OFTL: Ordering-aware FTL for Maximizing Performance of the Journaling File System

Daekyu Park †‡, Dong Hyun Kang†, and Young Ik Eom†
†Sungkyunkwan University, Korea ‡Samsung Electronics, Korea
{dekay.park,kkangsu,yieom}@skku.edu

ABSTRACT

Journaling of ext4 file system employs two FLUSH commands to make their data durable, even though the FLUSH is more expensive than the ordinary write operations. In this paper, to halve the number of FLUSH commands, we propose an efficient FTL, called OFTL, that completely ensures the write order of the journal data blocks in storage-level. For performance comparison, we implemented our OFTL on Jasmine OpenSSD and measured its performance with three different workloads. Our experimental results show that OFTL outperforms the up-to-date FTLs on the existing journaling modes by up to 1.97 times.

1 INTRODUCTION

For the past few decades, the FLUSH command has been used to guarantee data durability in storage devices (e.g., HDD, SSD, and eMMC) and it is one of the most common commands in storage interfaces (e.g., SATA, NVMe, and SCSI). In particular, most file systems issue the FLUSH command to the storage device at the point where data consistency should be ensured. For example, the ext4 file system guarantees the data consistency of the file system by adopting journaling mechanisms based on the FLUSH command. To guarantee the consistency of data stored on the file system even when the system crash occurs, the journaling mechanism ordinarily writes the updated metadata to the journal area before writing the corresponding data to the data blocks, in order to safely recover the file system on the system crash or failure. Unfortunately, the journaling mechanism suffers from substantial performance degradation induced by FLUSH commands, because of its heavy behaviors inside the storage device (flushing buffers and storing metadata). Nevertheless, the mechanism always issues the FLUSH command twice to ensure the transaction consistency; (1) the first FLUSH command is issued before saving the commit block, to persistently store journal data blocks, and (2) the second FLUSH command is issued after saving the commit block, to ensure the current transaction is persistent.

To relieve the overhead of the FLUSH command, many researchers focused on the journaling mechanism and proposed several alternative approaches for omitting the FLUSH command in the kernel-level. For example, the IRON file system employed a checksum mechanism instead of the first FLUSH command [1]. Also, some researchers introduced the coerced cache eviction (CCE) [2] or zero padding scheme [3], where both schemes efficiently replace the FLUSH commands with EVICTION operations for performance improvement. Other researchers revisited the synchronization mechanisms (e.g., *fsync*, *sync*, *msync*) of the traditional file system and redesigned the file system to improve its efficiency [4, 14].

In this paper, we focus on the storage device. In particular, we study the internal structure and mechanisms inside the SSD device to answer the following question: *Why are the FLUSH commands used to ensure the order of writes, even though journal data blocks and the commit block are issued sequentially?* We then propose a novel flash translation layer (FTL) scheme, called ordering-aware FTL (OFTL), that supports the order of incoming data by handling *dependency* among the write requests. Therefore, OFTL can help to improve the performance of journaling by removing the first FLUSH command, while guaranteeing the order of writes. To the best of our knowledge, this work is the first attempt to remove the FLUSH command by supporting a new FTL scheme in the storage-level. We implemented OFTL on Jasmine OpenSSD board [5] and modified the existing journaling code (i.e., jbd2) in the Linux kernel to ensure that the journaling mechanism can omit the first FLUSH command during the process of journaling. Our evaluation results clearly confirm that OFTL fully supports the journaling mechanism. In particular, the modified jbd2 on OFTL improves the overall performance by up to 1.97 times compared with the ordered mode journaling.

In the rest of this paper, Section 2 presents the background and design motivation. Then, Section 3 describes the OFTL in detail. Next, we present our evaluation results in Section 4 and compare them with other solutions in Section 5. Finally, Section 6 covers the related work and Section 7 concludes this work.

2 BACKGROUND AND MOTIVATION

2.1 Journaling Mechanism

Journaling is one of the crash consistency mechanisms and it is widely used in various file systems (e.g., ext3, ext4, and XFS) to prevent data loss when a system crash occurs. In this section, we

DAC '18, June 24–29, 2018, San Francisco, CA, USA

ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3196082

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) and this research was supported by the MSIT(Ministry of Science and ICT), Korea, under the SW Starlab support program(IITP-2015-0-00284) supervised by the IITP(Institute for Information & communications Technology Promotion)

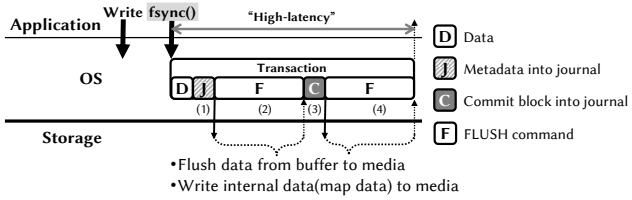

Figure 1: Journaling procedure (ordered mode)

describe in detail how ext4 guarantees data consistency with ordered mode journaling. In ext4, file system consistency is guaranteed by writing modified data twice: writing to the journal area and writing to the original area. However, the write for journaling degrades the overall performance of the file system because it has an ordering constraint as shown in Fig. 1: (1) it first scans the page cache layer to find modified data and writes the metadata of the modified file to the journal area (i.e., *journal data block*); (2) then, it issues FLUSH command to the storage device; (3) subsequently, it writes the commit block to the journal area for marking the transaction as committed; and (4) finally, it issues the FLUSH command again to the storage device. In this sequence of journaling, the first FLUSH command is used to ensure the correct order of writes between the journal data blocks and the commit block (i.e., *FLUSH for ordered writes*). This ordering is very important because the commit block indicates that all journal data for the current transaction are valid. In other words, all journal data blocks should be stored in the journal area before storing the commit block for successful file system recovery. The second FLUSH command completes the current transaction by reflecting the commit block immediately to the non-volatile media (i.e., *FLUSH for data durability*). Unfortunately, the FLUSH command significantly harms the storage performance, and it has been a well-known performance issue in the research area of file systems.

2.2 Need for FLUSH Command

The FLUSH command is used in the file systems because of the internal structure and mechanisms of the storage devices; the buffering scheme inside the storage and parallelism of multiple flash chips.

Buffering mechanism. The major reason for using FLUSH command is because of the buffering mechanism inside the storage device. As we know, most storage devices use DRAM-based write buffer to enhance storage performance [16]. In order to improve the performance, the write buffer temporarily holds incoming data until some of the requirements are fulfilled (e.g., reclaim of the buffer space). Unfortunately, this buffering causes unintended delay on the writing of the data blocks in the storage media. In particular, since these mechanisms strongly depend on the policy of the storage manufacturers, the file system cannot predict when the data blocks will be reflected on the persistent storage media (i.e., NAND flash memory for SSD). Therefore, file systems should issue the FLUSH command when the durability of the data issued for write needs to be guaranteed. This is because the FLUSH command makes all the data in the write buffer be written explicitly to the persistent storage media.

Parallelism. In the case of the SSD device, when the data is transferred to the storage, the prediction of the point-in-time for

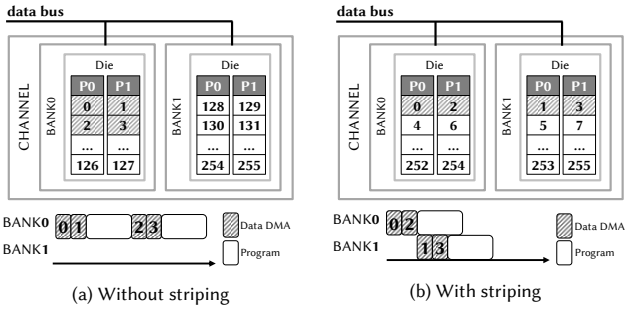

Figure 2: Striping policy in the NAND flash memory

the data to be written to the flash memory is more difficult and complex even without considering the DRAM buffer. This is due to the *parallelism* inside the SSD device (e.g., channel, bank, and plane-level parallelism). As we know, the parallelism significantly improves the storage performance [15]. In addition, the flash-based storage device adopts the Logical Page Number(LPN) stripi ng policy to effectively perform the parallelism. This is because the LPN striping forces all chips to operate simultaneously in parallel by distributing LPN on the configured flash chips. Fig. 2 shows the effectiveness of the chip parallelism and an example of striping a write request in the SSD device. The device with no LPN striping (Fig. 2a) immediately places the incoming data on the single flash chip, so that the flash chip operates one at a time in a serial way. Accordingly, it does not fully utilize the high-performance multi-bank architecture of the device. On the other hand, the device with LPN striping (Fig. 2b) distributes the data in a logical page granularity and determines the location of the page based on the incoming order with the dynamic allocation method [6]. This causes the data to be evenly striped across multiple banks, so that the striping scheme helps to simultaneously write multiple pages on multi-bank channels. However, the problem with the LPN striping is that the host cannot predict the time at which its data will be written to the flash memory. This is because the system is very complex to exactly figure out the striping policy and the granularity inside the storage device. In conclusion, to ensure that the behavior of the device is deterministic in terms of data durability, the FLUSH command should be issued to the flash-based storage at the time needed.

2.3 Motivation

In the SSD devices, techniques for parallelism are critical for performance improvement. However, as mentioned above, these parallelisms cause unpredictable data durability. This means that the storage device may mix the order of data to be written in the flash memory, because some write requests can be delayed by the internal components of the device: (a) when the program unit of the flash memory is not filled, the write cannot be issued to the flash chip; (b) when the flash chip is already in operation with the previous requests (e.g., program, read, and erase), the current write may be delayed. In short, the order of writes to the flash memory may not correspond to the incoming order from the host due to the chip parallelism. Therefore, as for the journaling mechanisms previously mentioned, if the file system wants to definitively guarantee the order of writes between the journal data blocks and the commit block, it should issue the

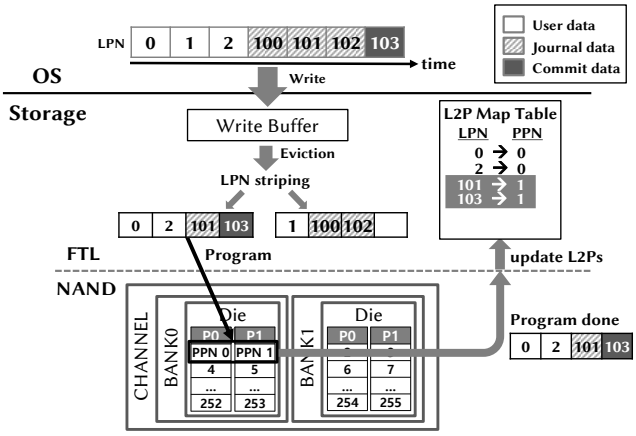

Figure 3: Inversion of the update order induced by using LPN striping policy. (The L2P map of the 103 was updated prior to 100 and 102.)

FLUSH command before writing the commit block. Otherwise, the transaction consistency may not be guaranteed by writing the commit block prior to some of the journal data blocks. Fig. 3 shows an example of how the order of journal data blocks is scrambled on the internal SSD device in the case of no FLUSH command. In this example, we assume that the write buffer adopts the simple FIFO algorithm as its eviction policy. Therefore, the write buffer does not affect the order of writes. Also, the stripe granularity is one logical page and it determines the location of a page based on its LPN. Consequently, the data evicted from the write buffer is placed on each bank by the striping policy; LPN 0, 2, 101, and 103 are placed on the first bank and the others (LPN 1, 100, and 102) are placed on the second bank. Then, the first bank starts to write its own data because the program unit of the flash memory is fulfilled. However, the second bank cannot start to write because it should wait to be fulfilled for the write operation. As a result, the commit block (LPN 103) is written to the flash memory in advance prior to the journal data blocks (LPN 100 and 102). If a system crash occurs at this time, the file system consistency cannot be guaranteed. In response to this type of inversion in the order of writes at storage-level, we introduce an enhanced FTL scheme in this paper.

3 DESIGN AND IMPLEMENTATION

In this paper, our goal is to improve the IO performance by omitting the FLUSH command whenever possible. To meet this goal, we designed a novel Ordering-aware FTL (OFTL), that guarantees the order of writes inside the storage device. To ensure the order of writes, the OFTL serializes the order of updates on the L2P map, rather than the order of writes on the data itself in the flash memory. This is because the durability of data is not guaranteed unless the L2P map is updated. Consequently, when using a storage device with OFTL, the journaling mechanism of the file system does not need the FLUSH command that is used to guarantee the order of writes. Fig. 4 shows the architectural overview of the OFTL with the three newly designed components.

3.1 Request Classifier

The Request Classifier (RC) determines whether the order should be guaranteed by using the logical block address (LBA) of the

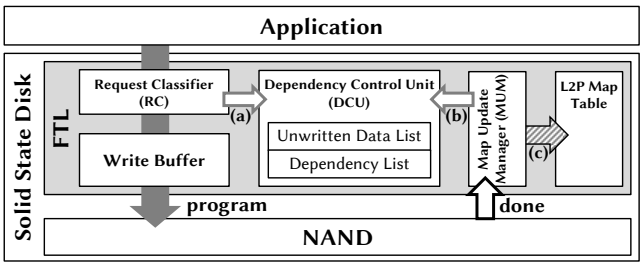

Figure 4: Overview of the architecture of OFTL

data requested to be written from the system. This procedure is necessary because guaranteeing the order of writes of all the requests is not always required. Therefore, the storage device needs an order-critical range of LBAs to classify the data types, and the range information can be received through IOCTL system call from the host. In our scheme, we defined the journal area of the Linux ext4 file system as an order-critical range. As mentioned previously, in the case of the journal area, the data order has a critical impact on the recovery of the file systems. Through the journal area information, the RC classifies the type of requested data into the journal data and normal data. The data types assorted by the RC are passed to the Dependency Control Unit (DCU) together with the LPN information.

3.2 Dependency Control Unit

The DCU performs two functions to ensure the order between the journal data and normal data by using the information received from the RC. The first function is used to configure the *map update dependency* among the data according to the predefined rules. The second function manages the *map update dependencies* in the form of lists.

Map update dependency is established only between journal data and normal data, and employs two lists: *unwritten data list* and *dependency list*. This dependency implies that the L2P map of the journal data should be updated after the L2P map of normal data is updated. The DCU configures the dependencies using the following steps. (a) When normal data is requested from the host, the DCU registers the information of the normal data to the *unwritten data list* until the next journal data is requested. (b) Subsequently, when the journal data comes in, the DCU constructs the dependency between the journal data and the normal data previously registered on the *unwritten data list*. (c) If no unwritten normal data are present when the journal data is requested, the DCU forms the dependency of the journal data with no normal data (we call this dependency an *empty dependency*). (d) The DCU sequentially transforms the dependency constructed by the previous rules into the *dependency list*. In the transition, the sequential order is very important because it means that the L2P map of the journal data is updated in sequence. Therefore, the DCU ensures that the L2P map of the journal data is updated in the order of the system requests. Further details are described in the next section.

3.3 Map Update Manager

The existing FTL updates the L2P map immediately when the data is programmed into the flash memory, while OFTL determines whether or not to immediately update the L2P map de-

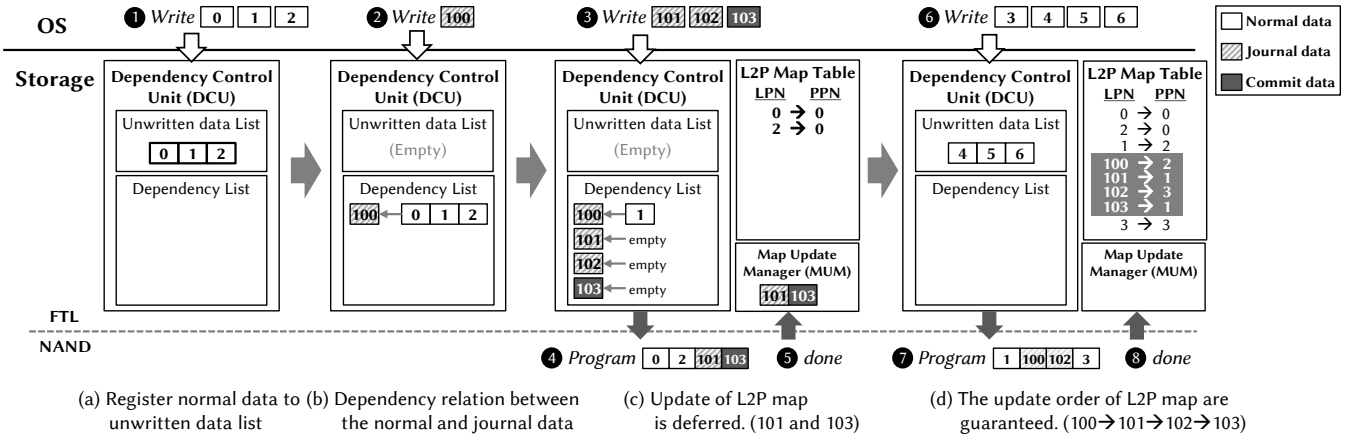

Figure 5: Operation flow of OFTL. When the storage device receives the normal and journal data, the behaviors of the OFTL follow the four steps shown in the figure.

pending on the data type, using the map update manager (MUM). In case of normal data, the L2P map is immediately updated when the data is programmed into the flash memory. The MUM then requests the DCU to release the dependency of the normal data. Therefore, the L2P map of the journal data related to some of the normal data can be updated later. On the one hand, if the normal data has been kept in the *unwritten data list* without any dependency, the DCU removes the information of normal data from the list. On the other hand, in case of journal data, the L2P map can be updated only when the following two conditions are satisfied. First, all dependencies of the connected normal data should be released. This means that the L2P map of all related normal data has been updated. Second, in the *dependency list* of DCU, all previously registered dependencies should be released. This means that the order registered to the *dependency list* is the same as the order of map updates among the journal data. If any of the above two conditions are not met, the update of the L2P map is delayed. The deferred L2P map of journal data is updated when the above two conditions are satisfied.

Fig. 5 shows an example of the operation flow when ensuring the order of writes by OFTL, in the order which is requested. (a) Writes are requested from the host in the order of LPN 0, 1, and 2. The LPNs of these requests are not included in the order-critical range and are therefore registered in the *unwritten data list* of the DCU. (b) Next, another write is requested to the LPN 100, which belongs to the order-critical range. Therefore, the journal data is registered in the *dependency list* by forming a dependency with the unwritten normal data. (c) Then, the writes to LPN 101, 102, and 103 are requested when the *unwritten data list* is empty. This time, the requested journal data forms an *empty dependency* and sequentially registers it in the *dependency list*. At this point, LPN 0, 2, 101, and 103 are programmed on the same physical page by the LPN striping policy, as shown in Fig. 3. When the data is programmed on the physical page, the L2P map for the data is updated or delayed. The L2P maps of LPN 0 and 2, which are normal data, are immediately updated. Then, the dependency connected to LPN 100, the journal data that formed the dependency, is released. On the other hand, since LPN 101 is journal data, the two conditions of MUM should be satisfied to update the L2P map. The first condition is satisfied because the dependency of LPN 101 is formed as an *empty dependency*. However, since the LPN 100 registered before the LPN 101 in the *dependency list* has not yet been released, the second condition is not satisfied. Therefore, the update of the L2P map for LPN 101 is delayed. The update of the L2P map for LPN 103 is also delayed for the same reason. (d) When more writes are requested (LPN 3, 4, 5, and 6), the data of LPN 1, 100, 102, and 3 are programmed into the single physical page on the flash memory. After the program is completed, the L2P maps of LPN 1 and 100 are updated according to the above-mentioned conditions. At this time, since the previous dependencies of LPN 101 in the *dependency list* are released, it is possible to update the map of LPN 101. Subsequently, the maps of LPN 102 and 103 can also be updated.

In applying our new components (RC, DCU, and MUM), OFTL incurs two kinds of overhead: space and latency. First, our scheme uses additional memory space to maintain the dependency. However, this overhead is negligible because we only require 10 bytes for each 4K entry that reside in the write buffer (0.24% of the whole write buffer space). Second, latency overhead comes from the computation for updating the dependency. However, this overhead is for processing write operation, and so, it can be hidden behind long NAND write time. Therefore, we believe the second overhead is also negligible.

4 EVALUATION

We conducted the evaluation in Ubuntu 16.04 LTS (kernel version 4.4) and employed Jasmin OpenSSD board [5] that consists of eight NAND flash chips (2 channels and 4 banks) and has 4KB of the logical page. In order to implement OFTL, we extended a well-known page-mapping based FTL on OpenSSD [5] and modified codes in the ext4 journaling layer (i.e., jbd2) to remove the first FLUSH command. For practical comparison, we also implemented the stream FTL [17] where the stream functionality is modified to ensure that it guarantees the order of writes issued by jbd2. Especially, stream FTL is similar to OFTL in that it can omit the first FLUSH command. However, it is different in that the stream FTL should handle the writes in a serial way because it cannot ensure the order of writes across multiple-chips.

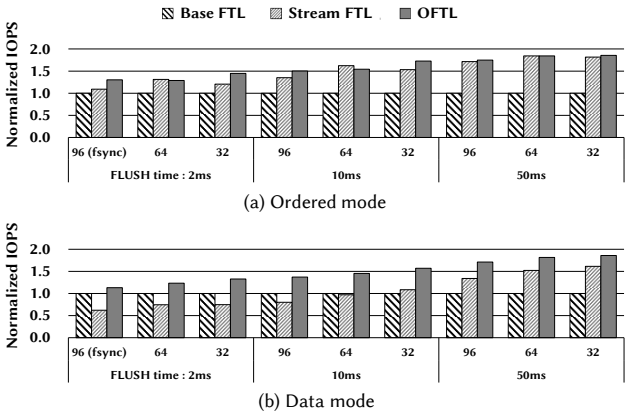

Figure 6: Performance of FIO workload configured with FLUSH time, frequency of *fsync*, and journaling modes.

Since the latency for processing each FLUSH command inside the storage device is correlated with the overall performance, we repeatedly evaluated OFTL and compared it with the other two FTLs by varying the processing time: 2ms, 10ms, and 50ms. We decided these processing times based on those of the commercial SSDs (from a few milliseconds to tens of milliseconds).

Synthetic workload. We first evaluated OFTL with FIO (Flexible I/O) microbenchmark that generates synthetic workloads along with *libaio* engine and the I/O queue depth of 32. For evaluation, we ran FIO to generate a sequential write of a 30GB file at a granularity of 4KB. To further stress the file system and the underlying storage device, we called an *fsync* operation with different frequencies: 32, 64, and 96. In this experiment, when the frequency is set to 32, FIO calls *fsync* operation after every 32 consecutive writes.

Fig. 6 shows the performance of the synthetic workload of all configurations, and the results are normalized to those of the page-mapping based FTL (we call this the base FTL). As shown in the results, the performance of the OFTL is substantially superior to that of the base FTL in all cases. Especially, OFTL outperforms the base FTL up to 1.8x on both journaling modes in case of the frequency of 32 and FLUSH time of 50ms. As expected, as the FLUSH time is decreased, the performance gap between OFTL and the based FTL decreases. This is because the FLUSH time becomes a less dominant factor influencing the performance. On the other hand, although the stream FTL removes the first FLUSH command similar to OFTL, the results of the stream FTL show worse performance than those of OFTL. Furthermore, in the case of a short FLUSH time on the data mode (Fig. 6b), the results of the stream FTL reveal lower performance compared to those of the base FTL. This performance drop occurs because the stream FTL writes the data issued by jbd2 in a serial way to guarantee the order of writes; it reduces the chip parallelism. In contrast, OFTL fully utilizes the parallelism through *dependency relation* that helps to stripe LPNs across multiple chips.

Real-world workload. To further confirm the effectiveness of OFTL, we used two real-world workloads: MySQL of Sysbench [8] and Varmail of Filebench [7].

Sysbench emulates the OLTP database workload based on MySQL and it frequently calls *fsync* operation to guarantee its database consistency. In this experiment, a total of 100,000 tables were created and 100,000 requests were handled by a maximum of 500 connections. Fig. 7 shows the performance results of Sysbench that are normalized to the base FTL. As shown in Fig. 7, OFTL shows better performance than the base FTL in all cases. Such performance improvements are possible because OFTL reduces the number of FLUSH commands by omitting the first FLUSH command in every commit procedure. Meanwhile, unlike the results shown in Fig 6, the stream FTL in Fig. 7 shows comparable performance to that of OFTL. These results are acceptable because this workload incurs frequent *fsync* calls along with a small number of data writes; the small number of writes reduces the performance impact that derives from the chip parallelism.

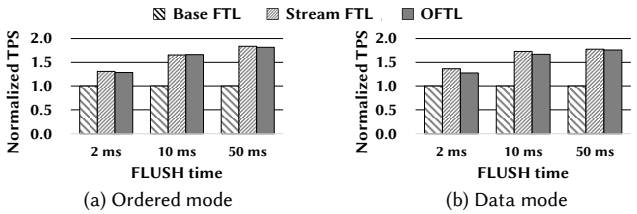

Figure 7: Normalized performance of the MySQL of Sysbench while adopting various FLUSH times.

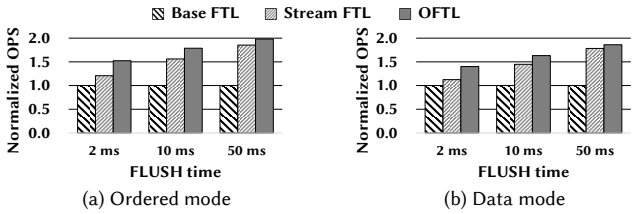

Figure 8: Normalized performance of the Varmail of Filebench while adopting various FLUSH times.

The Varmail generates a workload for a multi-thread mail server and performs file operations, such as *create*, *delete*, *read*, *append*, and *fsync*. The workload is configured by 50% write ratio and very frequent *fsync* operations. As shown in the results of the ordered mode (Fig. 8a), OFTL outperforms the base FTL by 1.97 times in the case of 50ms of FLUSH time. In addition, the trends shown in Fig. 8a are very similar to those shown in Fig. 6, but the performance gap is larger than that in Fig. 6. To illustrate this gap, we analyzed the workload inside the storage device and found that Varmail calls the FLUSH command more frequently than the synthetic workload.

5 DISCUSSION

Since OFTL efficiently supports the write order among the journal data blocks at the storage-level, the journaling can omit the FLUSH command for the ordering during the commit operation. Similarly to the OFTL, the current Linux kernel also provides a special option, called *asynchronous commit*, which replaces the FLUSH command with the calculation of CRC checksum. If this option is set with the ext4 file system, the journaling calculates CRC of all journal data blocks and inserts the CRC value into the commit block instead of issuing the FLUSH command. If a system crash occurs, the journaling first calculates the CRC value by scanning the journaled data blocks that are placed before the commit block on the journal area. If the calculated CRC value

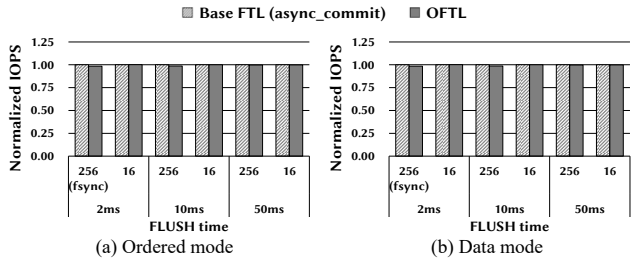

Figure 9: Comparison of the performance of *asynchronous commit* and OFTL using FIO workload

matches the CRC value stored in the commit block, the journaling recovers the journaled data blocks. Otherwise, the journaled data blocks are ignored during the recovery process. Therefore, all data on the journal area can be kept persistent along with the CRC, without the FLUSH command for the ordering among the journal data blocks. For performance comparison, we performed additional evaluations. Fig. 9 shows the results for FIO microbenchmark. As expected, the performance results on ext4 of which the *asynchronous commit* is set are similar to those of OFTL. This is because the *asynchronous commit* removes the FLUSH command for the ordering among the journal data blocks in the kernel-level, while OFTL omits the FLUSH command in the storage-level. Unfortunately, the *asynchronous commit* cannot be used for the system that requires a robust file system because it may cause data corruption because of the CRC collision. On the other hand, OFTL fully supports crash consistency without any corruption, by managing the *map update dependency*. Therefore, we believe that our OFTL is a realistic way to remove the FLUSH command without any side effects.

6 RELATED WORK

Many studies have been performed to relieve the overhead of journaling in the file system. Some researchers studied the relationship between the application and the journaling in terms of redundant data copy. They proposed novel file systems [4, 9, 10] and system calls [11, 12]. Some other researchers proposed JFTL that mitigates the journaling overhead at the storage-level [13]. The JFTL is designed to remove the redundant writes in journaling by remapping the address of the journal data. However, to adopt these previous approaches, we need to modify a large number of lines of code both in the kernel and application layer. Meanwhile, other researchers focused on the FLUSH command because it is very expensive compared to normal writes. For example, CCE and ext4-ZP introduce the scheme of padding data blocks, which triggers EVICTION operation inside the storage instead of the FLUSH command [2, 3]. Unfortunately, in case of the SSD devices, the data padding negatively affects the storage system in terms of endurance. Most similar previous work to OFTL is IRON file system. In order to remove the FLUSH overhead, the IRON file system replaced the FLUSH command for write ordering with the calculation of CRC checksum (i.e., *asynchronous commit*). As mentioned previously, this approach may cause data corruption due to the CRC collision.

Some standards on the storage interface define the auxiliary features and commands to guarantee the data ordering: tagged command queuing (TCQ) for SCSI [18] and barrier command for eMMC [19]. The ordered mode of TCQ ensures that incoming commands are fetched in order from the queue inside the storage device. However, this mode does not concern the order of writes on the storage media. Meanwhile, the barrier command is used to ensure that the order of transferred data before and after issuing this command is correctly enforced on the storage media. However, the barrier command should always be issued additionally for each request in order to support the same consistency semantics as OFTL.

7 CONCLUSION

In this paper, we first studied the relationship between the journaling mechanism and the underlying SSD device, and described the necessity of the FLUSH command for data durability. Then, we proposed a novel FTL, called OFTL, which can improve the performance of journaling by ensuring the write order of journal data in storage-level. We also showed the performance benefit of OFTL with three different benchmarks. In the best case, OFTL improves the performance by up to 1.97 times compared with the default ordered mode.

REFERENCES

[1] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-dusseau, and R. H. Arpaci-dusseau, "IRON File Systems," in Proc. of the ACM SOSP, pp. 206-220, 2005.

[2] A. Rajimwale, V. Prabhakaran, D. Ramamurthi, A. C. Arpaci-dusseau, and R. H. Arpaci-dusseau, "Coerced Cache Eviction and Discreet Mode Journaling: Dealing with Misbehaving Disks," in Proc. of the IEEE/IFIP DSN, pp. 518-529, 2011.

[3] D. H. Kang and Y. I. Eom, "TO FLUSH or NOT: Zero Padding in the File System with SSD Devices," in Proc. of the ACM APSys, pp. 1-9, 2017.

[4] V. Chidambaram, T. S. Pillai, A. C. Arpaci-dusseau, and R. H. Arpaci-dusseau, "Optimistic Crash Consistency," in Proc. of the ACM SOSP, pp. 228-243, 2013.

[5] Jasmine OpenSSD Platform, http://www.openssdproject.org/

[6] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, "Performance Impact and Interplay of SSD Parallelism through Advanced Commands, Allocation Strategy and Data Granularity," in Proc. of the ACM ICS, pp. 96-107, 2011.

[7] Filebench, http://filebench.sourceforge.net/wiki/index.php/Main_Page.

[8] A. Kopytov, "Sysbench: A System Performance Benchmark," URL: http://sysbench.sourceforge.net, 2004.

[9] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area Cooperative Storage with CFS," in Proc. of the ACM SOSP, pp. 202-215, 2001.

[10] T. S. Pillai, R. Alagappan, L. Lu, V. Chidambaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Application Crash Consistency and Performance with CCFS," in Proc. of the USENIX FAST, pp. 181-196, 2017.

[11] S. Park, K. Terence, and S. Kai, "Failure-atomic msync (): A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data," in Proc. of the ACM EuroSys, pp. 225-238, 2013.

[12] W. Lee, K. Lee, H. Son, W.-H. Kim, B. Nam, and Y. Won, "WALDIO: Eliminating the Filesystem Journaling in Resolving the Journaling of Journal Anomaly," in Proc. of the USENIX ATC, pp. 235-347, 2015.

[13] H. Choi, S Lim, and K. Park, "JFTL: A Flash Translation Layer Based on a Journal Remapping for Flash Memory," ACM Trans. on Storage, vol. 4, no. 14, pp. 1-22, 2009.

[14] Q. Chen, L. Liang, Y. Xia, H. Chen, and H. Kim, "Mitigating Sync Amplification for Copy-on-write Virtual Disk," in Proc. of the USENIX FAST, pp. 241-247, 2016.

[15] J. Kang, J. Kim, C. Park, H. Park, and J. Lee, "A Multi-channel Architecture for High-performance NAND Flash-based Storage System," Journal of Systems Architecture, vol. 53, no. 9, pp. 644-658, 2007.

[16] R. Kardela, S. Love, and B.G. Wherry, "Caching Strategies to Improve Disk System Performance," IEEE Trans. on Computer, vol. 27, no. 3, pp. 38-46, 1994.

[17] J. U. Kang, J. Hyun, H. Maeng, and S. Cho, "The Multi-streamed Solid-state Drive," in Proc. of the USENIX HotStorage, pp. 13-13, 2014.

[18] JEDEC Standard, "Embedded Multi-Media Card Electrical Standard 5.1," 2014.

[19] ANSI, "Small Computer System Interface – 2," 1994.