Analysis for the Performance Degradation of `fsync()` in F2FS

Gyeongyeol Choi
Hanyang University
Seoul, Korea
chl4651@hanyang.ac.kr

Youjip Won
Hanyang University
Seoul, Korea
yjwon@hanyang.ac.kr

ABSTRACT

It is necessary to guarantee the order of write requests being written into the storage device for the filesystem consistency. `fsync()` guarantees the consistency of the filesystem by the flush-based synchronization technique. There have been the various attempts to improves the performance of `fsync()`, and some of them contribute to improve the performance of `fsync()`. Among these attempts, two approaches (F2FS and Supercap-SSD) contribute to decrease of the `fsync()` overhead. However, we discovered that the huge bottleneck occurs in the process of `fsync()` when using these two techniques. We identified the cause of this problem is the scanning overhead for the free node IDs, and digitalized the effect of the `fsync()` performance degradation. Consequently, we proved that the scanning overhead in `fsync()` of F2FS decreases the I/O performance if the overhead at the FLUSH command diminishes.

CCS Concepts

• **Information systems→Database management system engines**

Keywords

F2FS; Cache Flush; I/O request; NAND flash storage; Supercapacitor; `fsync()` system call; Logging (or Journaling)

1. INTRODUCTION

It is necessary to guarantee the order of write requests being written to the storage device for the consistency. The existing linux kernel uses the FLUSH-based synchronization function: `fsync()` (or `fdatasync()`) to ensure the write order [1,2]. Because `fsync()` is the synchronous operation, it is a very expensive system call among the file operations. In the some workloads (e.g. mail server [3]) that call the `fsync()` frequently, `fsync()` becomes the big bottleneck in the overall performance. To solve the performance problems of `fsync()`, there have been the various attempts, and some of them contribute to improve the performance of `fsync()`.

ICCIC'18, January 5–7, 2018, Manila, Philippines.

DOI: http://dx.doi.org/10.1145/12345.67890

The first approach is to design the new filesystem. F2FS [4] is the log-structured filesystem [5,6] optimized for NAND flash storage that is widely used from mobile devices such as smart phones and tablets to server devices that perform high-speed operations. F2FS shows the higher performance than other journaling filesystems as well as other log-structured filesystems on several workloads [4]. Especially, in the `fsync()`-intensive workloads [7], the performance gap is larger.

The second approach is to reduce the overhead at the FLUSH command. The FLUSH command used in the `fsync`() has a large overhead. There are the several efforts to reduce the overhead of the FLUSH command. You can replace some FLUSH requests in the `fsync()` process with the transactional checksum [8,9,10], set the filesystem to ignore the FLUSH requests (`nobarrier` option [11]), or put the supercapacitor [12,13] inside storage. As follows, the overhead for the FLUSH command can be reduced. Among these attempts, it is an attractive solution to place the supercapacitor inside the storage. This solution can ensure the reliability of the FLUSH command as well as reduce the overhead of the actual FLUSH command. The Storage with the supercapacitor (Supercap-SSD) does not wait until the data in the writeback cache of the storage is written into the disk surface persistently during the processing of the FLUSH command. In the event of a sudden system crash because of the power failure, the supercapacitor installed on the storage solves the problem by powering the storage until the data blocks in the writeback cache are written to the storage.

The two approaches (F2FS, Supercap-SSD) that can reduce the overhead of `fsync()` are effective in improving the performance of `fsync()`, respectively. However, when both techniques are combined, the performance of the real workloads that calls `fsync()` shows the unexpected result. We found that when the overhead at the FLUSH command is removed from F2FS file system, the performance of the F2FS decreases compared to other file systems with no-flush. In this paper, we quantified the performance of the F2FS for the Supercap-SSD and analyzed the cause of the performance degradation problem in `fsync()` when reducing the overhead at the FLUSH command in the F2FS.

We construct the section of the paper as follows. Section 2 explains the important background; FLUSH command and `fsync()` mechanism in F2FS. Section 3 introduces the problem that the performance of `fsync()` in F2FS decreases on Supercap-SSD. Section 4 proves the performance degradation in `fsync()` affects the I/O performance by performing the several experiments. Section 5 summarizes the contents, and concludes the paper.

2. BACKGROUND

2.1 FLUSH Command

Cache Flush [1,7] is the storage command that writes data blocks stored in the writeback cache of the storage into the disk surface permanently. Although the data block is stored in the writeback cache of the storage, sudden power failure makes the data damaged because the writeback cache is volatile. Therefore, the function of the FLUSH command is important to guarantee the consistency of the filesystem.

However, the FLUSH command is a serious bottleneck in processing the I/O operations. This is because the processing of the FLUSH command takes a long time and the I/O stack delays the processing of the subsequent I/O requests until the completion of the FLUSH command.

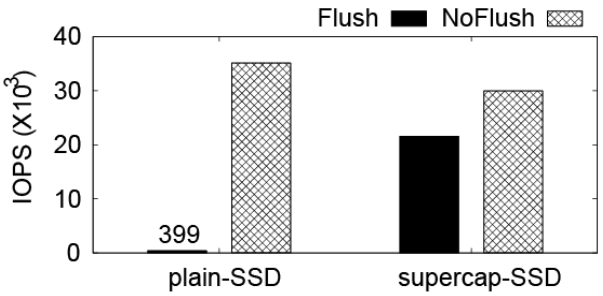

Figure 1. FLUSH Overhead (4 KB Random Write followed by `fdatasync()`)

We conducted a simple experiment with two storage devices to confirm how much the FLUSH command affects the performance of the actual I/O operations. Figure 1 shows the performance of 4 KB random write (overwrite) + `fdatasync()` to the 10 GB file based on EXT4 [14] (Ordered mode). We used Samsung 850PRO 128 GB for plain-SSD, and Samsung SM843 480 GB (with the supercapacitor) for supercap-SSD. We measured the performance of the NoFlush situation by using the `nobarrier` option in the EXT4 filesystem to set the filesystem not to call the FLUSH command.

The result of the experiment shows that both storage devices have the higher I/O performance in the NoFlush situation compared to FLUSH. However, the size of the performance improvement is different. In Plain-SSD, the performance gap in Flush and NoFlush is bigger than 88X while performance gap is 40 % in supercap-SSD. The reason for the performance difference from the FLUSH command in supercap-SSD is that when the FLUSH command is received from the storage, the data in the writeback cache is returned immediately without waiting to be written to the disk surface. Through the following experimental result, we could confirm that the FLUSH command can be a serious bottleneck in processing I/O operations.

2.2 `fsync()` System Call in F2FS

Figure 2 shows the I/O processing when `fsync()` is called on F2FS. Firstly, F2FS dispatches the data segment of the file into the storage, and waits until the transfer of the data is completed. After the transfer of the data, F2FS issues the node segment and waits until the transfer of the node is completed. Finally, F2FS issues the FLUSH commands to guarantee the durability of the data and node segment. Although the application calls `fsync()`, F2FS does not issue the metadata of the filesystem. The metadata of F2FS is only issued when the checkpoint, or the segment cleaning is triggered.

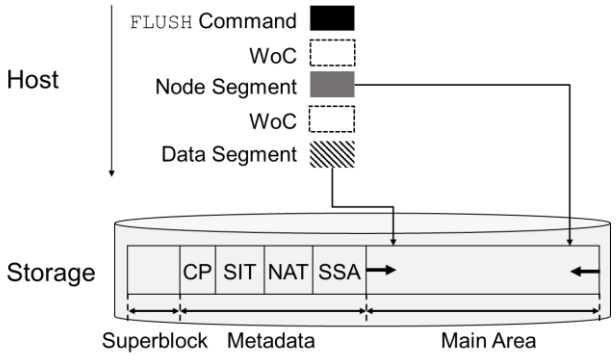

Figure 2. `fsync()` in F2FS (WoC: Wait-on-Complete)

F2FS has a high processing speed when performing `fsync()` compared to other file systems that support recovery. For example, when compared to the journaling file system EXT4, which is currently used in the Linux kernel, `fsync()`-intensive workload (Filebench [15] – Varmail) showed a 2.5X performance improvement [4]. This difference in performance of `fsync()` is affected by the number of FLUSH requests processed by the `fsync()` system call.

The `fsync()` operation of the EXT4 filesystem issues two FLUSH commands. First FLUSH command guarantees the order between file data and metadata. Second FLUSH command is used for the persistency of the transaction. On the other hand, `fsync()` of F2FS issues one FLUSH command. After writing data segment and node segment, the `fsync()` calls the FLUSH command at the end. This is because the logging of F2FS is performed by periodically called checkpoint operation whereas the EXT4 performs the recovery by using the data in the journal area in units of `fsync()`. When `fsync()` is called, the EXT4 filesystem triggers the JBD [16] daemon that operates the journaling of EXT4. The JBD daemon issues the metadata in the transaction into the storage after the data blocks in the file are transferred to the writeback cache in the storage. To check whether the metadata is successfully committed, the JBD daemon issues the journal commit record after the metadata is written into journal area. The commit record is the block to ensure that the corresponding transaction commits successfully. If a sudden system crash occurs due to factors such as power failure during journaling, EXT4 attempts to roll-forward recovery to the disk area using the metadata written in the journal area. In this case, not all the metadata blocks are recovered in the disk area, but only the metadata in the transaction in which the commit record is successfully recorded is recovered. In order to ensure the accuracy of such a recovery operation, the additional FLUSHing occurs because it is necessary to ensure that the metadata in the transaction is recorded persistently before the commit record block.

On the other hand, F2FS provides the more efficient logging mechanism for the recovery against system crash situations. First, F2FS supports the roll-back recovery through the checkpoint technique. F2FS configures the filesystem metadata (e.g, NAT, SIT, SSA, etc.) into a Checkpoint Pack during the checkpoint process and writes it permanently in storage. F2FS provides a

recovery point of the filesystem by performing the checkpoint periodically (default: 30 seconds). If a system crash occurs, the roll-back recovery of F2FS will recover the filesystem with the latest checkpoint pack. In addition, F2FS performs the roll-forward recovery based on the node segment processed through `fsync()` and the latest checkpoint pack information.

Therefore, in `fsync()` of F2FS, only the data segment and node (metadata of the file) segment of the file are written to the storage. If the checkpoint is not triggered during `fsync()`, the write requests for the metadata are not issued. Therefore, unlike the EXT4 file system, the `fsync()` of F2FS issues the FLUSH request after the write requests for the data and node segment are processed.

3. `fsync()` Performance Degradation Problem in F2FS

We found that the `fsync()` performance decreases when both F2FS and Supercap-SSD are used. We analyzed this performance degradation problem of `fsync()`, and confirmed that the cause of this problem is the scanning overhead to find the free node ID in the Node ID bitmap.

F2FS manages the node (metadata information in the file), and the location information of each node is managed by filesystem metadata called NAT (Node Address Table). In F2FS, the free nids run out quickly in case of the intensive node allocation. Therefore, F2FS caches the NAT entry in the memory to prevent the filesystem from generating the I/O for NAT in the storage whenever the location information of nodes in the file system is changed [17]. However, in the case of the intensive node allocation such as the Varmail workload, the number of cached NAT entries increases rapidly in the course of the workload operation, resulting in exceeding the threshold value of cached NAT entries (DEF_NAT_CACHE_THRESHOLD). If the number of the cached NAT entry is more than the threshold value, `fsync()` triggers `f2fs_balance_fs_bg()`. This function reduces the number of the cache NAT entries less than the threshold value by deleting the cached NAT entries as many as 4 KB block through the FIFO (First-In, First-Out) algorithm.

However, `f2fs_balance_fs_bg()` performs the additional operation as well as the control of the cache NAT entry. This function searches the free nids (node IDs) in the cached NAT entry using `free_nid_bitmap` (node ID bitmap array), and inserts the free nid into `nid_list [FREE_NID_LIST]`, the list of the free nids. Furthermore, in order to insert the free nids searched through `free_nid_bitmap` into `nid_list`, F2FS should scan `nid_list [FREE_NID_LIST]` because the free nids searched through `free_nid_bitmap` might already be located in `nid_list [FREE_NID_LIST]`. Immediately, although all free nids that F2FS searches through `free_nid_bimap` are already located in `nid_list [FREE_NID_LIST]`, F2FS should scan the `nid_list [FREE_NID_LIST]` every free nid. Until the number of the cached NAT entries decreases below the threshold value, `fsync()` in F2FS triggers `f2fs_balance_fs_bg()` with the scanning operation for the free node ID. Consequently, if using Supercap-SSD in F2FS, the overhead at the FLUSH command decreases, but it causes the serious scanning overhead because the intensive node allocation makes the `fsync()` system call trigger `f2fs_balance_fs_bg()`.

We performed the experiment to confirm the `fsync()` performance degradation problem. Figure 3, 4 exhibit the trend of `fsync()` latency on EXT4 and F2FS respectively when performing the Varmail workload (`fsync()`-intensive workload) in Filebench. In EXT4, the `fsync()` latency is maintained consistently over the runtime of Varmail.

On the other hand, F2FS maintains the lower latency than EXT4 in the beginning. However, `fsync()` latency of F2FS is increased after 20 seconds. We confirmed the time when `fsync()` latency is increased is identical with the time when F2FS starts the scanning operations. In other words, scanning operation in F2FS decreases the performance of `fsync()`. The average `fsync()` latency after the scanning operation increases by 2.16X against the average `fsync()` latency before the scanning operation (183 usec vs. 396 usec).

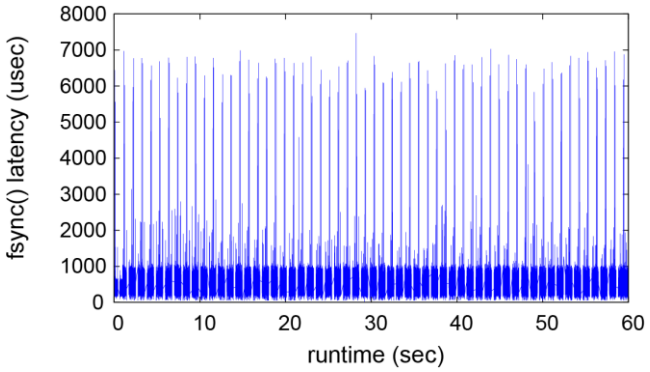

Figure 3. `fsync()` latency in EXT4

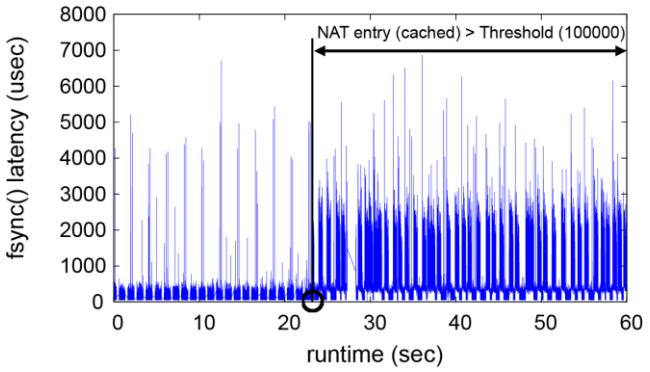

Figure 4. `fsync()` latency in F2FS

4. EXPERIMENTS

We performed the experiments to confirm the effect of the `fsync()` performance degradation problem in F2FS. The experiments were conducted on the EXT4 (Ordered mode) and F2FS in Linux 4.14-rc5 (Ubuntu 14.04). For the evaluation, we used the PC Server with Intel Core i7-4790 (4 cores), 4 GB Memory, and Samsung SM843 480GB (Supercap-SSD). We repeated all experiments 10 times, and calculated the average of the results.

4.1 Microbenchmark

We used `Mobibench` [18,19] to measure the performance of `fsync()`. We set the two workloads: 4 KB Sequential Write and 4 KB Random Write (Overwrite) to the 10GB-sized files. `fsync()` is called per each 4 KB write.

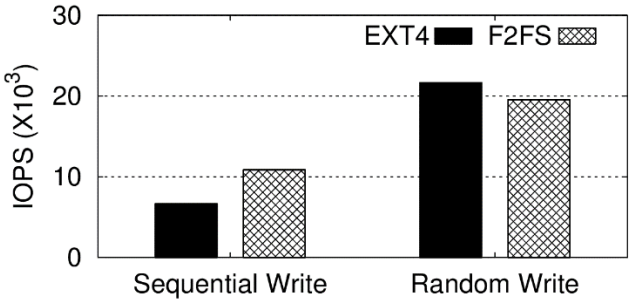

Figure 5. Microbenchmark (4 KB `write()` followed by `fsync()`)

Figure 5 exhibits the result of 4KB Write followed by `fsync()` on the EXT4 and F2FS filesystems. In this experiment, the performance degradation of `fsync()` does not occur because the amount of the node allocation is not sufficient for the threshold of NAT cache. When EXT4 performs `fsync()` in the sequential write workload, it writes the additional journal log (e.g. journal descriptor, journal commit record) as well as the data in the files, and the metadata. Furthermore, `fsync()` in EXT4 performs the additional FLUSH command against `fsync()` in F2FS because of the reliability of the transaction commit. Consequently, F2FS exhibits 1.63X (6,656 IOPS vs. 10,872 IOPS) performance against EXT4.

On the other hand, in the random write workload, EXT4 exhibits 1.11X (21,654 IOPS vs. 19,550 IOPS) performance against F2FS. This is because most `fsync()`s of EXT4 does not have the dirty metadata in the transaction. `fsync()` in EXT4 is performed like `fdatasync()` if there is no metadata submitted into the storage. Random **Overwrite** only changes the timestamp [20] information in the files [21]. Therefore, if the several `fsync()`s are called in the same time interval, the `fsync()`s do not call the journaling thread excluding one `fsync()`. Consequently, in random write, most `fsync()`s in EXT4 are performed like `fdatasync()`. `fdatasync()` only issues the data block, and calls one FLUSH command. On the other hand, `fsync()` of F2FS always issues the data and node segments despite the random write workload. On this account, the performance of the random write in EXT4 is higher than F2FS.

4.2 Server Workload

We used the Filebench - Varmail workload among the server workloads to show the `fsync()` performance degradation in F2FS. Varmail is `fsync()`-intensive workload [7]. In Varmail, 16 threads perform the working set: `create()`, `unlink()`, `read()`, `write()`, `fsync()`, and etc on the 1000 files in a directory.

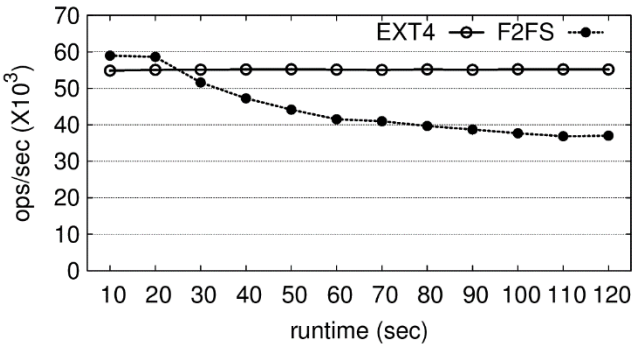

Figure 6. Server Workload (Filebench - Varmail)

Figure 6 exhibits the performance of the Varmail workload on the runtime in EXT4 and F2FS. F2FS has the higher performance than EXT4 before the 30 sec. This is because the additional journal write occurs in the EXT4 filesystem. However, the performance of F2FS continuously decreases after the 30 seconds while the performance of EXT4 is consistent. Consequently, in the 120 sec, the performance of F2FS is 0.67X against EXT4. This experimental result shows that the scanning operation in `fsync()` decreases the performance of F2FS.

5. CONCLUSION

In this paper, we analyzed the `fsync()` mechanism in F2FS. Based on the analysis, we investigated the cause of the `fsync()` performance decreases in F2FS with Supercap-SSD, and finally confirm the scanning overhead of the free node IDs becomes the bottleneck in `fsync()` of F2FS. Furthermore, we prove the correlation between the FLUSH overhead in `fsync()` and the scanning overhead for the free node IDs of F2FS. If the FLUSH overhead diminishes, the scanning overhead in F2FS increases because F2FS will allocate the nodes more quickly. Finally, we confirm the impact of the `fsync()` Performance Degradation by performing the several experiments. The `fsync()` latency increases by 2.16X, and the performance of F2FS is 0.67X against EXT4 due to the scanning overhead.

6. ACKNOWLEDGMENTS

This work was supported by the BK21 plus program through the National Research Foundation (NRF) funded by the Ministry of Education of Korea, the ICT R&D program of MSIP/IITP (R7117-16-0232, Development of extreme I/O storage technology for 32Gbps data services), Basic Research Laboratory Program through the National Research Foundation (NRF) funded by the Ministry of Science, ICT & Future Planning (MSIP) (No. 2017R1A4A1015498), and the MSIP(Ministry of Science, ICT&Future Planning), Korea, under the ITRC(Information Technology Research Center) support program (IITP-2016-H8501-16-1006) supervised by the IITP(Institute for Information&communications Technology Promotion).

7. REFERENCES

[1] Steigerwald, M., "Imposing order." *Linux Magazine,* (May, 2007).

[2] Chidambaram, Vijay, et al. “Optimistic crash consistency.” Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP 13, 2013, doi:10.1145/2517349.2522726.

[3] Sehgal, P., Tarasov, V., and Zadok, E. (2010, February). Evaluating Performance and Energy in File System Server Workloads. *In Proceedings of USENIX Conference on File and Storage Technologies* (pp. 253-266).

[4] Lee, C., Sim, D., Hwang, J. Y., and Cho, S. (2015, February). F2FS: A New File System for Flash Storage. *In Proceedings of USENIX Conference on File and Storage Technologies* (pp. 273-286).

[5] Rosenblum, Mendel. “Log-Structured file systems.” The Design and Implementation of a Log-Structured file system, 1995, pp. 53–85., doi:10.1007/978-1-4615-2221-8_4.

[6] Rodeh, Ohad, et al. “Btrfs.” ACM Transactions on Storage, vol. 9, no. 3, Jan. 2013, pp. 1–32., doi:10.1145/2501620.2501623

[7] McDougall, R., and Mauro, J. (2005). FileBench. *URL: http://www. nfsv4bat. org/Documents/nasconf/2004/filebench. Pdf*

[8] Prabhakaran, V., Bairavasundaram, L. N., Agrawal, N., Gunawi, H. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2005). IRON file systems. *In Proceedings of ACM Symposium on Operating Systems Principles* (Vol. 39, No. 5, pp. 206-220).

[9] Girish S. Journal Checksums. (May, 2007). URL: http://wiki.old.lustre.org/images/4/44/Journal-checksums.pdf

[10] Kim, Hyeong-Jun, and Jin-Soo Kim. “Tuning the Ext4 Filesystem Performance for Android-Based Smartphones.” Frontiers in Computer Education Advances in Intelligent and Soft Computing, 2012, pp. 745–752., doi:10.1007/978-3-642-27552-4_98.

[11] Jonathan C. Barriers and journaling filesystems. URL: http://lwn.net/Articles/283161/

[12] Narayanan, Dushyanth, et al. “Write off-Loading.” ACM Transactions on Storage, vol. 4, no. 3, Jan. 2008, pp. 1–23., doi:10.1145/1416944.1416949.

[13] Guo, Jie, et al. “Low Cost Power Failure Protection for MLC NAND Flash Storage Systems with PRAM/DRAM Hybrid Buffer.” Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013, 2013, doi:10.7873/date.2013.181.

[14] Stephen C Tweedie. “Journaling the Linux ext2fs filesystem.” Proceedings of Annual Linux Expo, 1998.

[15] Wilson, A. “The new and improved Filebench.” Proceedings of the USENIX FAST 2008 (San Jose, CA, USA, Feb 2008)

[16] Park, Stan, et al. “Failure-Atomic msync().” Proceedings of the 8th ACM European Conference on Computer Systems - EuroSys 13, 2013, doi:10.1145/2465351.2465374.

[17] Chao Yu, [PATCH] f2fs: introduce free nid bitmap. URL:https://lkml.org/lkml/2017/2/21/131

[18] Sooman J., Kisung L., Jungwoo H., Seongjin L, Youjip W. AndroStep: Android Storage Performance Analysis Tool. *The 1st European Workshop on Mobile Engineering* (Aachen, Germany, February 26).

[19] Kisung Lee, Mobile Benchmark Tool (MOBIBENCH). URL: https://github.com/ESOS-Lab/Mobibench

[20] Yupu Zhang, Chris Dragga, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. “*-Box: Towards Reliability and Consistency in Dropbox-like File Synchronization Services.”, Proceedings of USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage), 2013

[21] Son, Hankeun, et al. “Coarse-Grained mtime update for better fsync() performance.” Proceedings of the Symposium on Applied Computing - SAC 17, 2017, doi:10.1145/3019612.3019739.

Authors’ background

Your Name	Title*	Research Field	Personal website
Gyeongyeol Choi	Master student	Filesystem Block Device Layer Storage	
Youjip Won	Full professor	Operating system Filesystem New Memory Storage Block Device Layer	