When Address Remapping Techniques Meet Consistency Guarantee Mechanisms

Dong Hyun Kang* Gihwan Oh* Dongki Kim† In Hwan Doh†
Changwoo Min‡ Sang-Won Lee* Young Ik Eom*
*Sungkyunkwan University †Samsung Electronics ‡Virginia Tech

Abstract

Guaranteeing crash consistency is still one of the most expensive operations in traditional file systems because it causes redundant writes in a journaling file system, excessive read/writes in a log-structured file system, and tree-wandering problem in a copy-on-write file system. In this paper, we argue that such overhead can be significantly reduced by leveraging address remapping technique, which is already essential in many flash SSD devices. We thoroughly explore the potential of address remapping technique to ease the cost of guaranteeing consistency in two traditional file systems (*i.e.*, Ext4 and F2FS) and one database system (*i.e.*, MySQL). In particular, we introduce address remapping-based techniques to guarantee consistency, for file system journaling (*i.e.*, SOJ and SDJ), segment cleaning (*i.e.*, SSC), and application-level data journaling (*i.e.*, SADJ). To evaluate the proposed techniques, we developed a PCIe SSD prototype, which exposes the address remapping capability to the upper layer as a `share` command. Our experimental results using the PCIe SSD with the `share` command confirms that the address remapping, though simple, is very effective in reducing the read/write amplification due to the conventional ways of guaranteeing consistency in the existing file systems and database applications.

1 Introduction

In recent years, address remapping techniques have raised a lot of interests in academia and industry because it opens new optimization opportunities in designing efficient consistency mechanisms [6, 11, 12, 17, 26]. Especially, these remapping techniques give performance benefits by reducing expensive consistency operations in file systems (*e.g.*, journaling, log-structured, and copy-on-write) and consistency-critical applications (*e.g.*, MySQL [3], SQLite [4], git, and vim).

Generally, to guarantee the consistency of file metadata, data blocks, and versions, modern file systems have heavily resorted to various techniques such as journaling, logging, and copy-on-write. Unfortunately, they suffer from heavy read/write amplification incurred by the mechanisms inherent in each scheme, including redundant write [1, 24], segment cleaning [13, 23], and tree wandering [22]. Furthermore, because they do not provide application-level crash consistency [15, 19–21, 25], many consistency-critical applications should implement their own idiosyncratic mechanisms to ensure the recovery of their data from unexpected crashes, which are, in some cases, still crash-vulnerable [20, 21, 27]. In these scenarios where several types of consistency should be guaranteed, the remapping technique opens up a new chance of achieving high data consistency with low performance overhead.

However, much research has not been conducted to orchestrate the benefits of the remapping technique across existing storage stacks, such as file system, block device, and internal storage layer; previous work has not yet explored how the remapping technique can be leveraged to application-level crash consistency. For example, JFTL [9] and ANViL [26] implemented the functionalities of atomic address remapping in internal storage and host block layer, respectively. They clearly confirmed that the address remapping is useful in various cases, such as single-write journaling, snapshot, file copy, and de-duplication. Unfortunately, previous studies did not consider the *application-level crash consistency* even though it is a common functionality necessary in modern consistency-critical applications. We observed that the consistency overheads in applications have essentially the same features with those in modern file systems.

In this paper, we perform a comprehensive study on leveraging address remapping technique in optimizing the file system-level and *application-level crash consistency* mechanisms. For our case studies, we utilized *SHARE* flash storage interface [17] that allows host programs to explicitly remap one or more pairs of LBAs atomically at the flash storage FTL layer, and imple-

mented *SHARE* by modifying the FTL firmware of the commercial high-end PCIe M.2 SSD, Samsung 960 EVO. We first present that the address remapping technique, *SHARE*, can significantly reduce the overhead of journaling file system and log-structured file system. In particular, we present two *SHARE*-based journaling schemes, namely SOJ and SDJ, on Ext4 [14] and present a *SHARE*-based segment cleaning scheme, namely SSC, on F2FS [13]. We also offer valuable insights about how to efficiently adopt *SHARE* for the application-level crash consistency. Our evaluation results confirm that the effect of the remapping technique is very promising. As an example, when Ext4 guarantees the application-level crash consistency using *SHARE*, the performance of an OLTP benchmark on MySQL/InnoDB DBMS is improved by 6.16 times over the default configuration.

2 Flash memory, FTL, and *SHARE*

An out-of-place update strategy is commonly used in a flash storage device because flash memory does not allow to update pages in place. Thus, to maintain the ever-changing mapping between logical and physical flash memory addresses, the flash storage device is equipped with a firmware module called FTL (flash translation layer) [10], which manages a mapping table between logical block address (LBA) and physical page address (PPA) in a page granularity. To leverage this indirection of page-level address mapping in flash storage, recently in database community, Oh *et al.* [17] proposed the *SHARE*. It exposes an interface that allows host applications to explicitly ask FTL to change its internal address mapping. To be concrete, as illustrated in Figure 1, upon receiving a `share` command from the host with a pair of two logical block addresses, *LBA2* and *LBA7*, as its parameter, FTL atomically changes the PPN (physical page number) of *LBA2* in its page-mapping table to that of *LBA7*, thus the latter physical page being shared by the former logical address. A `share` command can have an optional third argument, `length`, when the length of data to be shared is longer than the FTL mapping granularity (*i.e.*, 4KB). Though the description so far assumes that the `share` command is associated with a single pair of LBAs, it can have multiple LBA pairs in a batch [17].

3 File System-level Crash Consistency

The concept of *SHARE* is simple, but there are some design challenges in integrating it with the consistency mechanisms of existing file systems.

In this section, we show how two file systems, Ext4 and F2F2, can leverage the `share` command in supporting their data consistency with low overhead. For our studies, we ran all experiments on a system with a quad-core processor (Intel i7-6700) and 8GB memory, running Linux kernel version 4.6.7.

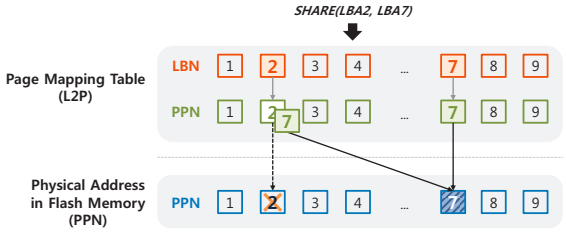
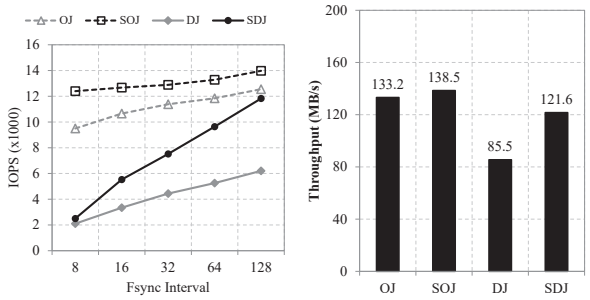


Figure 1: An illustration of *SHARE* command. Upon `share(LBA2, LBA7)`, FTL atomically remaps the PPN of LBA2 to that of LBA7 so that LBA2 and LBA7 share the same physical page.

Case Study 1: Ext4 Guaranteeing crash consistency is one of the most important functionalities in designing a file system. But, there is a trade-off between consistency level and performance. For this reason, the Ext4 file system takes a relaxed consistency level, i.e., the ordered journaling mode (OJ), as its default mode. The OJ mode provides *metadata consistency* [8], which only guarantees that metadata is entirely consistent and the data read by a file legitimately belongs to that file. Therefore, under the OJ mode, a file can point to the older version of its data, which is a source of the well-known *torn page* problem in database systems. In contrast, the full data journal mode (DJ) supports *version consistency* [8], where the metadata version is guaranteed to match to the version of the referred data. But, this higher-level consistency of the DJ mode comes at the expense of considerable performance degradation due to the double-write journaling of both data and metadata.

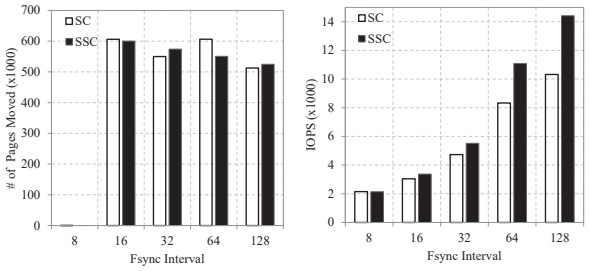
Recall that *SHARE* allows the host programs to explicitly and atomically change the address mappings being internally managed by FTL. With the help of *SHARE*, the DJ mode can achieve higher performance and consistency almost for free by offloading the burden of guaranteeing system-wide version consistency from the file system to the flash storage. In this paper, we design *SHARE*-aware ordered journaling (SOJ) and *SHARE*-aware data journaling (SDJ) by slightly modifying the existing journaling modes of Ext4. A key challenge in using the *SHARE* is how to track a set of LBA pairs of the journaled location and home location. Since a page is, once buffered in page cache, generally updated more than once before being flushed to its home location, it is very crucial to track a set of LBA pairs up to date; missing such LBA pairs might crash the file system even without power failure or system crash. To solve this challenge, we employ an auxiliary red-black tree, called A-tree (atomic-tree), where LBA pairs are inserted or updated whenever the relevant journal data is recorded on the journal location.

Now, let us explain two phases for guaranteeing crash consistency in SOJ and SDJ: commit and checkpoint. While the commit phase is almost same to that of Ext4,



(a) FIO **(b)** Varmail

Figure 2: The effect of *SHARE* on the journaling modes in Ext4.



(a) Number of pages moved **(b)** Performance

Figure 3: The effect of *SHARE* on segment cleaning in F2FS.

the checkpoint phase is in stark contrast with that of Ext4. When a commit operation is triggered, each write operation for journaling is first recorded in the journal area and then the relevant LBA pair is inserted into the A-tree for *SHARE*. At each checkpoint, we issue `share` commands by searching LBA pairs on the A-tree belonging to the checkpoint transaction. Then, upon receiving the `share` command, flash storage will carry out the atomic address remapping for a given set of LBA pairs so that the home locations are shared with the journaled locations. Finally, for the next checkpoint, the previous LBA pairs in the A-tree are discarded. In this way, SOJ and SDJ can avoid unnecessary overhead caused by redundant journaling writes.

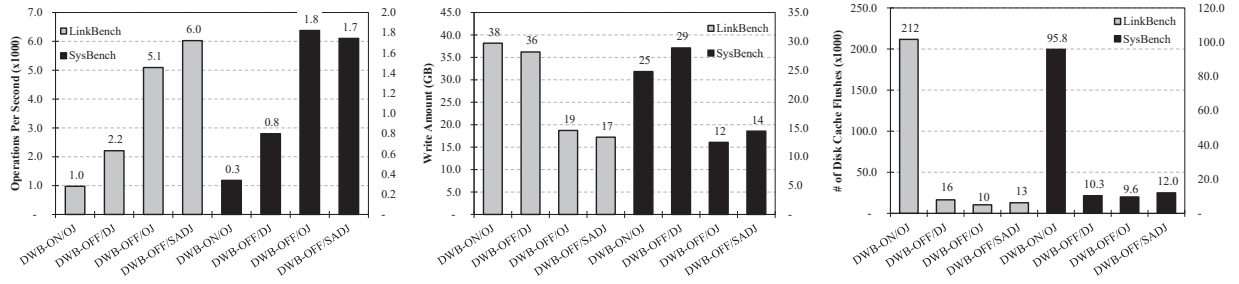
We implemented our design as a prototype in Linux kernel. Note that SOJ and SDJ do not necessitate any extra operations in employing the address remapping because they strictly adhere to the fundamental design principles of journaling in Ext4 and are implemented at the file system layer; meanwhile, ANViL [26] requires additional operations for both garbage collection and its metadata consistency at the virtualized block layer. We first evaluated *SHARE*-aware journaling modes using the FIO benchmark. In this experiment, the benchmark is configured to write 2GB data randomly to 10,000 files with a 8KB granularity. In order to test the effect of `fsync()` interval, which represents the number of write operations between two consecutive `fsync()` calls, we repeated the same experiment while varying `fsync()` interval.

Figure 2a presents the throughput in IOPS for our experiments. In Figure 2a, SOJ always has the highest performance and is, on average, 12% better than OJ in Ext4. We note that SOJ can violate data durability and/or consistency requirement under power failure or system crash. On the other hand, SDJ shows similar performance to that of OJ mode at `fsync()` interval 128 while it guarantees the version consistency. Unfortunately, the performance of SDJ degrades as the `fsync()` interval is shortened. The major reason for the performance drop is that SDJ has to trigger checkpoint operations more frequently than OJ mode in order to reclaim journal blocks; the journal area will fill up more quickly in SDJ as both data and metadata have to be stored in the journal area in SDJ mode. Fortunately, this performance issue, though orthogonal to the main theme of this paper, can be mitigated simply by increasing the default journal size [6].

To further understand the benefits of SDJ, we used the Varmail in Filebench that is one of the representative real-world benchmarks and configured it to generate multi-threaded I/O workload with 10,000 files and 100 concurrent threads. Figure 2b demonstrates the throughput of Varmail. From this figure, we can confirm that SDJ works well in real-world workload and the results have similar patterns to those in Figure 2a.

Case Study 2: F2FS Now, we explore another use case of *SHARE* in log-structured file system (LFS) [23]. A log-structured writing scheme is widely adopted for flash storage devices, but it still suffers from the inevitable segment cleaning overhead for securing large chunks of free space. Though several techniques, such as data grouping [16], slack space recycling [18], and in-place-update (IPU) mode in F2FS [2], have been proposed, none of them completely remove copy-back overhead of valid blocks. In contrast, our approach can fundamentally remove the copy-back overhead of valid blocks by incorporating *SHARE* into the segment cleaning procedure. That is, instead of copying valid blocks in a victim segment to a new segment, we simply call *SHARE* for address remapping from the victim segment to the new segment.

We implemented this *SHARE*-aware segment cleaning (SSC) by modifying about 100 LoC of F2FS. For evaluation, we first filled up the file system to make its utilization to 50% of the total space. Then, we performed experiments with FIO benchmark, which was configured to perform random writes to 40% of the total storage capacity, by varying `fsync()` interval from 8 to 128. Figure 3 shows the total number of pages moved during the segment cleaning and the performance results (IOPS) of each segment cleaning. Figure 3a shows how many valid pages are moved during the segment cleaning and we can see that there is no significant difference between SC and SSC. Interestingly, when `fsync()` interval is 8, SSC and



(a) Performance **(b)** Write amount **(c)** Disk cache flush count

Figure 4: OLTP benchmark results of Sysbench and LinkBench using MySQL. SysBench in an OLTP mode handles a 10 GB database (20 files) with 40 million rows for 1,000,000 operations. We ran LinkBench with 4,800,000 operations for a 50 GB database (24 files) after two minutes warm-up. In both experiments, MySQL/InnoDB engine was configured to use 5 GB as a buffer pool with sixteen concurrent threads, and all under buffered I/O mode.

SC do nothing. This is because current F2FS was modified to allow an in-place update when the `fsync()` interval is smaller than 16. Meanwhile, Figure 3b demonstrates that SSC outperforms the original segment cleaning by 10%–39% although the number of copy-backed pages is similar. This performance improvement can be explained with the fact that SSC updates only metadata blocks, such as segment information table and segment summary area, while avoiding the copy-back overhead of data blocks.

In summary, the above studies show that the simple *SHARE*-based address remapping functionality of flash storage helps the existing file systems to remove the overhead of consistency mechanisms and thus to boost their performance significantly.

4 Application-level Crash Consistency

For some applications such as databases and key-value stores, even the system-wide version consistency of Ext4 DJ mode, despite its double-write journaling, fails to meet their stringent requirements for transactional atomicity. For this reason, each application should devise its own *application-level crash consistency* mechanism. For instance, it is well-known that MySQL relies on costly application-level journaling mechanism for transactional atomicity, called *double-write-buffer* (for short, DWB) [3]. However, such application-level crash consistency mechanisms bring two problems. First, they usually suffer from poor performance and reduced lifespan of the underlying flash storage because of write amplification and frequent `fsync()` calls from the application layer. Second, the application-level update protocols are so complex and error-prone, and there still exist some subtle bugs even in widely-deployed applications [20,21,27]. Therefore, it would be desirable for file system to support the application-level crash consistency as its first class citizen functionality.

Case Study 3: MySQL/InnoDB We show how file systems can utilize the *SHARE* in supporting application-level crash consistency in database applications. One main challenge in utilizing *SHARE* for this purpose is how to ensure the ACID semantics of conventional database transactions on top of transactional file systems. Because the set of pages that has to be atomically written by a database transaction may span over multiple transactions of file system, file system may break the atomicity of the database application. For example, a problematic case occurs when one application's transaction is chopped by the transactions of the file system due to `fsync()` or time threshold (*i.e.*, 5 seconds commit) in an unintended fashion. To prevent such a case, we applied the semantics of the failure-atomic update APIs (*i.e.*, `O_ATOMIC`, `syncv()`, and `msync()`) [19, 25] to our prototype (*i.e.*, SDJ), and we call this as SHARE-*aware application-level data journaling (SADJ)*. In SADJ, these APIs guarantee the atomic write of multiple scattered pages in either single or multiple files opened with `O_ATOMIC` flag. For files opened with `O_ATOMIC`, SADJ defers writing of dirty blocks to their home location, and these dirty blocks will be written to their home location only by synchronization operations (*e.g.*, `fsync()`, `fdatasync()`, `msync()`, and `syncv()`). More interestingly, SADJ can halve the amount of writes to the storage by issuing *SHARE* instead of the redundant writes, hence doubling the application's performance.

Unfortunately, as mentioned in §3, SADJ sometimes faces a challenge due to the small-size journal area (*e.g.*, 128MB) because it quickly fills up the journal area by allocating journal blocks for application's data as well as file system's metadata. In addition, in terms of application-level crash consistency, some applications may need large journal area to support the DBMS-like ACID transactions. In order to address this challenge caused by small-size journal area, we decided to allocate rather large-size journal area (*e.g.*, 1GB), which we believe is large enough to preserve all the data that are required for application-level crash consistency. Of course, one alternative solution to completely address the issue is to dynamically extend the journal area, but it is beyond the scope of our study. For this study, MySQL/Inn-

oDB storage engine was used. Because SADJ can guarantee the *version consistency*, the database is safe from the risk of data inconsistency despite the DWB mode in the storage engine is turned off. To evaluate the effect of SADJ on MySQL/InnoDB database, we ran two popular OLTP benchmarks, SysBench [5] and LinkBench [7], under four different modes: (1) `DWB-ON/OJ`(default), (2) `DWB-OFF/DJ`, (3) `DWB-OFF/OJ`, and (4) `DWB-OFF/SADJ`. The results are presented in Figure 4. Note that the third mode `DWB-OFF/OJ` does not prevent data corruption while the other three modes do; we deliberately added the *crash-inconsistent* `DWB-OFF/OJ` mode so as to stress that `DWB-OFF/SADJ` can outperform even the crash-inconsistent mode. As Figure 4a shows, `DWB-OFF/SADJ` outperforms the default mode `DWB-ON/OJ` by 6.16 times and the `DWB-OFF/DJ` by 2.73 times. This performance gain is, as is clearly shown in Figure 4b, in part due to the reduction of the amount of writes which comes from avoiding the redundant writes at either `DWB` or `DJ` mode and instead calling the address remapping at the storage layer. However, this reduction cannot solely explain the large performance gap between `DWB-ON/OJ` and `DWB-OFF/SADJ` modes. The other main reason for the gap is the difference in the number of disk cache `FLUSH` operations invoked in the two modes. The default `DWB-ON/OJ` mode frequently calls `fsync()` to guarantee the consistency of database files[1]. On the other hand, `DWB-OFF/SADJ` calls one `FLUSH` operation after writing all database files together because SADJ can provide application-level data ordering and durability for free. Thus, as Figure 4c shows, `DWB-OFF/SADJ` invokes 16.4x less disk cache `FLUSH` operations than the original version.

In summary, applications can benefit in terms of performance without sacrificing any consistency if file systems can explicitly and efficiently support *application-level crash consistency* on top of flash storage devices with the atomic address remapping functionality.

5 Related Work

Our study is not the first work on exploiting the address remapping mechanism for system performance optimization [6, 11, 12, 17, 26]. Now, we will compare the closest previous studies, ANViL [26] and JFTL [9], with our study. Weiss *et al.* [26] proposed a small set of storage APIs, based on address remapping at the virtualized block device layer, and showed that those primitives are useful in a variety of case studies, such as single-write journaling, snapshot, file copy, and de-duplication. In this respect, our study could not be regarded as unique. But, we argue that our study is in stark contrast with ANViL from two perspectives. First, we show for the first time that the application-level crash consistent file system can be made practically with the help of atomic address remapping in flash storages. Second, from the perspective of performance, we think that the right place to embody the functionality of atomic address remapping is not in the host-side block device layer, but the FTL layer as in *SHARE*; because the space for address mapping in ANViL is managed in a log-structured manner, another garbage collection scheme is required.

To our knowledge, JFTL [9] is the first approach to suggest the atomic address remapping functionality in FTL so as to avoid the redundant write overhead in journaling file system. In this sense, it is the closest approach to our study. But, the authors of JFTL did not take into account the benefits and challenges of the application-level crash consistency at all. In addition, since it assumes a proprietary interface between the host and flash storage which is introduced only for the purpose of journal data remapping, its atomic address remapping, unlike ANViL [26] and *SHARE*, could not be fully and flexibly leveraged by host systems.

6 Conclusion

In this paper, we presented a comprehensive study where address remapping technique can be used to relieve the file system's burden of guaranteeing the crash consistency as well as data consistency, and boost the performance of the consistency-critical applications. In our study, to show the practical portability of the address remapping, we implemented its functionality inside a commercial SSD as firmware. Our experimental results show that *SHARE*-based file systems perform similar to or outperform conventional ones, while providing the higher-level version consistency.

Meanwhile, the idea of leveraging the address remapping in file systems is not limited to journaling file systems and log-structured file systems. We expect that the *SHARE* would be also helpful in mitigating the tree-wandering problem in CoW-based B-tree file systems, such as btrfs [22]. This would be an interesting topic for future work.

7 Acknowledgments

This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT (No. NRF-2015M3C4A7065696). This research was supported in part by IITP under the "SW Starlab" (IITP-2015-0-00314) and in part by Samsung Electronics.

[1] When a dirty page is replaced from the buffer cache, DWB appends new copy to the double-write-buffer and then overwrites the old copy in its original location. In each step, an `fsync()` call is made to enforce ordering and durability.

References

[1] Ext4 Disk Layout/Journal (jbd2). https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout#Journal_.28jbd2.29.

[2] f2fs: introduce flash-friendly file system. https://lwn.net/Articles/518718/.

[3] MySQL 5.7 Reference Manual. http://dev.mysql.com/doc/refman/5.7/en/.

[4] SQLite. http://www.sqlite.org/.

[5] SysBench (Branch 1.0). https://github.com/akopytov/sysbench.

[6] Aghayev, A., Ts'o, T., Gibson, G., and Desnoyers, P. Evolving Ext4 for Shingled Disks. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies* (2017), FAST'17, USENIX Association, pp. 105–119.

[7] Armstrong, T. G., Ponnekanti, V., Borthakur, D., and Callaghan, M. LinkBench: a Database Benchmark based on the Facebook Social Graph. In *Proceedings of the 39th SIGMOD International Conference on Management of Data* (2013), SIGMOD'13, ACM, pp. 1185–1196.

[8] Chidambaram, V., Sharma, T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Consistency Without Ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (2012), FAST'12, USENIX Association, pp. 1–16.

[9] Choi, H. J., Lim, S.-H., and Park, K. H. JFTL: A Flash Translation Layer Based on a Journal Remapping for Flash Memory. *Transactions on Storage 4*, 14 (Feb. 2009), 1–22.

[10] Gupta, A., Kim, Y., and Urgaonkar, B. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (2009), ASPLOS'09, ACM, pp. 229–240.

[11] Hahn, S. S., Lee, S., Ji, C., Chang, L.-P., Yee, I., Shi, L., Xue, C. J., and Kim, J. Improving File System Performance of Mobile Storage Systems Using a Decoupled Defragmenter. In *Proceedings of the 2017 USENIX Annual Technical Conference* (2017), ATC'17, USENIX Association, pp. 759–771.

[12] Kim, H., Shin, D., Jeong, Y. H., and Kim, K. H. SHRD: Improving Spatial Locality in Flash Storage Accesses by Sequentializing in Host and Randomizing in Device. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies* (2017), FAST'17, USENIX Association, pp. 271–283.

[13] Lee, C., Sim, D., Hwang, J.-Y., and Cho, S. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (2015), FAST'15, USENIX Association, pp. 273–286.

[14] Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A., and Vivier, L. The New Ext4 Filesystem: Current Status and Future Plans. In *Proceedings of the Ottawa Linux Symposium* (2007), OLS'07, pp. 21–33.

[15] Min, C., Kang, W.-H., Kim, T., Lee, S.-W., and Eom, Y. I. Lightweight Application-Level Crash Consistency on Transactional Flash Storage. In *Proceedings of the 2015 USENIX Annual Technical Conference* (2015), ATC'13, USENIX Association, pp. 221–234.

[16] Min, C., Kim, K., Cho, H., Lee, S.-W., and Eom, Y. I. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (2012), FAST'12, USENIX Association, pp. 1–16.

[17] Oh, G., Seo, C., Mayuram, R., Kee, Y.-S., and Lee, S.-W. SHARE Interface in Flash Storage for Relational and NoSQL Databases. In *Proceedings of the 2016 International Conference on Management of Data* (2016), SIGMOD '16, ACM, pp. 343–354.

[18] Oh, Y., Kim, E., Choi, J., Lee, D., and Noh, S. H. Optimizations of LFS with Slack Space Recycling and Lazy Indirect Block Update. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference* (2010), SYSTOR'10, ACM, pp. 1–9.

[19] Park, S., Kelly, T., and Shen, K. Failure-atomic Msync(): A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), EuroSys '13, ACM, pp. 225–238.

[20] Pillai, T. S., Chidambaram, V., Alagappan, R., Al-Kiswany, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation* (2014), OSDI'14, USENIX Association, pp. 433–448.

[21] Pillai, T. S., Chidambaram, V., Alagappan, R., Al-Kiswany, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Crash Consistency. *Communications of the ACM 58*, 10 (Oct. 2015), 46–51.

[22] Rodeh, O., Bacik, J., and Mason, C. BTRFS: The Linux B-Tree Filesystem. *Transactions on Storage 9*, 3 (Aug. 2013), 1–32.

[23] Rosenblum, M., and Ousterhout, J. K. The Design and Implementation of a Log-structured File System. *ACM Transactions on Computer Systems 10*, 1 (Feb. 1992), 26–52.

[24] Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., and Peck, G. Scalability in the XFS File System. In *Proceedings of the 1996 USENIX Annual Technical Conference* (1996), ATC'96, USENIX Association, pp. 1–14.

[25] Verma, R., Mendez, A. A., Park, S., Mannarswamy, S., Kelly, T., and Morrey, C. B. Failure-atomic Updates of Application Data in a Linux File System. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (2015), FAST'15, USENIX Association, pp. 203–211.

[26] Weiss, Z., Subramanian, S., Sundararaman, S., Talagala, N., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. ANViL: Advanced Virtualization for Modern Non-volatile Memory Devices. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (2015), FAST'15, USENIX Association, pp. 111–118.

[27] Zheng, M., Tucek, J., Huang, D., Qin, F., Lillibridge, M., Yang, E. S., Zhao, B. W., and Singh, S. Torturing Databases for Fun and Profit. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation* (2014), OSDI'14, pp. 449–464.