripe size - die allocation granularity).

After finishing the series of overwrite operations, the file's pages would be placed in a single die, which is denoted as *Overwrites* on the graph in Fig. 11. Note that the file's data blocks will remain contiguous at the file system level even after the overwrites are performed.

Fig. 11b shows the results for the overwrite experiments. The performance degradation of the NVMe SSD was similar to that of a fragmented file, showing a significant performance drop to a quarter. However, when our approach was applied, the performance degradation was reduced to an average of 1% compared to the contiguous case. In the case of SATA SSDs, their performance degradation was smaller due to the difference in die allocation granularity mentioned earlier, but they still showed 27% and 16% decrease in performance, respectively. However, our approach was able to successfully achieve a similar level of performance as before the overwrites were executed. The efficacy of our approach was observed for all four SSDs. The largest performance degradation under our approach was merely 1.2% for SATA-B.

From this analysis, we confirmed that the proposed approach can effectively prevent the loss of read performance even for heavily fragmented files and also successfully avoid read performance degradation caused by overwrites.

Table 2: Parameters used for NVMe emulation.

SSD	Capacity	60 GB
	Host Interface	PCIe Gen3 ×4
	FTL L2P Mapping	Page Mapping [1, 6]
	Channel Count	4
	Dies per Channel	2
Flash Memory [22]	Read/Write Unit Size	32 KB
	Read Time	36 µs
	Write Time	185 µs
	Channel Speed	800 Mbps

5.2 Effectiveness for Application Workloads

To evaluate the holistic effectiveness of the proposed approach, we implemented the host-side part of the proposed scheme both in the ext4 file system and the Linux kernel's NVMe device driver.[1] This allowed applications to directly utilize our approach via the file system. On the SSD side, we implemented our proposed approach's write command extension and page-to-die allocation mechanism within NVMeVirt.[2] The parameters for NVMeVirt were sourced from Table 2. The die allocation granularity for the emulated SSD was set at 32 KB, and the stripe was set to 256 KB, which mirrors the settings of NVMe-B. To mitigate the onset of fragmentation, the ext4 file system was adjusted in accordance with the experimental configurations delineated in Section 3.

First, we executed experiments based on the configuration depicted in Fig. 11 using the aforementioned implementation. In contrast to prior experiments using actual SSDs that required meticulous control over dummy write sizes, the implementation can sustain optimal die mapping even when random offsets interleave between successive file block writings. This enabled us to extend our analysis beyond just the worst-case conditions, incorporating cases more reflective of

[1]The source code of the NVMe driver and ext4 extension implemented in the Linux kernel can be accessed at https://github.com/yuhun-Jun/kernel_5_15_DA.

[2]The SSD emulator enabled with our approach can be found at https://github.com/yuhun-Jun/nvmevirt_DA.

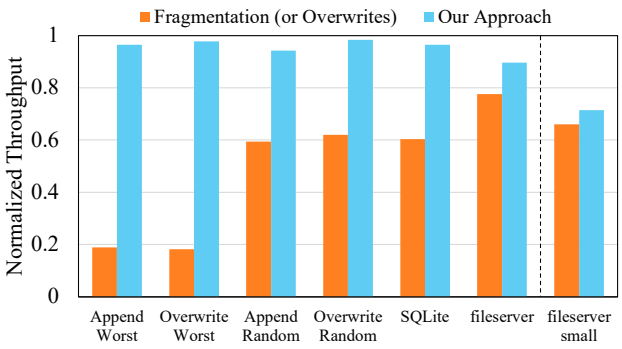


Figure 12: Normalized read throughput of applications executed with the implementation of our approach relative to that with ideal file block and flash page placement.

real-world operations where intervening dummy writes between target file writes are of random sizes. Consequently, instead of confining file blocks' pages to a single die as in earlier experiments, they were distributed randomly across all dies after finishing the append or overwrite operations.

In the worst-case experiments, where all file blocks were allocated to a single die due to fragmentation or overwriting, the results shown in Fig. 12 aligned with those seen in Fig. 11. We observed a significant drop in read performance, which stood at only 19% of the normal throughput for fragmented files and 18.2% for overwritten ones. Our proposed approach successfully preserved the read performance after the append and overwrite operations and only resulted in a 3.5% performance dip for *Append Worst* and 2.3% for *Overwrite Worst*.

In the random disturbance experiments, where the size of dummy file writes between target file writes varied randomly at the 32 KB granularity, ranging from 32 KB to 32 MB, the performance decreased to 59.4% of the ideal for the fragmenting append experiment and 62% for the overwrite experiment. Our approach again successfully suppressed the read performance degradation to 5.8% for *Append Random* and 1.6% for *Overwrite Random*.

In addition to the hypothetical workloads, we analyzed the effectiveness of the proposed approach with SQLite [29] and Filebench's fileserver workload [38].

We established a table and inserted 10,000 records, each 16 KB in size, with SQLite. Simultaneously, we appended 100 KB chunks repeatedly to a dummy file. Following this, we executed a `select` query to retrieve all 10,000 records from the resulting database file, which had a DoF of 5,005. As depicted in the *SQLite* column of Fig. 12, the select query's performance was only 60% of the case where no disturbing writes were performed. In contrast, under our approach, the database file blocks were stored on consecutive dies as intended even with the existence of the dummy writes. As a result, we observed a performance increase of 1.6 times, which represents only a 3.5% drop compared to the case without fragmentation.

The fileserver workload mimics the I/O patterns of a file server. For this, it employs multiple threads executing file creation, random-sized append writes of up to 16 KB, sequential reads on random files, and random file deletion on a file set consisting of 10,000 files averaging 128 KB each. To induce more severe file fragmentation, we modified the workload so that it preallocates a file set of 10,000 128 KB files, each of 10 threads performs 32 KB size append writes on random files from the file set for a duration of 1 minute and measured the read performance. We also removed the file creation and deletion from the workload. At the end of the experiment, the average file size was around 600 KB, the average DoF was 15.7, and the total file set size was 11 GB. The results showed a read performance at 80% of the level seen when files were stored in contiguous file blocks. This lesser performance degradation compared to the previous experiments was due to multiple threads reading simultaneously, increasing the number of outstanding commands. This ensured that most dies continually received operations, enhancing die-level parallelism. Our approach was able to recover the sequential read performance on fragmented files to 93% of the ideal file placement condition.

The *fileserver small* shown in Fig. 12 is from an experiment with settings identical to *fileserver*, but where the append write size was set to 16 KB, smaller than the die granularity. In this experiment, fragmentation further reduced read performance. When writing 32 KB chunks, a single flash page, of which size is 32 KB, can accommodate one write request. However, when writing 16 KB chunks, two chunks are combined and written to a single flash page. As a result, writes from two different files could be recorded on the same page, meaning files of the same size ended up being stored across more pages. This leads to a higher number of flash page reads when reading the file. We confirmed that this phenomenon also occurs when a file uses `fallocate` to pre-allocate consecutive file system blocks and then fills in data in small increments to make a contiguous file, especially if small writes for dummy files intervene. This serves as further evidence that the fragmentation-induced performance degradation is not directly due to fragmentation but rather an issue of data placement within the SSD.

This experiment underscores the limitations of the proposed approach. While it is designed to achieve consecutive die allocation of file blocks during file fragmentation, it's not equipped to counteract the effects of small intervening writes, leading to a file write potentially spanning multiple pages. Consequently, its performance enhancement stood at 8.2%. Addressing the flash page-level fragmentation issue, which may also occur to contiguous files at file system level when the write size is smaller than the flash page size, requires a novel page allocation strategy to counteract that. Such a study would go beyond the scope of this paper and points to an interesting topic for future research.

6 Conclusion

Contrary to early beliefs that file fragmentation does not impact SSD performance, it has now been recognized that SSDs can indeed suffer significant declines in read performance due to file fragmentation. In this paper, we have shown that the root cause of this performance degradation is not delays in the kernel I/O path caused by request splitting, as previously described in the literature. Instead, it arises from misalignments in the SSD's page-to-die mapping, which increase die-level collisions. Furthermore, we demonstrated that such misalignments can occur not only during file fragmentation but also when files are overwritten.

To address this issue, we proposed an NVMe command extension that enables the file system to provide hints about the write operation to SSDs, as well as a novel page-to-die mapping scheme considering the hints for the SSD controller. This ensures that pages are allocated to contiguous dies based on their order in the file. The resulting well-ordered page-to-die mappings effectively prevent additional die-level collisions caused by both file fragmentation and overwrites. Our evaluation showed that, without resorting to costly defragmentation or file rewriting, the proposed approach effectively suppresses the read performance degradation for fragmented or overwritten files to a mere few percent.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Peter Desnoyers, for their valuable suggestions for this paper.

This research was supported by Samsung Electronics, and by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (2021R1A2C200497612).

References

[1] Amir Ban. Flash file system, April 4 1995. US Patent 5,404,485.

[2] Mingming Cao, Theodore Y Tso, Badari Pulavarty, Suparna Bhattacharya, Andreas Dilger, and Alex Tomas. State of the art: Where we are with the Ext3 filesystem. In *Proceedings of the Ottawa Linux Symposium (OLS 05)*, pages 69–96. Citeseer, 2005.

[3] Feng Chen, Rubao Lee, and Xiaodong Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture (HPCA 11)*, pages 266–277. IEEE, 2011.

[4] Alex Conway, Ainesh Bakshi, Yizheng Jiao, William Jannen, Yang Zhan, Jun Yuan, Michael A Bender, Rob Johnson, Bradley C Kuszmaul, Donald E Porter, et al. File systems fated for senescence? nonsense, says science! In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 45–58, 2017.

[5] Alex Conway, Eric Knorr, Yizheng Jiao, Michael A Bender, William Jannen, Rob Johnson, Donald Porter, and Martin Farach-Colton. Filesystem aging:It's more usage than fullness. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[6] Intel Corporation. Understanding the flash translation layer (FTL) specification, 1998.

[7] Giel de Nijs, Ard Biesheuvel, Ad Denissen, and Niek Lambert. The effects of filesystem fragmentation. In *Proceedings of the Linux Symposium*, volume 1. Citeseer, 2006.

[8] BTRFS documentation. btrfs-filesystem(8). https://btrfs.readthedocs.io/en/latest/btrfs-filesystem.html.

[9] NVM Express. NVMe Base Specification Revision 1.4c. 2021.

[10] Windows 8 Help Forums. Optimize drives - defrag HDD and TRIM SSD in Windows 8. https://www.eightforums.com/threads/optimize-drives-defrag-hdd-and-trim-ssd-in-windows-8.8615/.

[11] Congming Gao, Liang Shi, Mengying Zhao, Chun Jason Xue, Kaijie Wu, and Edwin H.-M. Sha. Exploiting parallelism in I/O scheduling for access conflict minimization in flash-based solid state drives. In *Proceedings of the Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11. IEEE, 2014.

[12] Samsung Semiconductor Global. SSD performance FAQs | support. https://semiconductor.samsung.com/consumer-storage/support/faqs/03/.

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

[14] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Shuping Zhang. Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of the international conference on Supercomputing (ICS 11)*, pages 96–107, 2011.

[15] Woopyo Jeong, Jae-woo Im, Doo-Hyun Kim, Sang-Wan Nam, Dong-Kyo Shim, Myung-Hoon Choi, Hyun-Jun Yoon, Dae-Han Kim, You-Se Kim, Hyun-Wook Park, et al. A 128 Gb 3b/cell V-NAND flash memory with 1 Gb/s I/O rate. *IEEE Journal of Solid-State Circuits*, 51(1):204–212, 2015.

[16] Cheng Ji, Li-Pin Chang, Sangwook Shane Hahn, Sungjin Lee, Riwei Pan, Liang Shi, Jihong Kim, and Chun Jason Xue. File fragmentation in mobile devices: Measurement, evaluation, and treatment. *IEEE Transactions on Mobile Computing*, 18(9):2062–2076, 2018.

[17] Cheng Ji, Li-Pin Chang, Liang Shi, Chao Wu, Qiao Li, and Chun Jason Xue. An empirical study of file-system fragmentation in mobile storage systems. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[18] Yuhun Jun, Jaehyung Park, Jeong-Uk Kang, and Euiseong Seo. Analysis and mitigation of patterned read collisions in flash SSDs. *IEEE Access*, 10:96997–97009, 2022.

[19] Dawoon Jung, Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. Superblock FTL: A superblock-based flash translation layer with a hybrid address translation scheme. *ACM Transactions on Embedded Computing Systems*, 9(4):1–41, 2010.

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

[21] Jeong-Uk Kang, Jin-Soo Kim, Chanik Park, Hyoungjun Park, and Joonwon Lee. A multi-channel architecture for high-performance NAND flash-based storage system. *Journal of Systems Architecture*, 53(9):644–658, 2007.

[22] Sang-Hoon Kim, Jaehoon Shim, Euidong Lee, Seongyeop Jeong, Ilkueon Kang, and Jin-Soo Kim. NVMeVirt: A versatile software-defined virtual NVMe device. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST 23)*, Santa Clara, CA, February 2023.

[23] Aneesh Kumar KV, Mingming Cao, Jose R Santos, and Andreas Dilger. Ext4 block and inode allocator improvements. In *Proceedings of the Linux Symposium*, volume 1, 2008.

[24] Linux man page. blktrace(8). https://linux.die.net/man/8/blktrace.

[25] Linux man page. e4defrag(8). https://man7.org/linux/man-pages/man8/e4defrag.8.html.

[26] Linux man page. filefrag(8). https://man7.org/linux/man-pages/man8/filefrag.8.html.

[27] Linux man page. xfs_fsr(8): filesystem reorganizer for XFS. https://man7.org/linux/man-pages/man8/xfs_fsr.8.html.

[28] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new Ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33. Citeseer, 2007.

[29] SQLite Home Page. Sqlite. https://sqlite.org/index.html.

[30] Debian Man pages. defrag.f2fs(8) — f2fs-tools — debian testing. https://manpages.debian.org/testing/f2fs-tools/defrag.f2fs.8.en.html.

[31] Jonggyu Park and Young Ik Eom. Fragpicker: A new defragmentation tool for modern storage devices. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP 21)*, pages 280–294, 2021.

[32] Jonggyu Park and Young Ik Eom. File fragmentation from the perspective of I/O control. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage 22)*, pages 126–132, 2022.

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

[34] SATA-IO. Serial ATA Revision 3.0, 2009.

[35] Takashi Sato. Ext4 online defragmentation. In *Proceedings of the Linux Symposium*, volume 2, pages 179–86. Citeseer, 2007.

[36] Chang Siau, Kwang-Ho Kim, Seungpil Lee, Katsuaki Isobe, Noboru Shibata, Kapil Verma, Takuya Ariki, Jason Li, Jong Yuh, Anirudh Amarnath, Qui Nguyen, Ohwon Kwon, Stanley Jeong, Heguang Li, Hua-Ling Hsu, Tai-yuan Tseng, Steve Choi, Siddhesh Darne, Pradeep Anantula, Alex Yap, Hardwell Chibvongodze, Hitoshi Miwa, Minoru Yamashita, Mitsuyuki Watanabe, Koichiro Hayashi, Yosuke Kato, Toru Miwa, Jang Yong Kang, Masatoshi Okumura, Naoki Ookuma, Muralikrishna Balaga, Venky Ramachandra, Aki Matsuda, Swaroop Kulkani, Raghavendra Rachineni, Pai K. Manjunath, Masahito Takehara, Anil Pai, Srinivas Rajendra, Toshiki Hisada, Ryo Fukuda, Naoya Tokiwa, Kazuaki Kawaguchi, Masashi Yamaoka, Hiromitsu

Komai, Takatoshi Minamoto, Masaki Unno, Susumu Ozawa, Hiroshi Nakamura, Tomoo Hishida, Yasuyuki Kajitani, and Lei Lin. A 512Gb 3-bit/cell 3D flash memory on 128-wordline-layer with 132MB/s write performance featuring circuit-under-array technology. In *Proceedings of the International Solid-State Circuits Conference (ISSCC 19)*, pages 218–220. IEEE, 2019.

[37] Keith A Smith and Margo I Seltzer. File system aging—increasing the relevance of file system benchmarks. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems (SIGMETRICS 97)*, pages 203–213, 1997.

[38] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41(1):6–12, 2016.

[39] Micron Technology. Should you defrag an SSD? https://www.crucial.com/articles/about-ssd/should-you-defrag-an-ssd.

[40] Micron Technology. Tn-29-28: Memory management in NAND flash arrays overview. 2005.

[41] Tenforums. Optimize and defrag drives in Windows 10. https://www.tenforums.com/tutorials/8933-optimize-defrag-drives-windows-10-a.html.

[42] Guangyu Zhu, Jeongeun Lee, and Yongseok Son. An efficient and parallel file defragmentation scheme for flash-based SSDs. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing (SAC 22)*, pages 1208–1211, 2022.

Artifact Appendix

Abstract

The provided artifacts consist of shell scripts designed to replicate the experimental results introduced in the paper. These experiments were aimed at analyzing read performance degradation caused by fragmentation and assessing the efficacy of the proposed approach. Additionally, the artifacts include a customized NVMeVirt implementation featuring the proposed page placement scheme for the FTL. This is complemented by a modified Linux kernel equipped with the necessary file system and NVMe device driver support for the customized NVMeVirt.

Scope

These artifacts include shell scripts that enable the replication of results presented in Section 3, as depicted in Figs. 3, 4, 7, and 8. The shell scripts for demonstrating the effectiveness of the proposed approach, as illustrated in Fig. 11, are also included in the artifacts.

Furthermore, the artifacts comprise a customized NVMeVirt utilizing an FTL that implements the proposed page placement scheme. This is complemented by the modified Linux kernel, which is also a part of the artifacts. It provides NVMeVirt with the page placement hints. The shell scripts for conducting experiments on the workloads used in Section 5.2 are provided as well. These scripts were instrumental in obtaining the experimental results showcased in Fig. 12.

Contents

The shell scripts below run the experiments introduced in Section 3 and Section 5.1. In the file names, the `*` is replaced with the target device name, such as `NVMe_A` or `SATA_B`.

`varyingdof_*.sh`: These shell scripts are for the experimentation analyzing the read time change according to the varying DoF of files stored on NVMe and ramdisk, as shown in Figs. 3 and Fig. 4, respectively.

`interface_*.sh`: These measure the time taken to read 8 MB of data from the target SSD, depending on the unit size of the read operation. This was used to produce the results illustrated in Fig. 7.

`alignment_*.sh`: These measure the throughput of read operations while varying the interval between starting points of consecutive operations, as shown in Fig. 8.

`pseudo_(append|overwrite)_*.sh`: These shell scripts mimic the write patterns for three cases: when files are written contiguously, when written in a fragmented manner, and when written according to the write patterns that occur in our approach. It then measures the read performance for each of these cases. This was used to produce the results introduced in Fig. 11.

The following shell scripts are intended for the experiments explained in Section 5.2.

`hypothetical_(append|overwrite).sh`: These shell scripts measure the read throughput for a file after performing a series of append write or overwrite operations to it. The append and overwrite operations can be configured to follow the worst-case pattern or the random pattern. The results of executing these on NVMeVirt are shown in Fig. 12.

`sqlite.sh`: This was used to obtain the experimental results shown in Fig. 12. It triggers write operations to create a fragmented database file when running SQLite. Subsequently, it performs select operations through SQLite on the fragmented database file and measures the performance.

`fileserver.sh`: This was also used to obtain the experimental results shown in Fig. 12. This shell script measures the performance in circumstances where the files generated by Filebench's fileserver workload become fragmented.

`fileserver_small.sh`: This script is similar to `fileserver.sh`, except that the append operations are performed with a size smaller than the flash memory page size.

The detailed instructions can be found in the `README.md` file located in the GitHub repository.

Hosting

The GitHub repository for the artifacts is https://github.com/yuhun-Jun/fast24_ae. The results introduced in this paper were produced from the commit version `89ba3a9` of the `main` branch.

Requirements

The shell scripts for analyzing fragmentation-induced performance degradation must be configured according to the internal parameters of the target SSD. The provided artifacts are set up for the devices introduced in Table 2. For other SSDs, settings including the write offset must be appropriately adjusted.

For the customized NVMeVirt to function properly, the support from OS Kernel's file system and NVMe driver is mandatory. Therefore, it operates correctly only when executed on the provided Linux Kernel. Furthermore, as NVMeVirt utilizes main memory to emulate storage space, stable experimental outcomes require that the workload operates exclusively within a single NUMA domain. This approach avoids cross-NUMA domain memory accesses, which can significantly vary in execution time and potentially affect the consistency of results.