When F2FS Meets Address Remapping

Yongmyung Lee*, Jong-Hyeok Park*, Jonggyu Park*, Hyunho Gwak*,
Dongkun Shin*, Young Ik Eom**, Sang-Won Lee*
*Sungkyunkwan University
** Dept. of Electrical and Computer Engineering College of Computing and Informatics,
Sungkyunkwan University
{feellym,akindo19,jonggyu,gusghrhkr,dongkun,yieom,swlee}@skku.edu

ABSTRACT

While gaining popularity in mobile devices, F2FS, a flash-friendly variation of log-structured file system, reveals three drawbacks: segment cleaning overhead, metadata update overhead, and file fragmentation, which becomes conspicuous under random update workloads. This paper suggests for the first time to leverage the address-remap technique in flash storage to remedy such pitfalls in F2FS. Our approach can, while preserving the benefit of log-structured writes, achieve the eventual effect of in-place update, completely preventing three drawbacks of F2FS. It can thus significantly outperform ext4 as well as vanilla F2FS under random update workloads. Armed with another write mode, F2FS will become competitive for a wider range of applications.

CCS CONCEPTS

• **Information systems → Storage architectures**; Flash memory; • **Software and its engineering** → File systems management.

KEYWORDS

F2FS, Address-Remap

ACM Reference Format:
Yongmyung Lee*, Jong-Hyeok Park*, Jonggyu Park*, Hyunho Gwak*,, Dongkun Shin*, Young Ik Eom**, Sang-Won Lee*. 2022. When F2FS Meets Address Remapping. In *14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '22), June 27–28, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3538643.3539755

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 ACM 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 '22, June 27–28, 2022, Virtual Event, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9399-7/22/06…$15.00
https://doi.org/10.1145/3538643.3539755

1 INTRODUCTION

Owing to its flash-awareness, the F2FS filesystem [8] has gotten the spotlight in various flash-memory storage systems. In particular, smartphone manufacturers have launched products employing F2FS to access universal flash storage devices. Compared to the in-place-update (IPU) policy of legacy file systems (*e.g.* , ext4), the out-of-place-update (OPU) policy of F2FS enables to better exploit the performance potential of the flash storage by transforming many small random writes into a single large sequential one [17].

However, the OPU strategy incurs several drawbacks. First, OPU generates invalid blocks at each update operation. To reclaim the invalid blocks, F2FS should perform filesystem-level garbage collection, known as segment cleaning. Second, OPU requires additional metadata updates to keep track of the up-to-date data location because the corresponding block locations are changed by update operations. Finally, even when a file has initially been allocated with sequential blocks, its blocks can be disassembled, so called file fragmentation, when the file is randomly updated. The fragmented file will show a poor sequential read performance, since many small I/O requests must be transferred to the storage device [14].

To remedy such intrinsic problems of OPU in F2FS, this paper suggests to exploit the address-remap (AR) technique in flash storage, which modifies the internal address mapping table of the flash storage to change the logical block address (LBA) of written data. Although the AR technique has been used to mitigate various filesystem overheads such as journaling [5, 7, 19] and segment cleaning [7, 15, 21], those studies apply the AR technique only to a particular component of the filesystems, instead of generic write operations.

In contrast to the previous researches, we present RM-IPU (Remap-based In-Place Update), which incorporates AR into F2FS to relieve the drawbacks with OPU. Data blocks are first written in the OPU manner and then are changed as if they are updated in place. Specifically, RM-IPU first stores the updated data in a log-structured manner and modifies the device-internal mapping table so that the old filesystem blocks seamlessly point to the updated data. In this way, RM-IPU achieves the benefit of the OPU block allocation, which

Figure 1: Random Write Performance

generates sequential writes, and the IPU policy, which maintains the filesystem-level contiguity of the original blocks and eliminates metadata update. Furthermore, since RM-IPU immediately invalidates newly allocated filesystem blocks for the updated data after AR, it can minimize the segment cleaning overheads.

In summary, the RM-IPU approach is novel in that it keeps F2FS's strength (*i.e.* , flash-friendly sequential write) but at the same time can overcome three weaknesses resulting from the OPU policy in F2FS. As a result, RM-IPU will provide both benefits of IPU-based and OPU-based filesystems. We believe RM-IPU facilitates the wide adoption of F2FS in practice by gracefully handling random update workloads, which are pervasive in database systems.

2 BACKGROUND AND MOTIVATION

2.1 F2FS: Basics

F2FS is a variant of log-structured filesystem, which is renovated with flash-awareness. F2FS divides the storage space into segments, and multiple consecutive segments compose a section that are designed to align with the erase blocks of the underlying flash storage. Also, F2FS adopts log-structured block allocation with OPU, thereby reshaping small random writes into a large sequential one. However, due to such design choices, F2FS inevitably shows several drawbacks.

First, F2FS should collect stale data to secure free space because it does not allow IPU. Next, the location of data keeps changing whenever the data are updated. Thus, F2FS should re-write the corresponding metadata blocks to correctly point to the up-to-date location of data. Finally, F2FS is vulnerable to fragmentation because it allocates blocks in a log-structured manner without IPU. For example, suppose that two threads simultaneously update existing data. Due to the OPU policy, those data will be appended at the end of the log, thereby being apart from the other blocks of the files. Moreover, if the update operations involve synchronous operations, those data from different threads interleave in the same segment because of the log-structured block allocation.

There has been previous research to mitigate the aforementioned problems. First, SSR (Slack Space Recycling)[12] reuses invalid blocks without performing segment cleaning to eliminate the segment cleaning overheads. However, it generate small-sized random writes to fill up the holes. Also, SSR still requires metadata updates since the data location is still modified. Second, delayed allocation defers block allocation to the flush time and merges multiple blocks from the same file into a contiguous fragment. However, synchronous operations dilute the effectiveness of delayed allocation since they force the corresponding data to be flushed immediately.

2.2 Drawbacks of F2FS

In this section, we experimentally demonstrate the aforementioned three drawbacks of F2FS.

Segment Cleaning Overheads To quantitatively measure the segment cleaning overheads of F2FS, we perform a random write experiment with Samsung 970 evo 250GB. First, we fill up half of the storage space with dummy files. Second, we repetitively generate multi-threaded 4KiB random writes while issuing *fsync* every 10 operations. We conduct experiments using ext4 filesystem and two F2FS variants configured with default (denoted F2FS) and lfs mode (denoted F2FS-L) mount option which employs adaptive logging and append logging, respectively. Figure 1 shows the bandwidth of the random write operations. At first, F2FS outperforms ext4 by 42% before 100GiB data are written, because F2FS reshapes small random writes into larger sequential ones. Specifically, the average I/O size of F2FS is 41.2KiB whereas that of ext4 is 4.2KiB. However, as the segment cleaning occurs, the performance of F2FS begins to decrease significantly due to the increased amount of filesystem I/Os induced by the segment cleaning. As a result, F2FS shows 40% lower bandwidth than ext4 after 100GiB are written. Although the SSR method of F2FS mitigates the segment cleaning overheads, it still suffers from performance degradation because it incurs small random writes and still generates metadata update operations.F2FS-L, like F2FS, shows a better performance before 100GiB data are written, but after that, the performance decreased significantly due to segment cleaning overhead, and showed 53% lower performance than ext4.

Metadata Update Overheads As mentioned before, F2FS suffers from frequent metadata updates because the OPU policy of F2FS changes the location of data whenever the data are updated. To measure the amount of metadata additionally updated, we create 1GiB files and perform 4KiB random writes to the files while measuring the amount of writes at the block layer. Here, we also use buffered I/Os and issue *fsync* every 10 operations. As a result, F2FS requires around 1.8 times more amount of writes for the same amount of data, compared with ext4. This difference comes from the fact that F2FS continuously updates the metadata to refer to the corresponding data at every update while the location of

data is invariable on ext4 due to its IPU. Specifically, F2FS generates 1.02GiB of metadata writes whereas the amount of metadata writes on ext4 shows only 0.12GiB.

Fragmentation Overheads Since F2FS allocates new data blocks in a log-structured way without IPU, it appends all the updated blocks, instead of maintaining the original data layout. Therefore, F2FS experiences severe file fragmentation under update-heavy workloads [13]. To analyze this overhead, we first create a single 8GiB file and measure the sequential read performance after issuing 2GiB random writes. In terms of throughput, F2FS shows around 43% lower performance than ext4 because it generates more I/O requests for the same amount of data due to file fragmentation. Specifically, F2FS requires 42.5 times more I/O requests than ext4. In the meantime, the mean request size of ext4 is 733KiB while that of F2FS is only 17KiB.

Overall, the flash-awareness of F2FS such as log-structuring shows performance benefits over ext4 when it comes to random writes. However, in order to maintain flash-awareness, F2FS is inevitably taxed in the form of segment cleaning, metadata overheads, and fragmentation. In addition, as shown in Section 4, F2FS suffers from excessive device-level physical write amplification for random updates, which contradicts to the belief about F2FS.

3 REMAP-BASED IN-PLACE-UPDATE

Key Idea To mitigate the drawbacks of log-structured writes such as segment cleaning and metadata update overhead, we leverage the address remap technique in the write process of F2FS. To be concrete, as shown in Figure 2, once new update blocks are successfully written to the storage in log-structured manner, we issue the address remap command to modify the device-internal mapping table so that the LBAs of old blocks point to the physical pages of new blocks in flash storage. In this way, while keeping the benefit of flash-friendly write pattern for random updates, F2FS can achieve the effect that new segment writes for update blocks are not made and accordingly need not update the metadata (*i.e.* , direct-index blocks). In addition, the logical address contiguity of file objects will be preserved despite random updates. Once this idea is properly embodied, RM-IPU will have the benefits of both in-place-update and out-of-place file systems: fast random writes, no file system-level write amplification, and better sequential read bandwidth.

How RM-IPU Works: An Illustration Let us illustrate how RM-IPU works using Figure 2. Assume a file consists of four data blocks (D0 - D3) whose current LBAs are LBA#12 - LBA#15, respectively. Provided that newly appended blocks are located at cold segment, all blocks of D0, D1, D2, and

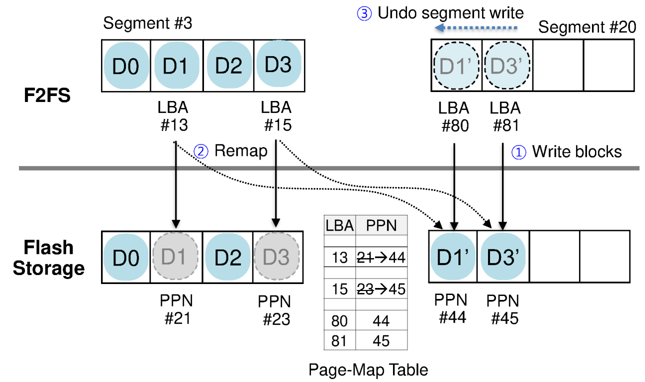

Figure 2: RM-IPU : An Illustration

D3 are stored in a cold segment (*i.e.* , segment #3 in Figure 2). Then, for new update blocks for D1 and D3 (*i.e.* , D1' and D3'), those two update blocks will be now directed to a hot segment, segment #20 (① in Figure 2). For these updates, RM-IPU keeps tracks that their original and new LBA pairs (LBA#13, LBA#80) and (LBA#15, LBA#81) . Once all the blocks are durably written to the storage, the address mapping for LBA#13 and LBA#15 are changed to point to new physical pages for D1' and D3' by calling the address-remap command (② in Figure 2). It should be noted that, unlike vanilla F2FS, the direct-index block need not be updated. In this way, for D1 and D3, even though updated, their old LBAs remain intact. This in turn indicates that the metadata for the updated data block D1 and D3 need not be updated.

In summary, the address-remap in RM-IPU is, once new update blocks are persistently written to segment in a log structured manner, used to revert the effect of appending data blocks to segment (③ in Figure 2). The reverted segment space will be reused to store new update blocks.

Write and Address Remapping Writes operations are categorized into two types: append writes and update writes. The append writes indicate that new blocks are appended at the end of a file object, while the update writes overwrite old blocks with new blocks. RM-IPU performs the original append logging of F2FS in the case of append writes and exploits the address remap command to deal with update writes. For this reason, RM-IPU is expected to perform best for workloads with small random updates for files (*e.g.* , OLTP databases with high fsync overhead for direct-index blocks). RM-IPU also assumes the multi-head logging strategy is used in F2FS [8], which places data blocks into different segment logs according to their hotness. In the case of append writes, they are allocated to cold segment and will remain there until updated. On the other hand, RM-IPU stores the updated writes in the hot segments without metadata update since they become invalidated immediately after the corresponding AR is successfully performed. Therefore, RM-IPU can

minimize the segment cleaning overhead as well as metadata update overhead.

We observe that the current RM-IPU design and implementation can cause the consistency problem upon system crashes. In particular, RM-IPU which assumes the roll forward recovery in F2FS [8] will not recover the consistent state when a crash is encountered after calling the remap commands for update writes and also writing the metadata blocks for append writes. This issue is left as future work.

Segment Cleaning Both old and new blocks involved in an address-remapped command should not be garbage-collected by F2FS until the command completes inside storage. Otherwise, data corruption may occur since the remap command may point to an invalid LBA. Therefore, RM-IPU prevents such blocks from being garbage-collected (*i.e.* , segment cleaning) by keeping track of all the segments containing any address-remapping blocks and discarding them from victim candidates.

F2FS Recovery and RM-IPU While rebooting from crashes, F2FS checks the latest valid checkpoint and performs the roll-back recovery to this point. Afterward, F2FS performs the roll-forward recovery, which searches for metadata blocks (i.e., node blocks of F2FS) with a fsync mark written after the checkpoint. In this way, it can restore to the checkpointed state with additional updates of fsynced data.

On the other hand, the current version of RM-IPU follows the recovery mechanism of ext4 ordered-mode and F2FS with O_DIRECT and IPU mode. If a system crash occurs while performing writes followed by fsync, RM-IPU may recover certain data even without fsync completion if its remap operation has been completed. For example, suppose an application issues write operations followed by fsync. First, RM-IPU (1) allocates new blocks, (2) performs remap operations, (3) writes corresponding metadata, and (4) returns fsync completion to the application. Here, if a system crash takes place between (2) – (3), RM-IPU recovers the updated data although the fsync completion is not return to the application. This recovery mechanism is different from the original F2FS in that F2FS negates such updates by restoring the filesystem state to the latest checkpoint. To follow the recovery semantic of F2FS or ext4 data journal mode, we can add a special module to invalidate the remap operations.

4 PERFORMANCE EVALUATION

In this section, we present performance evaluation of our implementation of RM-IPU under various workloads. Through experiments, we confirm the following:

- RM-IPU can significantly improve random write performance by reducing segment cleaning overhead.
- RM-IPU enables sequential continuity in F2FS, which genuinely achieves the aim of LFS and avoids scattered read.

4.1 Evaluation Setup

All the experiments were conducted with FEMU, a QEMU based NVMe SSD emulator [10] to which we added the address-remap functionality. We run FEMU on Intel i7 CPU 3.40GHz processor and 64GB DRAM. The kernel of the FEMU host system is Linux 4.19 and a 32GB SSD with 4GB over-provisioning space (12.5 %) is emulated. The flash page size is set to 4KB and the page read, program, and erase latencies are 50us, 500us, and 5ms, respectively. In this experiment, we assume that reverse L2P mapping table is maintained at the byte-addressable non-volatile memory to guarantee the consistency of address remapping. Hence, we imposed additional latency(*i.e.* 0.1us) per mapping table entry. In addition, the volatile write cache option [11] and the discard command [18] in FEMU are enabled to reflect the fsync() overhead and to reduce SSD's internal write amplification, respectively. To analyze the effectiveness of RM-IPU, we experience the ext4 filesystem and two F2FS variants mentioned in section 2.2 above.

4.2 Workloads

We use a synthetic `fio` workload and realistic workload, TPC-C on MySQL 5.7 to verify the efficacy of RM-IPU.

Synthetic Workload The FIO benchmark tool [2] is used to demonstrate the advantages of RM-IPU for random writes. we created four 4GB files and then measured the amount of writes at the block layer while running four threads, each of which performs 4KiB random writes to the files. This configuration is similar to the one used in Section 2.

Realistic Workload The TPC-C benchmark [16] is used to understand the benefit of RM-IPU against random write intensive OLTP workload. The benchmark was running 16 clients against each of initial database of 200 warehouses with 16KB page size.

4.3 Performance Analysis

We explain the overall performance benefit of RM-IPU using Figure 3 and 4. While running the workloads, we measured bandwidth (throughput in Figure 4) as well as filesystem and FTL WAF (denoted as **FS-WAF** and **FTL-WAF**, respectively).

F2FS Drawbacks As mentioned in Section 2.2, F2FS suffers from considerable write amplification due to chronic drawbacks of LFS which of excessive use of logical space. Figure 3 shows that F2FS and F2FS-L fall to performance degradation when the free blocks run out (❶ in Figure 3). This indicates that LFS exacerbates the FTL-WAF more quickly due to the metadata overhead. After securing the free blocks,

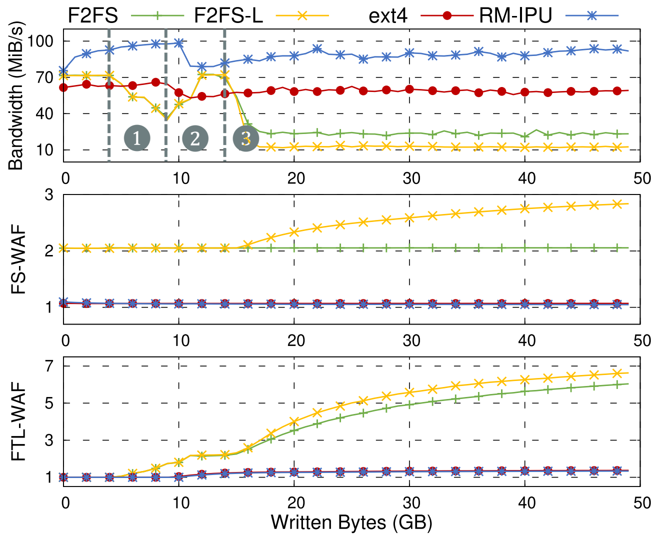

Figure 3: Synthetic Workload (FIO)

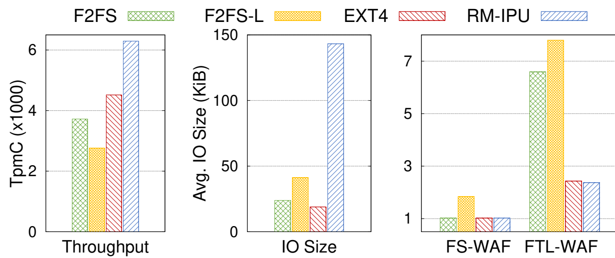

Figure 4: Realistic Workload (TPC-C)

the performance is recovered (❷ in Figure 3). However, the throughput deteriorates again (❸ in Figure 3) because all free segments are exhausted. This comes with the cost of random writes arising from valid pages in victim blocks for F2FS and segment cleaning overhead for F2FS-L. We also observed a log-on-log problem [9, 20] that worsen the FTL-WAF because the out-of-update policy of F2FS uses excessive space, which is still considered valid pages on SSD. Although the SSR alleviates the segment cleaning overheads, it is unable to resolve the performance degradation problem because it incurs small random writes and experiences severe metadata update overhead under the random write-intensive workload. ext4 also achieves ideal WAF in virtue of in-place-update manner data layout and apt use of discard command. However, ext4 experiences severe performance degradation due to the small random writes and thus RM-IPU outperforms upto 1.4x in terms of throughput.

In the realistic workload experiment, we found two distinct differences from synthetic workload. First, the overall throughput gap in the TPC-C benchmark is much smaller

Table 1: request size and count in sequential read

	F2FS	ext4	RM-IPU
Mean request size (KiB)	17	740	737
Request count	498K	11K	11K

than the FIO benchmark. Because the read-to-write ratio of TPC-C benchmark is 1.9:1 [3] and all methods are similar in the light of random read operation. In particular, the random write performance gain from buffer hits is amortized. Second, the FIO benchmark fills only half of the storage space but as the SSD was filled up with the ever-growing database in the TPC-C benchmark, the FTL-WAF of both ext4 and RM-IPU is increased up to 2.5 and 2.4 respectively. Especially, the FS-WAF of F2FS metadata update overhead The IO size in Figure 4 explains the performance benefit of RM-IPU. Both ext4 and F2FS show small random write size, particularly, SSR mode in F2FS gives rise to excessive small random writes. F2FS-L suffers from writes induced by the segment cleaning. Conversely, RM-IPU invalidates blocks that are randomly updated using address remapping, eliminating segment cleaning overhead and limiting the file fragmentation in F2FS.

Sequential Continuity Now we examine how RM-IPU can remedy the fragmentation problem in F2FS. Table 1 shows the mean request sizes and counts under sequential read operations with 1MB block size following a random write of 8GB files. Due to the file fragmentation, F2FS shows a small size of 17 KiB with a considerable number of requests because it experiences severe file fragmentation. However, RM-IPU retains similar size and number of request with ext4's. This indicates that address remapping functionality of RM-IPU can prevent fragmentation proactively which does not incur unnecessary data copying. We also confirmed that the same performance gap (*i.e.* 43%) between F2FS and RM-IPU which is consistent with the experimental results using commercial SSD in Section 2.2.

To sum up, experimental results confirmed that RM-IPU reduces the segment cleaning overhead and prevents the file fragmentation problem under the random write-intensive workloads, and thus it resolves the drawbacks of F2FS.

5 FUTURE DIRECTIONS

We have so far shown that the address remapping technique can be used to remove a few intrinsic drawbacks of F2FS particularly under the workloads with small random updates. Though, several key technical issues have to be resolved for our RM-IPU approach to be viable and also there are a few opportunities for further optimizations, as listed below.

Evaluations By evaluating using comprehensive workloads, we need to investigate the benefits and limitations of RM-IPU. In addition, it would be interesting to compare our RM-IPU with remap-based segment cleaning scheme [7, 21]. While RM-IPU invokes remap eagerly just after writes and thus aims at eliminating segment cleaning, the latters applies remap lazily upon segment cleaning.

Atomicity and Durability We will investigate how the in-storage address-remap command can be exploited to accelerate various performance-critical operations in F2FS such as checkpoint, fsync, and atomic write. For example, RM-IPU will enable to achieve the atomic propagation of multiple blocks with less DRAM resource and storage than the existing implementation in F2FS [4]. It is also worth investigating whether RM-IPU can easily support transactional features such as multi-file updates, shadow garbage collection, and stolen pages [1], with minimal change in F2FS codebase.

To Remap or Not The benefit of RM-IPU will stand out for small-sized random updates, but its overhead for the address-remap command exists. Thus, its benefit and overhead need to be judiciously traded. For instance, for large batch update (*e.g.* , > 64KB), the benefit by address-remap can be outweighed by its run-time overhead inside flash storage. As another example, when a file is already heavily fragmented, the benefit of RM-IPU is unclear.

Adaptive Write Mode Selection in F2FS F2FS currently supports three write modes of append logging (LFS), threaded logging (SSR), and in-place-update (IPU)[6]. RM-IPU can be regarded as a new write mode, remap-based eventual IPU, which works well for random update workloads. Armed with four write modes, F2FS will be competitive for more diverse workloads. To realize its full potential, it needs to adaptively select the best write mode for each workload. Furthermore, we can imagine more fine-grained write mode that supports according to the per-file write pattern.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and our shepherd, Keith Smith, for their valuable comments and feedback. This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1802-07.

REFERENCES

[1] 2022. exF2FS: Transaction Support in Log-Structured Filesystem. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA.

[2] Axboe, Jens. 2022. Fio-flexible io tester. http://freecode.com/projects/fio.

[3] Shimin Chen, Anastasia Ailamaki, Manos Athanassoulis, Phillip B Gibbons, Ryan Johnson, Ippokratis Pandis, and Radu Stoica. 2011. TPC-E vs. TPC-C: Characterizing the new TPC-E benchmark via an I/O comparison study. *ACM Sigmod Record* 39, 3 (2011), 5–10.

[4] Seungyong Cheon and Youjip Won. 2017. Exploiting Multi-Block Atomic Write in SQLite Transaction. In *Proceedings of the International Conference on High Performance Compilation, Computing and Communications*. 23–27.

[5] Hyun Jin Choi, Seung-Ho Lim, and Kyu Ho Park. 2009. JFTL: A flash translation layer based on a journal remapping for flash memory. *ACM Transactions on Storage (TOS)* 4, 4 (2009), 1–22.

[6] Jaegeuk Kim. 2012. F2FS. https://www.kernel.org/doc/Documentation/filesystems/f2fs.txt.

[7] Dong Hyun Kang, Gihwan Oh, Dongki Kim, In Hwan Doh, Changwoo Min, Sang-Won Lee, and Young Ik Eom. 2018. When Address Remapping Techniques Meet Consistency Guarantee Mechanisms. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*. 1–8.

[8] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. 273–286.

[9] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind. 2016. Application-Managed Flash. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. 339–353.

[10] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. 2018. The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. 83–90.

[11] NVM Express. 2019. NVM Express Revision 1.4. https://nvmexpress.org/developers/nvme-specification/.

[12] Yongseok Oh, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H. Noh. 2010. Optimizations of LFS with Slack Space Recycling and Lazy Indirect Block Update. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*. 1–9.

[13] Jonggyu Park and Young Ik Eom. 2020. Anti-Aging LFS: Self-Defragmentation With Fragmentation-Aware Cleaning. *IEEE Access* 8 (2020), 151474–151486.

[14] Jonggyu Park and Young Ik Eom. 2021. FragPicker: A New Defragmentation Tool for Modern Storage Devices. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 280—-294.

[15] Jonggyu Park, Dong Hyun Kang, and Young Ik Eom. 2016. File defragmentation scheme for a log-structured file system. In *Proc. ACM APSys*. 1–7.

[16] Percona-Lab 2008. tpcc-mysql benchmark. https://github.com/Percona-Lab/tpcc-mysql.

[17] Mendel Rosenblum and John K Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)* 10, 1 (1992), 26–52.

[18] Frank Shu. 2007. Notification of Deleted Data Proposal for ATA-ACS2. http://t13.org.

[19] Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2015. ANViL: Advanced Virtualization for Modern Non-Volatile Memory Devices. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. 111–118.

[20] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. 2014. Don't Stack Your Log On My Log. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)*. 1–10.

[21] You Zhou, Qiulin Wu, Fei Wu, Hong Jiang, Jian Zhou, and Changsheng Xie. 2021. Remap-SSD: Safely and Efficiently Exploiting SSD Address Remapping to Eliminate Duplicate Writes. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 187–202.