An Efficient Order-Preserving Recovery for F2FS with ZNS SSD

Euidong Lee
Seoul National University
Seoul, South Korea
led5198@snu.ac.kr

Ikjoon Son
Seoul National University
Seoul, South Korea
ikjoon.son@snu.ac.kr

Jin-Soo Kim
Seoul National University
Seoul, South Korea
jinsoo.kim@snu.ac.kr

ABSTRACT

Storage devices use write buffers to improve performance, where multiple write requests are processed in parallel and completed in a random order. This may result in data loss in the event of a sudden failure. Therefore, Linux filesystems provide the fsync() system call to prevent data loss and ensure write order. However, the fsync() system call in F2FS, one of the most popular filesystems, is inefficient and insufficient for guaranteeing data consistency.

We propose a new technique called Order-Preserving Recovery with Write pointer (OPRW) to ensure data consistency during the filesystem recovery. OPRW utilizes the write pointer in Zoned NameSpace (ZNS) SSDs to efficiently determine the persistence of data without the need for I/O operations. This approach allows OPRW to provide a higher level of consistency and performance improvement for fsync(), while minimizing the impact on recovery time. As a result, our solution improves performance by up to 1.2x on realistic workloads.

CCS CONCEPTS

• **Information systems → Storage management**.

KEYWORDS

Filesystems, F2FS, ZNS SSD

ACM Reference Format:
Euidong Lee, Ikjoon Son, and Jin-Soo Kim. 2023. An Efficient Order-Preserving Recovery for F2FS with ZNS SSD. https://doi.org/10.1145/3599691.3603416

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HotStorage '23, July 9, 2023, Boston, MA, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0224-2/23/07…$15.00
https://doi.org/10.1145/3599691.3603416

1 INTRODUCTION

Modern storage devices use a buffering technique for write commands that is essential for achieving high performance. After the host makes a write request to the storage device, it does not check to see if the writing is actually complete on the non-volatile media; instead, the device completes the write in a random order at a random point in time. In doing so, the device processes multiple write requests in parallel and handles them in a more efficient order to improve system performance. However, this policy may lead to data loss in the event of a sudden failure, so hosts need an additional mechanism to protect critical data.

Linux filesystems provide the fsync() system call to preserve critical data and guarantee write order. When fsync() is called for a specific file, the filesystem blocks the return until the data and metadata for that file are written to the physical media. Once this process is complete, the user can expect that the file has been physically written to the device and is safe from sudden power loss. Because the fsync() system call is called so frequently, filesystems must both provide data persistence and minimize response time.

We found that the persistence support of the fsync() system call in F2FS, one of the most popular filesystems, is insufficient and/or inefficient. Specifically, F2FS completes fsync() after writing the changed data block and then its associated node block, without writing any additional metadata. However, just writing two blocks one after the other cannot prevent reordering. Therefore, in F2FS, there is a risk of data corruption if a crash occurs during fsync().

The F2FS community has raised this issue and tried to solve it by offering the `strict` mode that uses atomic writes. However, in the `strict` mode, performance degradation is inevitable as it needs to check whether the block is written to the physical media frequently to preserve the write order.

We have designed a new solution to this problem in one of the next-generation storage devices, Zoned NameSpace (ZNS) SSDs [9]. Compared to the traditional SSDs, ZNS SSDs divide the whole disk space into fixed-size zones and only sequential writes are permitted within a zone. The device maintains a special *write pointer* to track the next position to be written for each zone. We find that this write pointer can actually provide useful information to the host in situations where recovery from a sudden power down is required. As a result, we show that the performance can be improved up to 1.2x when fsync() occurs periodically, while providing better data persistence.

The rest of the paper is organized as follows. In Section 2, we describe the background of our work. In Section 3 and 4, we present the problems in the current F2FS filesystem and the propose solution, respectively. Section 5 shows evaluation results and Section 6 concludes the paper.

2 BACKGROUND

2.1 fsync() system call

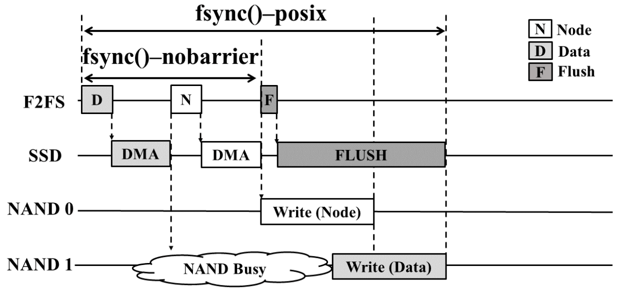
In a POSIX-compliant filesystem, fsync() is used to ensure two things: the order of write commands and the persistence of data. Performing fsync() is an expensive operation because it reveals the media's write response time during processing, which is, hence, known to be a performance bottleneck in many workloads [3, 14, 18].

Some devices, such as enterprise SSDs with built-in backup battery or mobile devices with a non-removable battery, provide hardware protection against sudden power loss. This feature, called Power-Loss Protection (PLP), ensures persistence for all data once the DMA transfer is complete. With these devices, some filesystems are trying to increase the performance of fsync() operations by providing a `nobarrier` mode. In this mode, the filesystem does not wait for media write times but users can still guarantee the persistence of data through fsync().

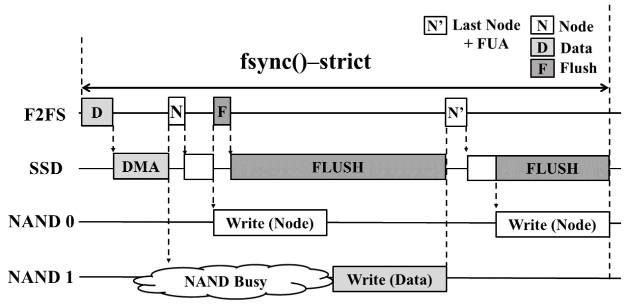
2.2 F2FS Filesystem

F2FS [13] is a Linux filesystem for flash storage. It is based on the log-structured filesystem (LFS), but manages filesystem metadata in a random write fashion to achieve high performance without cascaded metadata updates. F2FS is being widely used in Android-based mobile systems.

F2FS performs periodic checkpointing to ensure the filesystem consistency. Checkpoints preserve the logical structure of each file along with the states of nodes and segments at that point in time. In the event of a power failure, F2FS simplifies the recovery process by rolling back to the most recent checkpoint. For performance reasons, F2FS does not trigger a checkpoint for each fsync() call. Instead, F2FS just writes data blocks and their associated direct node blocks only on fsync(). Nodes that are written during fsync() are marked with a special flag, and after rollback, there is an additional process of sequentially recovering nodes that are flagged but still unrecovered, called *roll-forward*. During this process, the data blocks referenced by the target node are marked as valid, and the old blocks are invalidated. These changes are updated in the Segment Information Table (SIT).



(a) POSIX mode and `nobarrier` mode



(b) `strict` mode

Figure 1: fsync() in F2FS

Figure 1a illustrates the handling of fsync() in F2FS. When performing fsync(), a node block (denoted as `N`) is written after the DMA transfer of the associated data block (denoted as `D`) is finished. After all DMA transfers are completed, the normal POSIX mode uses the `flush` command to ensure durability [6], whereas the `nobarrier` mode returns immediately.

2.3 Zoned NameSpace SSDs

ZNS SSDs organize multiple blocks into a zone and rely on the host to determine the write order for each zone and when to erase zones. This allows the device to provide predictable response times by eliminating its own data placement and garbage collection operations. On the other hand, the host system requires a new software stack for the new zone interface.

Each zone of a ZNS SSD can only be written sequentially. As a result, it is necessary to maintain the last written location for each zone, and to do so, the device manages a *write pointer* and shares it with the host [2, 9]. In the event of a sudden power loss, ZNS SSDs perform their internal recovery mechanism to locate the last written position. Therefore, when the device is powered on, the host can know that LBAs smaller than the write pointer are already written persistently, while larger LBAs are unwritten. When the write pointer reaches the end of a zone, the zone is closed and no further write operations are allowed until being reset.

Since F2FS is based on LFS, it was easy to adapt to ZNS, and a version of F2FS that works on ZNS SSD is publicly available. In order to comply with the ZNS specification, the DMA order of writes to the same zone must be ensured, which is currently handled by I/O schedulers such as MQ-Deadline [8].

3 PROBLEM & MOTIVATION

3.1 Data Corruption in F2FS

F2FS has a potential risk of user data corruption when a crash occurs during fsync() [19]. Figure 1a shows the situation where a user data `D` and the associated node `N` should be written in that order. However, even if the DMAs are processed in order, there is a possibility that the two requests are written out of order inside the SSD; figure 1a shows the case that the NAND write operation for node is performed before the data because, for example, the NAND die 1 on which the data will be written is busy. If a crash occurs when the node block becomes durable before the data block, the node will point to invalid data, resulting in the loss of existing data. This is a violation of the data consistency requirement [4, 5].

We have confirmed through experiments that actual data corruption occurs in F2FS. According to our experiments, F2FS could not recover data in 3% of the tested cases (cf. Section 5.2 for details). Storage devices with PLP do not suffer from this consistency issue as the completion of DMA transfers guarantees durability. However, we show that the fsync() performance can be improved further even when the storage is equipped with PLP.

The original F2FS attempted to address this issue by providing the `strict` mode [19]. Figure 1b illustrates the process of the `strict`-mode fsync(). In the `strict` mode, F2FS inserts a `flush` command before the last node block, which ensures that the last node is not written until all the preceding blocks have been persisted. If the last node is not persisted, the remaining node blocks are not recovered, preventing the reversal of the persistence order between data and node blocks. However, this approach requires additional `flush` operations and sacrifices performance to ensure data consistency.

3.2 Persistence Check by Write Pointer

The aforementioned difficulties with fsync() come from the limited information provided by the block interface. For the storage devices based on the block interface, all blocks can be overwritten at any point in time and in any order, so there is no way to find out the temporal ordering among write operations. It is also assumed that there is always data in every block, and there is no way to determine which blocks contain valid data. Since this information is essential to implement a consistent filesystem, traditional filesystems use additional metadata such as allocation bitmaps combined with journaling or checkpointing. However, these techniques incur internal I/O and introduce space overhead.

In ZNS SSDs, in contrast, the write pointer can be used to determine durability. Since the write pointer is always increased sequentially, it is possible to determine the persistence of a series of blocks using a simple comparison operation without any additional I/Os. By utilizing this property of the write pointer, we aim to determine the ordering among write requests and resolve data consistency issues.

4 DESIGN

4.1 Recovery for Data Consistency

In order to avoid data corruption mentioned in Section 3.1, a naive solution is to enforce the written order between data blocks and node blocks during the fsync() process. With this eager and pessimistic approach, the node will be written after the data become durable, ensuring that the new data pointed to by the node is always valid. However, this approach is impractical because it will significantly increase the time to perform fsync(). Another solution to address this problem is to recover data consistently during the file system recovery process when an actual crash occurs. This lazy and optimistic approach can simplify the fsync() process, but it leads to a long recovery time due to additional I/O operations to determine the validity of data blocks.

We propose a new *Order-Preserving Recovery by Write pointer (OPRW)* technique to accelerate the fsync() performance while maintaining data consistency on ZNS SSDs. OPRW is based on the aforementioned optimistic approach and addresses the issue of long recovery time by leveraging the write pointer of ZNS SSDs. Figure 2 shows the overall process of OPRW in a situation where the filesystem is corrupted due to a power failure during the fsync() operation. The problem is the nodes recorded after the last completed checkpoint. As mentioned, the updated data blocks in these nodes may not be valid. In the roll-forward process, each time a node is recovered sequentially, it is first checked for data consistency.

The key idea is to utilize the write pointer to quickly identify the persistence of data and detect any reversal of the written order, without the need for I/O operations. Specifically, we check whether the data blocks pointed to by the node are located before the write pointer. Even when one of data blocks is located after the write pointer, it means that the consistency of the node's data has been compromised and the node should be discarded. In this case, the data consistency can be preserved because the existing nodes and data will remain intact. Also, since node blocks are written

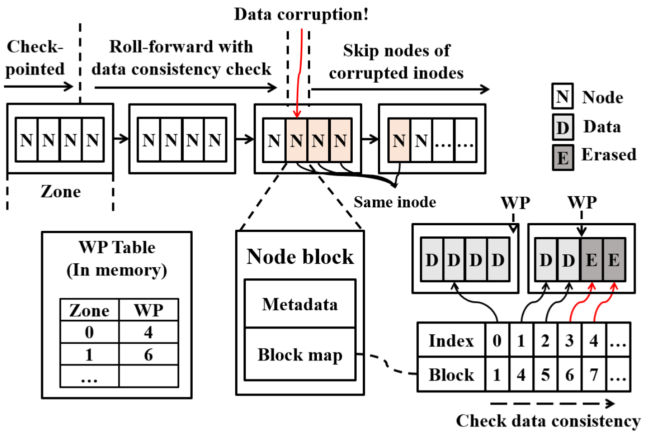


Figure 2: Order-Preserving Recovery by Write Pointer

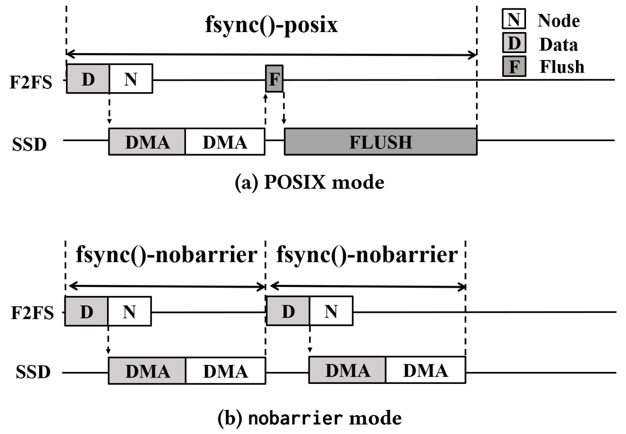


Figure 3: fsync() in OPRW

sequentially, the subsequent node blocks belonging to the broken files should be also discarded to guarantee the data consistency semantics.

The specific process for checking data consistency during roll-forward is described in Algorithm 1. First, each entry in the node block is examined. If the address pointed to by the entry is smaller than the write pointer of the zone that contains it, it is considered to be a valid pointer. If it is equal to or greater than the write pointer, it is considered to be pointing to invalid data. If there exists an entry pointing to invalid data block, the data consistency of the corresponding node is corrupted and it is excluded from the recovery target. Also, the inode of the corrupted node is added to the *skip_list*, so that subsequent nodes belonging to the same inode can be excluded from the recovery target. During this process, the entire write pointer table can be loaded into memory through a single zone management receive command [9], which allows the recovery process to be repeated without additional I/O.

Algorithm 1 Data Consistency Check

```
1:  $skip\_list \leftarrow \emptyset$
2:  for each $node \in roll\ forward\ target$ do
3:      $inode \leftarrow find\_inode(node)$
4:      if $inode \in skip\_list$ then
5:          $discard(node)$
6:          $continue$
7:      end if
8:      for $i = 0, 1, \ldots, N_{ptrs} - 1$ do
9:          $wp \leftarrow WP\_Table[zone(pointer[i])]$
10:         if $pointer[i] < wp$ then                ▷ Valid
11:             continue
12:         else                                     ▷ Invalid
13:             $skip\_list \leftarrow skip\_list \cup \{inode\}$
14:             $discard(node)$
15:             break
16:         end if
17:     end for
18: end for
```

4.2 Performance of fsync() modes

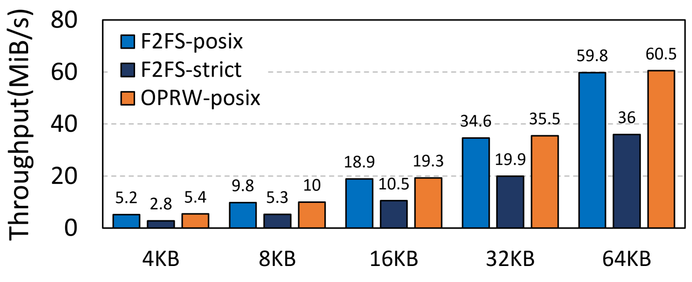
POSIX mode with non-PLP devices. In the POSIX mode, F2FS performs two DMA operations and one `flush` operation sequentially for the purpose of data consistency and persistence, as shown in Figure 1a. Although performing DMA transfers in order cannot guarantee the written order in the storage, now we are able to find out whether the written order has been reversed using the OPRW technique. Therefore, the write order of the data and node blocks need not be guaranteed. In other words, as shown in Figure 3a, it is possible to immediately send a write request for the node block without waiting for the completion of the DMA transfer of the data. As a result, the proposed technique can not only ensure consistency but also improve performance.

`nobarrier` mode with PLP devices. For PLP devices, OPRW can eliminate the unnecessary waiting for the DMA transfer of data blocks during the fsync() operation, similar to the POSIX mode. Without `flush`, the DMA transfer time in the `nobarrier` mode takes up a large portion of the fsync() execution time. As a result, the operations of host and storage can be overlapped and performed simultaneously, as shown in Figure 3b. This allows for increased I/Os to the storage and, in turn, more efficient utilization of the storage's internal parallelism.

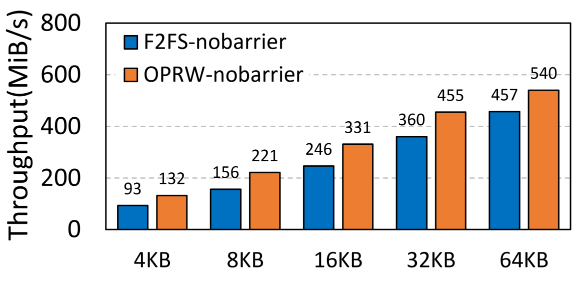
5 EVALUATION

5.1 Evaluation Setup

The evaluation was performed on a server with two Intel Xeon Silver 4116 2.10 GHz processors and 32 GiB of memory, with a kernel version of 5.14.4. For storage, we use



(a) Non-PLP device (NVMeVirt)



(b) PLP device (ZN540)

Figure 4: FIO Random Write Followed by fsync()

	Total	Failed
F2FS-posix	1000	30
F2FS-strict	1000	0
OPRW-posix	1000	0

Table 1: Data Consistency Test

a ZN540 [7] 4TB ZNS SSD from Western Digital. ZN540 supports PLP to ensure data persistence once it has been transferred. In addition, we use the NVMeVirt [11] emulator to evaluate non-PLP ZNS SSDs. The timing parameters of NVMeVirt were adjusted so that its performance is equivalent to ZN540, but `flush` latency was added based on the write time of NAND flash to emulate a ZN540 without PLP.

We have noticed that F2FS in the latest Linux kernel 6.3.3, does not insert an additional pre-`flush` before the node block even in the `strict` mode when running on ZNS devices. It is known that this change is made to avoid violating ZNS specifications due to command reordering [10]. Without performing pre-`flush`, however, the data consistency issue still remains. In order to measure the correct performance, we added a pre-`flush` after the DMA transfer of the data for the `strict` mode.

5.2 Data Consistency Test

First, we perform data consistency tests on NVMeVirt which emulates a ZN540 device without PLP. The sequence for verifying data consistency is as follows; a file is filled with a specific pattern, then a write request is made to the file with the same pattern, and a crash occurs at a random point in time. When the filesystem recovers, it reads the file and checks if the pattern matches.

Table 1 shows the results of the data consistency tests. In the POSIX mode, F2FS fails in 3% of the tested cases which means that performing in-order DMAs is not enough in the traditional fsync() implementation. On the other hand, the proposed OPRW scheme and F2FS with the `strict` mode recover all the data correctly after the roll-forward process.

5.3 Micro Benchmark

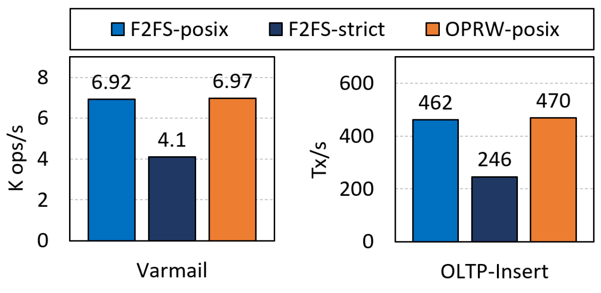
Figure 4 shows the performance improvement of the OPRW technique using FIO [1]. We measure the throughput of random writes followed by fsync() with one thread while varying the payload size.

The performance of fsync() is extremely slow on the PLP-disabled NVMeVirt. This is because both F2FS and OPRW need to perform the costly `flush` command at the end of fsync() to ensure data persistence. The existing solution, `strict` mode, performs 44% worse than the POSIX mode, showing that the cost of the additional `flush` command to preserve data consistency is very high. OPRW does not show a significant performance improvement over the original F2FS. This is because the DMA operations and software overhead are a very small fraction of the fsync() execution time compared to `flush` operations. However, OPRW has a benefit of ensuring full data consistency in the event of a power failure, without compromising performance.

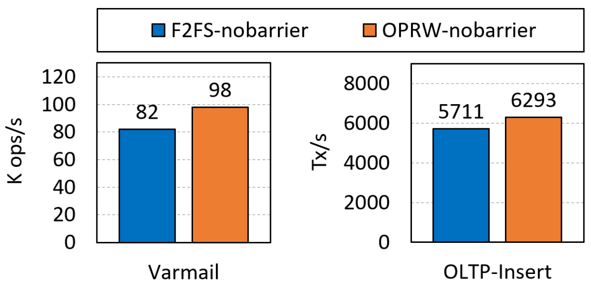
The performance difference is larger on PLP-enabled SSDs with the `nobarrier` mode, because they do not suffer from `flush` operations. As a result, OPRW outperforms the original F2FS by 18% ~ 42%. This indicates that the delay in writing node blocks is causing a performance bottleneck. There is a software overhead for checking DMA responses and sending write requests for blocks, and hiding this overhead while maintaining data consistency leads to a performance improvement.

5.4 Macro Benchmark

We evaluate the proposed scheme using varmail [16] and OLTP-Insert [12] workloads, with a single thread. Figure 5a and 5b show that the performance improvement of OPRW is also significant in realistic workloads. For varmail and OLTP-Insert, OPRW outperforms F2FS by 1.2x and 1.1x respectively, when running on ZN540 with PLP. The performance gains are particularly high for varmail, which relies more heavily on fsync() than OLTP-Insert [17].



(a) Non-PLP device (NVMeVirt)



(b) PLP device (ZN540)

Figure 5: Macro Benchmark

# of scanned nodes	40791	50962	57500
F2FS (ms)	4177	7307	9751
OPRW (ms)	4380	7584	10093
Difference (ms)	203	277	342

Table 2: Recovery Time

In figure 6, we evaluate the performance scalability by varying the number of threads in the varmail workload. With a small number of threads, OPRW outperforms the original F2FS significantly. However, the performance improvement is getting smaller as the number of threads increases. This is due to the inherent scalability problem of F2FS where multiple threads contend for a lock while they are writing data to a single log sequentially [15]. As the scalability of F2FS improves, we expect that OPRW will be more effective in the multi-threaded environment.

5.5 Recovery Time

Table 2 shows the filesystem recovery time depending on the amount of nodes recovered during roll-forward in NVMeVirt. We measured the duration of the roll-forward after a crash occurred during the execution of the varmail workload. As the number of scanned nodes increases, the time for roll-forward also increases. If the proposed data consistency checks are included in the roll-forward, it increases the recovery time by about 203ms ~ 342ms, but this is relatively insignificant compared to a full filesystem recovery time of several seconds or more. This is possible because OPRW only performs computations and memory accesses without performing any I/O operation. In addition, under the normal circumstances, the number of nodes to be scanned during the system recovery is limited due to periodic checkpointing. Therefore, data consistency checks using write pointers can improve the filesystem consistency with negligible impact on the recovery time.

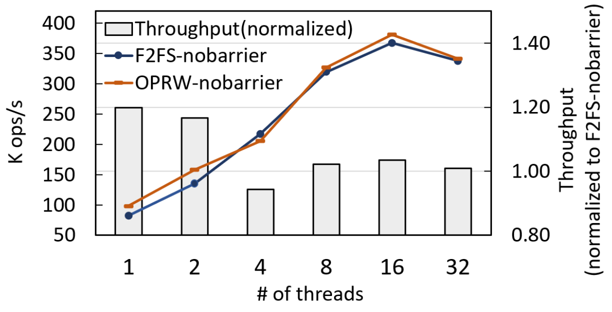


Figure 6: Scalability Test (varmail)

6 CONCLUSION

In this paper, we point out that the fsync() operation of F2FS is not sufficient to maintain data consistency by reproducing the issue through experiments. To solve this problem, we leverage the write pointers provided by the ZNS SSD interface, and propose a method to recover the filesystem's consistency with minimal overhead while ensuring data consistency. We also remove the synchronization actions previously used to insufficiently guarantee consistency. As a result, we show up to 1.2x improvement in the write performance on realistic workloads.

ACKNOWLEDGMENT

We would like to thank our shepherd, Youjip Won, and anonymous reviewers for their invaluable feedback. This work was supported by the National Research Foundation of Korea (NRF) grant (No. 2019R1A2C2089773), and Institute of Information & communications Technology Planning & Evaluation (IITP) grant (No. IITP-2021-0-01363) funded by the Korea government (MSIT). This work was also supported by SNU-SK Hynix Solution Research Center (S3RC).

REFERENCES

[1] Jens Axboe. 2014. fio: Flexible I/O Tester. https://github.com/axboe/fio.

[2] M. Bjørling, A. Aghayev, H. Holmberg, A. Ramesh, D. L. Moal, G. R. Ganger, and G. Amvrosiadis. 2021. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*. Virtual.

[3] Yun-Sheng Chang and Ren-Shuo Liu. 2019. OPTR: Order-Preserving Translation and Recovery Design for SSDs with a Standard Block Device Interface. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*. Renton, WA.

[4] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. 2013. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Farmington, PA.

[5] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R.H.Arpaci-Dusseau. 2012. Consistency Without Ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*. San Jose, California, USA.

[6] J Corbet. 2010. The end of block barriers. https://lwn.net/Articles/400541/.

[7] Western Digital. 2021. DC ZN540 Data sheet. https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/product/ultrastar-dc-zn540-ssd/data-sheet-ultrastar-dc-zn540.pdf.

[8] Western Digital. 2021. Write Ordering Control. https://zonedstorage.io/docs/linux/sched.

[9] NVM Express. 2022. Zoned Namespace Command Set Specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-Zoned-Namespace-Command-Set-Specification-1.1c-2022.10.03-Ratified.pdf.

[10] Jaegeuk Kim. 2022. f2fs: use flush command instead of FUA for zoned device. https://lore.kernel.org/lkml/20220419215703.1271395-1-jaegeuk@kernel.org/.

[11] Sang-Hoon Kim, Jaehoon Shim, Euidong Lee, Seongyeop Jeong, Ilkueon Kang, and Jin-Soo Kim. 2023. NVMeVirt: A Versatile Software-defined Virtual NVMe Device. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST)*. Santa Clara, CA.

[12] Alexey Kopytov. 2004. SysBench manual. https://imysql.com/wp-content/uploads/2014/10/sysbench-manual.pdf.

[13] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *Proceedings of the 13rd USENIX Conference on File and Storage Technologies (FAST)*. Santa Clara, California, USA.

[14] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. 2020. Write Dependency Disentanglement with HORAE. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Virtual.

[15] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. 2021. Max: A Multicore-Accelerated File System for Flash Storage. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*. Virtual.

[16] Richard McDougall and Jim Mauro. 2005. Filebench: A Model Based File System Workload Generator. http://www.nfsv4bat.org/Documents/nasconf/2004/filebench.

[17] Joontaek Oh, Seung Won Yoo, Hojin Nam, Changwoo Min, and Youjip Won. 2023. CJFS: Concurrent Journaling for Better Scalability. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST)*. Santa Clara, CA.

[18] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. 2018. Barrier-Enabled IO Stack for Flash Storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*. Oakland, California, USA.

[19] Chao Yu. 2021. f2fs: fix to force keeping write barrier for strict fsync mode. https://lkml.org/lkml/2021/6/1/350.