TO FLUSH or NOT: Zero Padding in the File System with SSD Devices

Dong Hyun Kang
Sungkyunkwan University
kkangsu@skku.edu

Young Ik Eom
Sungkyunkwan University
yieom@skku.edu

Abstract

Guaranteeing data persistency is extremely critical in the file system, and for data consistency, most file systems use the FLUSH command to force data to the persistent storage media, such as SSD devices. However, the FLUSH command significantly decreases the overall performance for flushing the data and the L2P mapping table stored on the *write buffer*. In this paper, we propose a novel alternative scheme, called *zero padding*, that ensures the same effect as physically issuing the FLUSH command while mitigating the performance problem of the FLUSH command. To achieve this, *zero padding* fully utilizes both the behaviors of the *write buffer* and the bandwidth of SATA 3.0 interface. We have successfully realized *zero padding* in ext4 and JBD2 with a small modification, called *ext4-zp*, and experimentally compared it with the traditional ext4 journal modes. Our experimental results show that *ext4-zp* provides the best throughput in various workloads and outperforms ext4 with ordered mode by up to 40%.

Keywords NAND flash storage device, write buffer, file system journaling

1. Introduction

Most NAND flash storage devices (*i.e.*, SSD) employ a small circular *write buffer* for buffering write operations between the host system and the underlying flash memory. The buffering enhances the performance of the storage system, even though it may lead to data loss after sudden power failure or system crash [25]. To avoid such data loss, the buffering imposes synchronization burden, called FLUSH, on the file system. For example, to persistently force data on the write buffer to the flash memory, ext4 journaling should issue a FLUSH command to the underlying storage before and after each commit() operation [12, 18]. Unfortunately, such a FLUSH command causes intolerable performance drop due to long write latency of the flash memory, which is induced by both buffered data and *logical-to-physical* (L2P) mapping table updates.

The recent trends in industry and academia are to reduce the number of FLUSH command executions, while keeping data persistent. For example, emerging SSD devices for data center are equipped with dedicated capacitors to preserve data on the write buffer upon unexpected power loss [3, 6, 14, 20], and therefore, it can defer flushing the buffered data and L2P mapping table for as long as possible by absorbing the FLUSH commands in the storage level. Another example is software approaches. Many researchers made diverse efforts to mitigate the large overhead of the FLUSH command in various software layers, such as application layer [15, 24, 26] and file system layer [16, 27, 30–34]. However, they had to modify a lot of lines of code to reduce the number of FLUSH command executions while guaranteeing data persistency. Some researchers focused on an EVICTION operation of the write buffer inside the storage, but they only focused on the hard disk drive (HDD) [32]. There were no considerations on the characteristics of the flash storage, such as no in-place update and limited lifetime.

In this paper, we propose new scheme called *zero padding* that can be used to replace FLUSH commands which are generally being used in the file system with SSD devices. The key idea of our scheme is to trigger the EVICTION operation for deferring L2P table updates as in the above hardware approaches while guaranteeing the same effect as physically issuing the FLUSH command. We make the following four contributions:

- We first revisit the behaviors inside the flash storage, which is performed to maintain data and L2P table persistently.
- We introduce a methodology that can extract some internal parameters and identify the *eviction spot* to trigger EVICTION operation from the *write buffer* of the storage.
- We investigate the impact of the EVICTION operation and compare it with that of FLUSH command in terms of the performance and persistency.

APSys 2017, September 2–3, 2017, Mumbai, India.

ISBN 978-1-4503-5197-3/17/09…$15.00

DOI: https://doi.org/10.1145/3124680.3124718

Type	Trigger	Data flush	L2P flush
FLUSH	Host	All data	Yes
FUA	Host	Some data	Yes
GC	Storage	All data	Yes
EVICTION	Host	Some data	No

Table 1: Comparison of the types of data persistency operations.

- We propose *ext4-zp* whose most FLUSH commands related to the journaling were replaced with *zero padding*, and implement it for evaluation.

To the best of our knowledge, this work is the first attempt to exploit a padding scheme to leverage the data persistency based on the flash storage. In the rest of this paper, we will discuss the background (§2) and design motivation (§3). Then, we will describe the details of *ext4-zp* (§4). Next, we present our evaluation results (§5). We also discuss *ext4-zp*'s limitations and potential future directions (§6). Finally, we cover related work (§7) and conclude this paper (§8).

2. Background

In this section, we describe the characteristics of the flash storage and its internal behaviors in terms of the data persistency.

2.1 Active Block and Recovery

Since NAND flash memory never allows in-place update, all flash storage devices employ a special software, called flash translation layer (FTL), which helps to transfer logical page number (LPN) received from the host to physical page number (PPN) with the logical-to-physical (L2P) mapping table. To enhance the resource utilization inside the flash storage (*e.g.*, garbage collection, internal parallelism, and wear-leveling), a recent FTL mainly adopts the page-level mapping; an incoming LPN along with data can be mapped to any PPN on the flash memory.

The page-level FTL first assigns one flash block as an *active block* and then sequentially stores incoming data within the *active block* by the order of writes [11, 13, 23]. The L2P mapping information belonging to the *active block* is not copied from DRAM inside the storage to the underlying flash memory until the block is filled up along with data. Thus, the updates for the L2P table can be delayed for a short period of time; whenever the *active block* is changed to another flash block, the L2P table on DRAM is persistently stored on the flash memory [23]. Despite such a delay operation, the FTL can guarantee the persistency of the L2P table because it always records *hints* for L2P information in the out-of-bound (OOB) area of each flash page where data is persistently stored [11, 17, 23]. In this paper, we call the hints *OOB-hints*. During this delay time, if system crash or power failure occurs, the storage would perform the power-off recovery (POR) operation that only scans the *active block* to recover the L2P mapping table based on the *OOB-hints* [11, 23]. Therefore, in the case of a recent FTL, the scanning cost for the recovery is inexpensive upon system crash or power failure.

Type	Average	Minimum	Maximum
BR	0.029ms	0.027ms	0.057ms
FR	0.227ms	0.107ms	3.446ms

Table 2: Comparison of read latencies for write buffer (BR) and flash memory (FR).

2.2 Data Persistency in the Flash Storage

In the flash storage, it is obvious that write buffer is the necessary and indispensable part to hide the write latency of NAND flash memory on-the-fly [1]. However, the write buffer is one of the most unreliable parts inside the flash storage because it is volatile cache. In this paper, we briefly categorize the mechanism that data is persistently stored inside the flash storage into four types: FLUSH, force unit access (FUA), garbage collection (GC), and EVICTION.

As shown in Table 1, the behaviors of the FLUSH and FUA command are very similar except for the amount of data flushed to the flash memory; the FLUSH command forces the storage to flush the whole buffered data and its L2P mapping table in a sequential way, whereas the FUA command only flushes some buffered data along with the corresponding L2P mapping table entries [1, 5]. GC also reflects modified data and the L2P table to their original locations in flash memory, like the two commands above, while it is periodically triggered by internal mechanisms of the flash storage (*e.g.*, idle time or no free space). On the other hand, the EVICTION is a unique operation in that it does not update the L2P table onto the flash memory immediately even though it provides the same data persistency as the other three types. As mentioned before, it is possible to guarantee the persistency of data evicted from the *write buffer* because the storage always records *hints* for L2P information in the out-of-bound (OOB) area of each flash page where evicted data is stored [11, 17, 23].

3. Motivation

3.1 Eviction Spots

To reduce write latency, most storage devices temporarily hold data written from the host in the write buffer until one of the predefined conditions (*e.g.*, active block change, FLUSH, FUA, EVICTION, etc.) is met. Unfortunately, storage manufacturers (*e.g.*, Samsung, Intel, and Micron) are reluctant to expose the details related to these conditions. To extract the *eviction spot*, which triggers the EVICTION operation from the write buffer, on a commercial flash storage, we have implemented two microbenchmark tools.

Buffer Read vs. Flash Read. The first benchmark measures the read latency of the write buffer and the flash memory. To measure the read latency of the write buffer (BR), it first writes 4KB data with O_DIRECT option and then records the response time of a read operation, which reads the 4KB

```
// write_unit: write request size (e.g., 4KB)
// write_inc: incremental write size (e.g., 4KB)
// read_unit: read request size (e.g., 4KB)
// max_unit: maximum write size (e.g., 5MB)
+ evictionspot_begin();
  while (write_unit <= max_unit) {
    // initialize the target storage
    storage_init(target_ssd);
    open(target_ssd, O_CREATE | O_DIRECT);
    // set write offset and write data
    lseek(target_ssd, 0, SEEK_SET);
    write(target_ssd, write_unit);
    // set read offset and read data
    lseek(target_ssd, 0, SEEK_SET);
    gettimeofday(&st, NULL);
    read(target_ssd, read_unit);
    gettimeofday(&et, NULL);
    printf("time %lld\n", elapsed_ms(&st, &et));
    // close file
    close(target_ssd);
    // update write request size
    write_unit += write_inc;
  }
+ evictionspot_end();
```

Figure 1: The pseudo code for detecting the *eviction spot*.

data written before (*i.e.*, write–read). This experiment implies that the read operation is handled by the write buffer because the write operation was just buffered on the write buffer inside the storage. On the other hand, The read latency of the flash memory (FR) is measured as the response time of a read operation after issuing the `FLUSH` command (*i.e.*, write–`FLUSH`–read). Table 2 shows the measured latency results; the latency of BR is in the range of 0.029ms to 0.057ms, whereas the latency of FR is in the range of 0.107ms to 3.446ms.

Detecting the Eviction Spot. The results from the first benchmark imply that if the latency of a read operation is larger than 0.1ms, the data has been forced to the flash memory by an `EVICTION` operation. The second benchmark can detect the *eviction spot* by measuring the response time for reading data at the zero offset after a write operation. Figure 1 specifies the pseudo code for detecting the *eviction spot*. For a fair comparison, it initializes the target storage by performing a series of operations, such as `umount()`, `fdisk()`, `format()`, and `mount()` operation, each time. Figure 2 shows the detected *eviction spots* on the real flash storage (more details described in §5). As we expected, the storage starts evicting data in the write buffer, after the size of write request is increased approximately larger than 1.8MB (gray dots). Interestingly, Figure 2 confirms the fact that a read request is never handled by the write buffer when the size of write request is increased further beyond 2.3MB (black dots). In summary, gray dots in Figure 2 mean the *eviction spot*, and write requests of 2.3MB make the write buffer evict the buffered data while the write buffer is temporally buffering the incoming data.

3.2 FLUSH vs. EVICTION

In this section, we study the performance and effectiveness of an `EVICTION` operation to answer the following two questions:

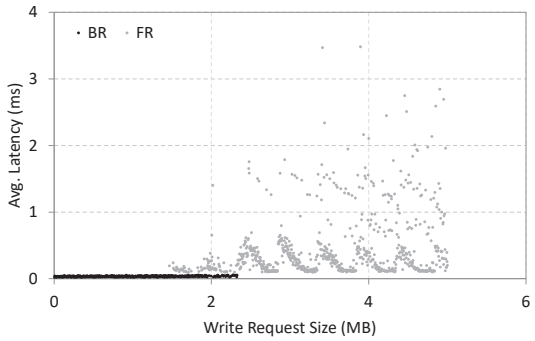


Figure 2: Buffering effect and *eviction spots*:*"BR" and "FR" indicate the buffer read and the flash read, respectively.*

- Which is faster between the `FLUSH` and the `EVICTION` in terms of latency measured by the host system?
- Can `EVICTION` ensure the same effect as the `FLUSH` command in terms of the data persistency?

Performance. As the storage devices and its interfaces (*e.g.*, SATA and NVM Express) have become dramatically faster, read and write operations also have become more lightweight. For example, SATA 3.0 interface provides the throughput of 6Gbps, two orders of magnitude faster compared with SATA 2.0 [1, 10].

In this section, we answer the first question. To measure the time during which data is persistently stored into the flash memory, we have implemented a microbenchmark that writes a few kilobytes data into the storage and then measures the `FLUSH` and `EVICTION` latencies. The `FLUSH` latency was measured as a response time of `fsync()` system call. The `EVICTION` latency was measured as a response time for writing 2.3MB data blocks that are padded with zero bit, we call the blocks *zero padding block* in the remainder of this paper. This is because *zero padding block* is big enough to trigger the `EVICTION` operation as mentioned before. We repeated the experiments in some different environments to confirm the performance gap in the SATA interface version.

Figure 3 shows the average latency of the `FLUSH` and the `EVICTION` while varying the amount of data written to the flash storage. There are two interesting results in Figure 3. First, the latencies of `FLUSH` commands are bounded by the storage, not the interface: the latencies of `FLUSH` show similar trends on different SATA versions. On the other hand, the latencies of `EVICTION` are bounded by the interface, not the storage; the trend on SATA 3.0 is similar to that on SATA 2.0, but the measured latencies of SATA 3.0 are shorter by up to two orders of magnitude than those of SATA 2.0. Second, surprisingly, even though the `EVICTION` additionally writes the 2.3MB data, it is faster than the `FLUSH` command in all cases when the flash storage is connected via SATA 3.0 interface. The reason behind these interesting results is that the `EVICTION` operation can defer the update of L2P table stored on the flash memory, as mentioned in §2.

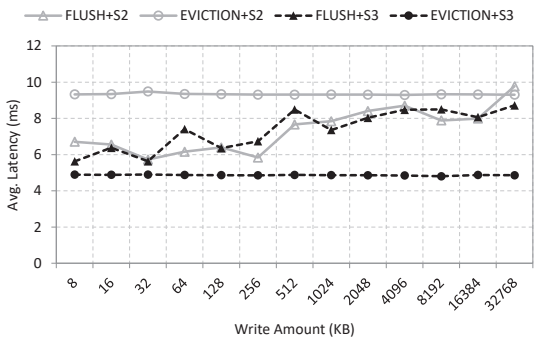


Figure 3: Average `FLUSH` and `EVICTION` response time: *"S2" and "S3" indicate the SATA interface version connected to the SSD.*

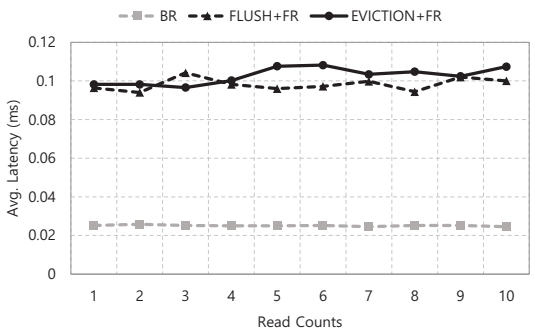


Figure 4: Average read latencies: *"BR" and "FR" indicate the buffer read and the flash read, respectively.*

Effectiveness. To answer the second question, we performed another evaluation with a microbenchmark. The microbenchmark first randomly writes 4KB data with `O_DIRECT` 10 times to make the flash storage accommodate incoming data to its write buffer. Then, it measures the read latency for each written data with three different conditions: BR, FLUSH+FR, and EVICTION+FR. BR does nothing and then reads the written data (*i.e.*, 10 writes–10 reads). FLUSH+FR issues `FLUSH` command before reading the data (*i.e.*, 10 writes–`FLUSH`–10 reads). Finally, EVICTION+FR sequentially writes *zero padding block* to lead the `EVICTION` operation inside the storage and then reads the written data (*i.e.*, 10 writes–*zero padding block* write-10 reads).

Figure 4 shows the results for each read latency. As we expected, in the case of BR, each read latency shows the performance of write buffer as shown in Figure 2 because of the buffering effect of the flash storage. Meanwhile, FLUSH+FR and EVICTION+FR show that each read operation is directly handled from the flash memory because the buffered data on the write buffer has been flushed to the underlying flash memory by `FLUSH` or `EVICTION` operation. Especially, the results of EVICTION+FR clearly confirm the fact that the write buffer algorithm has been designed to exploit the temporal locality (*i.e.*, Least Recently Used) [19, 22, 28]. This is because all data that was sorted in temporal order of writes has been simultaneously evicted from the write buffer by incoming new writes.

Figure 3 and Figure 4 verify that the `EVICTION` operation can provide better performance than that of `FLUSH` command and it also ensures the same effect as the `FLUSH` command in terms of data persistency. Such surprising benefits of the `EVICTION` operation motivated us to propose the *zero padding* scheme.

4. Design and Implementation

In this section, we introduce the *zero padding* scheme in detail and discuss its implementations in ext4 journaling mechanism.

Concept. *Zero padding* scheme is designed to make judicious use of the *write buffer* behaviors and SATA bandwidth. The key idea is to trigger the `EVICTION` operation from the write buffer inside the storage to ensure the same effect as physically issuing the `FLUSH` command. To achieve this, we fill up the write buffer by issuing a set of *zero padding block* to the flash storage until the previously buffered data is evicted from the write buffer (*i.e.*, *eviction spot*). This scheme is simple and straightforward and can be performed with relatively low cost due to the high bandwidth of SATA interface. In addition, it can considerably improve the throughput of the file system because the `EVICTION` operation does not need to immediately store the L2P table into the flash memory.

Journaling and Zero Padding. Since it is obvious that ext4 journaling calls many `FLUSH` commands to guarantee the consistency and durability of the file system [9], in this section, we describe how to gracefully replace the `FLUSH` commands of the ext4 journaling to `EVICTION` along with *zero padding block*, called *ext4-zp*. When a transaction commits, the traditional ext4 journaling issues a `FLUSH` command to the flash storage before and after writing a commit block to ensure the write ordering and to immediately store the commit block. As a result, the storage must reflect the whole buffered data and the corresponding L2P mapping table to the flash memory twice, consecutively. To avoid such a huge overhead, *ext4-zp* replaces the two `FLUSH` commands to the `EVICTION` operation with *zero padding block*, that does not require to update the L2P table, immediately. Figure 5 illustrates the `commit()` procedure of *ext4-zp*. *ext4-zp* goes through the following steps for each `commit()` operation.

1. *ext4-zp* writes journal data blocks (LPN 1–10), and those are directly stored on the write buffer inside the storage.
2. Unlike traditional ext4, *ext4-zp* fills up the write buffer by issuing the *zero padding block* (LPN 11–N), which triggers the `EVICTION` operation. As a result, the previous buffered blocks (LPN 1–10) on the write buffer are simultaneously evicted from the write buffer without updates of the L2P table.
3. Then, *ext4-zp* issues `TRIM` commands, which inform flash storage that the data has invalidated [21], to a set of LPNs of the previous *zero padding block* to prevent the *zero*

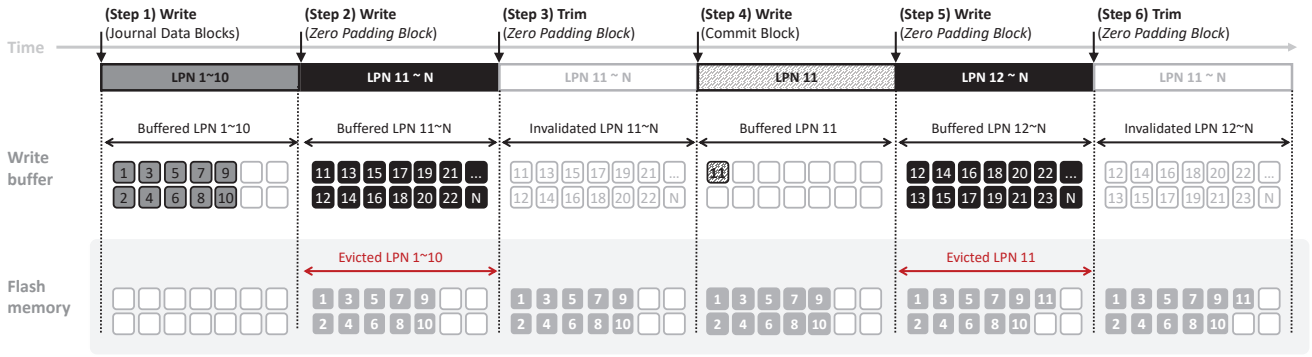


Figure 5: The overview of journaling process in *Ext4-zp*.

padding block on the write buffer from being flushed to the flash memory.

4. After completing the *zero padding block* invalidation, *ext4-zp* writes the commit block (LPN 11), inducing the block to be temporally buffered on the write buffer.
5. To guarantee a commit transaction, *ext4-zp* issues the *zero padding block* (LPN 12–N) and `TRIM` command once again. As a result, the commit block (LPN 11) is safely stored in the flash memory.

In this way, *ext4-zp* can guarantee the same consistency and durability of the ext4 file system without using any `FLUSH` commands. In addition, *ext4-zp* efficiently overcomes the wasted lifetime issue, which may be caused by flushing *zero padding block* to the flash memory, by exploiting the `TRIM` command. In *ext4-zp*, one negative fact is that *ext4-zp* incurs the interface overhead by linearly issuing `TRIM` commands. We can optimize *ext4-zp* by issuing the `TRIM` commands for all blocks in parallel instead of issuing them one after the other.

Recovery. As mentioned in §2.1, some flash storages provide the *OOB-hints* for L2P loss recovery. In this paper, we describe recovery procedure of *ext4-zp* assuming that *OOB-hints* are working as expected with the general flash storage devices. *ext4-zp* recovers the lost data based on commit blocks stored in the storage. Unlike ext4 file system, it uses *zero padding* to make the journal data and commit blocks persistent. Therefore, we now describe how to bridge the semantic gap between the data evicted from the write buffer and the corresponding L2P mapping table after system crash or power failure. When an entry of the write buffer is evicted, the flash storage generally stores the *logical addresses* (*i.e.*, P2L map) in its out-of-bound (OOB) area of the flash page along with the contents of the victim entry [11, 17, 23]. If a system crash or power failure occurs before flushing the L2P mapping table, the flash storage triggers the power-off-recovery (POR) operation at next boot time. The POR operation linearly scans the OOB area of each physical page belonging to the *active block* that was used to store incoming data before the system crash. It then recovers inconsistent L2P mapping table along with the corresponding P2L mapping information inside OOB area. Fortunately, this recovery operation is finished within a short period of time because the storage only scans the *active block* as mentioned in §2.1. As a result, *ext4-zp* completely recovers the contents of the entries evicted by the *zero padding blocks* without flushing the L2P mapping table.

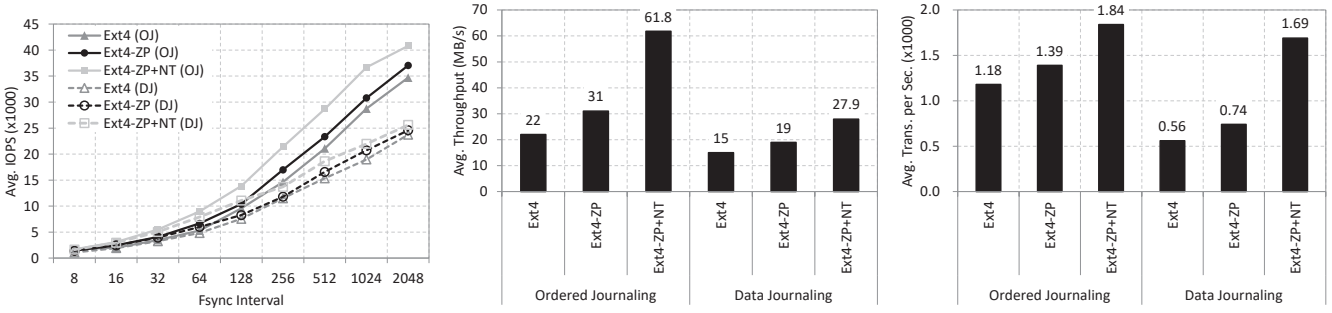
	Varmail		Sysbench	
	OJ	DJ	OJ	DJ
# of TXs	8213	3013	45949	22536
Minimum	3	3	3	3
Maximum	66	3569	896	8233
Average	17.2	60.9	3.1	52.5

Table 3: Analysis of real-world workloads:*"OJ" and "DJ" indicate the ordered mode journaling and the data mode journaling, respectively. The second, third, and fourth rows of the table shows the number of journal blocks per transaction.*

5. Evaluation

In this section, we present our evaluation results to answer the following question: *What are the zero padding's performance benefits and limitations on different workloads?*

Experimental setup. We have implemented the prototype of *ext4-zp* in Linux kernel 3.2.84 with a small modifications of ext4 and JBD2, and compared it with existing journaling modes, such as ordered and data journal mode. In order to measure how much performance overhead comes from the `TRIM` commands, we also implemented *no-trim* that skips the procedure of `TRIM` command of *ext4-zp* during execution. To stabilize the results, all experiments were repeated 10 times on a machine with an Intel i5-4670 3.4GHz 4-core CPU, 4GB of RAM, and a 128GB Samsung 850 Pro SSD. To clearly confirm the effect of the *zero padding*, we performed our evaluation using three kinds of microbenchmarks, including Flexible I/O [8], Filebench [7], and Sysbench [4]. Table 3 summarizes characteristics of the real-world workloads. It shows the number of journaled blocks per transaction that was measured at commit time based on Ext4 journaling modes on-the-fly.



(a) FIO (b) Varmail (c) Sysbench

Figure 6: Evaluation results for three different microbenchmarks:*"NT" indicates the no-trim.*

Flexible I/O (FIO) Microbenchmark. To simulate write-heavy workload, we first conducted a FIO workload with random write of 1GB data, a 4KB write granularity, and I/O queue depth of 1. We repeated the same FIO experiments with varying `fsync()` interval to confirm the relationship between the amount of data stored on the write buffer and `FLUSH` command. Figure 6(a) shows the performance in IOPS for each journaling mode. In this figure, we can see that as the `fsync()` interval increases, the IOPS of each journal mode is significantly improved. This is because the `FLUSH` command inefficiently blocks incoming write operations due to the data and the L2P table updates. Also, Figure 6(a) confirms that the *zero padding* improves the performance of the flash storage because it outperforms the conventional journal modes in all cases. Especially, In ordered mode, *ext4-zp* improves IOPS by up to 25% even though it includes the overhead of the `TRIM` command. Meanwhile, Figure 6(a) shows that *no-trim* outperforms *ext4-zp* in all cases. Especially, *no-trim* improves IOPS by up to 69% and 63% compared to ordered mode and data mode journaling, respectively.

Filebench Microbenchmark. To emulate a real-world workload, we used Varmail benchmark from Filebench. Varmail was configured with 16 threads, 1000 files, and a 16 KB file granularity, and each thread concurrently emulates a mail server by performing a set of operations: file create–append–sync and file read–append–sync. The read to write ratio is of this workload 1:1. Figure 6(b) compares the throughput of different journal modes in ext4. Compared with ordered mode, Figure 6(b) shows that *ext4-zp* results in up to 40% performance speedup. The reason behind these results is that Varmail frequently triggers the `fsync()` operation to ensure their data persistency and the traditional journal modes issuing the `FLUSH` command cannot avoid the L2P mapping table updates caused by the `fsync()` calls. On the other hand, *ext4-zp* can defer the L2P table update by replacing the `FLUSH` with the `EVICTION` with *zero padding block* on-the-fly. As shown in Figure 6(b), *no-trim* shows significant performance improvements: 2.8x and 86% to ext4 ordered and data journal mode, respectively. In addition, it results in up to 99% performance speedup compared with *ext4-zp*. These results clearly confirm that the `TRIM` overhead is non-negligible; *ext4-zp* always issues a set of `TRIM` commands to notify the underlying flash storage that *zero padding blocks* are no longer valid after triggering an `EVICTION` operation from the write buffer.

Sysbench Microbenchmark. In general, Sysbench is widely used to evaluate the online transaction processing (OLTP) workload. In this experiment, we measured MySQL [2] performance in transactions per second (TPS) with Sysbench. Figure 6(c) presents the results of transaction throughput. As we expected, *ext4-zp* achieves good performance even in the OLTP workload. Compared with ordered mode, *ext4-zp* improves the throughput by up to 17%. However, the performance gap among the journal modes is smaller than those of Figure 6(b). To understand this difference, we analyzed the characteristics of the two workloads and found out that they take different I/O patterns and frequency of `fsync()` operations; the ratio of read-write operations induced by this workload is 3:1.

In summary, our evaluation results clearly show that the proposed *zero padding* scheme successfully improves the throughput of the file system on three different workloads.

6. Discussion: Using Inference for Reliability

In this section, we discuss the *ext4-zp*'s limitations and an interesting future direction to mitigate them.

In this paper, our study is based on a Samsung SSD. Unfortunately, the internal structure and behaviors of a flash storage device are highly dependent on the manufacturers and their storage products. Especially, the OOB management policy and write buffer algorithm can be arbitrarily implemented by the manufacturer. Therefore, we should further explore flash storage devices on various aspects such as manufacturers (*e.g.*, Samsung, Intel, Micron), storage products (*e.g.*, NVMe, eMMC, SD Card), and environments (*e.g.*, mobile, desktop, enterprise).

We also explore a *conservative* implementation of *ext4-zp* that would guarantee correctness irrespective of unpredictable write buffer implementations. In particular, we en-

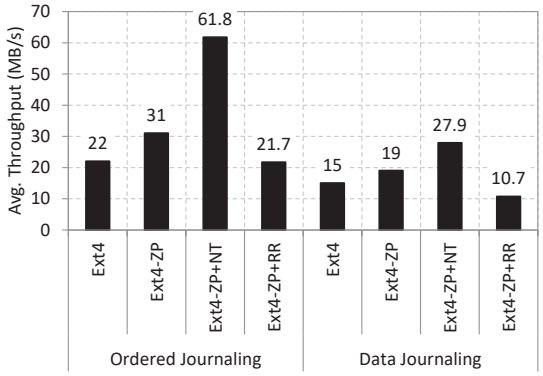


Figure 7: Evaluation results for Varmail microbenchmark:*"NT" and "RR" indicate the no-trim and re-read option, respectively.*

hanced *ext4-zp* with an additional option, *re-read*. We can use this option to confirm whether data belonging to the current commit transaction has been written to the flash memory, so that in the rare event that the block is not written to flash despite `EVICTION` through *zero padding* (due to some write buffer anomaly), we can fallback to a full flush (and thus be safe). When the *re-read* option is turned on, *ext4-zp* measures the latency for reading data included in the current commit transaction (*i.e.*, a journal descriptor block, journal data blocks, and a journal commit block) at a page granularity (*i.e.*, 4KB) and reports the location of data by comparing the measured result with the results in Table 2, before the transaction is completed, thus causing a higher overhead. Figure 7 shows the performance overhead of *re-read*. In this paper, we just show the results of Varmail microbenchmark since the results of Sysbench follow a similar pattern. As expected, the *re-read* option reduces the performance benefits of *ext4-zp* in both journal modes. However, as Table 3 shows, the distribution of the number of blocks per transaction is long-tailed. This gives us an opportunity to significantly cut down the *re-read* cost by adopting a hybrid approach. On a per-transaction basis, one could decide whether to use baseline `FLUSH` or *ext4-zp* with *re-read* based on the number of blocks in the transaction. For majority of transactions, the number of blocks is small, and hence *re-read* overhead would be small, and for the rest, one could do a regular flush. Combined with the parallel `TRIM` optimization above, we believe this hybrid design can significantly improve performance of the *zero padding* option. We plan to explore this in future work.

7. Related Work

A variety of mechanisms have been proposed to reduce the overhead of the `FLUSH` command that is caused by synchronous operations: `msync()`, `fsync()`, and `fdatasync()` [15, 16, 24, 26, 30, 31, 33, 34]. For example, Chen *et al.* [15], Shen *et al.* [33], and Prabhakaran *et al.* [30] revisited the existing journaling mechanism and modified it to adaptively trigger the journaling operation along with the `FLUSH` command. IRON file system [31] exploits a checksum scheme to make journal data persistent. Other researchers redesigned the file system to efficiently support data persistency. OptFS [16] employs new commands, such as `osync()` and `dsync()`, to decouple ordering of writes in terms of data durability. CFS [27] and CCFS [29] guarantees application-level consistency without redundant `FLUSH` commands. However, most previous efforts still leave the overhead of the `FLUSH` command that should force the data and the L2P mapping table from the write buffer to the flash memory immediately. The most similar work to our work is the coerced cache eviction (CCE) proposed by Rajimwale *et al.* [32]. Like us, the CCE issues extra writes to the special area, called *flush-zone*, to replace the `FLUSH` command to the `EVICTION` operation. However, since this approach only focused on the HDD, it has limitations on the flash storage. Especially, the CCE significantly shortens the lifetime of the flash storage because extra writes caused by the CCE always are flushed to the flash memory. On the other hand, we uses the `TRIM` command to inform the FTL inside the flash storage that *zero padding block* should be invalidated. Thus, our scheme efficiently prevents shortening the lifetime of the flash storage, while improving the overall performance.

8. Conclusion

In this paper, we introduced *zero padding* scheme that persistently evicts data stored on the write buffer to the flash memory and boosts substantial performance improvements. Also, we applied *zero padding* to the existing journaling mechanism to exploit the characteristics of journaling, such as sequential writes and round-robin policy. Our evaluation results confirm that *zero padding* is an alternative scheme to the `FLUSH` command because it improves the throughput of the file system by up to 40% compared with ordered mode journaling. Unfortunately, in this paper, we did not show the evaluation results on the correctness of *ext4-zp* because a recovery policy (*e.g.*, OOB area) inside a flash storage may be different for each manufacturer. In future, we would perform additional evaluation on the correctness issue and clearly show the results. We hope our study will encourage storage developers and researchers to better understand the semantic gap between the `FLUSH` operation and internal behaviors of the flash storage device.

Acknowledgments

We wish to thank Muthian Sivathanu, our shepherd, and the anonymous reviewers for the insightful comments. 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 & Future Planning (NRF-2015M3C4A7065696) and was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIP) (No. NRF-2017R1A2B3004660).

References

[1] Serial ATA International Organization: Serial ATA Revision 3.1. Technical report, ATA International Organization, July 2011.

[2] MySQL 5.7 Reference Manual. http://dev.mysql.com/doc/refman/5.7/en/, 2015.

[3] Intel Solid-State Drive DC S3700 Series: Specification. http://www.intel.com/content/www/us/en/solid-state-drives/ssd-dc-s3700-spec.html, 2015.

[4] SysBench (Branch 1.0). https://github.com/akopytov/sysbench, 2016.

[5] Disk Buffer. https://en.wikipedia.org/wiki/Disk_buffer, 2016.

[6] Power Loss Protection: How SSDs Are Protecting Data Integrity. https://insights.samsung.com/2016/03/22/power-loss-protection-how-ssds-are-protecting-data-integrity-white-paper, 2016.

[7] Filebench. https://github.com/filebench/filebench/wiki, 2017.

[8] FIO (Flexible IO Tester). https://github.com/axboe/fio, 2017.

[9] Ext4 Disk Layout. https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout, 2017.

[10] Serial ATA. https://en.wikipedia.org/wiki/Serial_ATA, 2017.

[11] M. Bjørling, J. Gonzalez, and P. Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proceedings of the USENIX conference on File and Storage Technologies*, FAST '17, pages 359–373, 2017.

[12] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly Media, 2005.

[13] F. Chen, R. Lee, and X. Zhang. Essential Roles of Exploiting Internal Parallelism of Flash Memory based Solid State Drives in High-speed Data Processing. In *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 266–277, 2011.

[14] F. Chen, T. Luo, and X. Zhang. CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives. In *Proceedings of the USENIX conference on File and Storage Technologies*, FAST '11, pages 1–14, 2011.

[15] Q. Chen, L. Liang, Y. Xia, and H. Chen. Mitigating Sync Amplification for Copy-on-write Virtual Disk. In *Proceedings of the USENIX conference on File and Storage Technologies*, FAST '16, pages 241–247, 2016.

[16] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '13, pages 228–243, 2013.

[17] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the ACM international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 229–240, 2009.

[18] D. Jeong, Y. Lee, and J.-S. Kim. Boosting Quasi-Asynchronous I/O for Better Responsiveness in Mobile Devices. In *Proceedings of the USENIX conference on File and Storage Technologies*, FAST '15, pages 191–202, 2015.

[19] H. Jo, J.-U. Kang, S.-Y. Park, J.-S. Kim, and J. Lee. FAB: Flash-Aware Buffer Management Policy for Portable Media Players. *IEEE Transactions on Consumer Electronics*, 52(2): 485–493, 2006.

[20] W.-H. Kang, S.-W. Lee, B. Moon, Y.-S. Kee, and M. Oh. "Durable Write Cache in Flash Memory SSD for Relational and NoSQL Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 529–540, 2014.

[21] B. Kim, D. H. Kang, C. Min, and Y. I. Eom. Understanding Implications of Trim, Discard, and Background Command for eMMC Storage Device. In *Proceedings of the IEEE 3rd Global Conference on Consumer Electronics*, GCCE '14, pages 709–710, 2014.

[22] H. Kim and S. Ahn. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In *Proceedings of the USENIX conference on File and Storage Technologies*, FAST '08, pages 1–14, 2008.

[23] H. Kim, D. Shin, Y. H. Jeong, and K. H. Kim. SHRD: Improving Spatial Locality in Flash Storage Accesses by Sequentializing in Host and Randomizing in Device. In *Proceedings of the USENIX conference on File and Storage Technologies*, FAST '17, pages 271–283, 2017.

[24] W.-H. Kim, B. Nam, D. Park, and Y. Won. Resolving Journaling of Journal Anomaly in Android I/O: Multi-Version B-tree with Lazy Split. In *Proceedings of the USENIX conference on File and Storage Technologies*, FAST '14, pages 273–285, 2014.

[25] H. Kumar, Y. Patel, R. Kesavan, and S. Makam. High-Performance Metadata Integrity Protection in the WAFL Copy-on-Write File System. In *Proceedings of the USENIX conference on File and Storage Technologies*, FAST '17, pages 197–211, 2017.

[26] 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 *Proceedings of the USENIX Annual Technical Conferences*, ATC '15, pages 235–347, 2015.

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

[28] S.-Y. Park, D. Jung, J.-U. Kang, J.-S. Kim, and J. Lee. CFLRU: A Replacement Algorithm for Flash Memory. In *Proceedings of the ACM International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '06, pages 234–241, 2006.

[29] 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 *Proceedings of the USENIX conference on File and Storage Technologies*, FAST '17, pages 181–196, 2017.

[30] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *Proceedings of the USENIX Annual Technical Conferences*, ATC '05, pages 1–16, 2005.

[31] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the ACM symposium on Operating systems principles*, SOSP '05, pages 206–220, 2005.

[32] 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 *Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems & Networks*, DSN '11, pages 518–529, 2011.

[33] K. Shen, S. Park, and M. Zhu. Journaling of Journal Is (Almost) Free. In *Proceedings of the USENIX conference on File and Storage Technologies*, FAST '14, pages 287–293, 2014.

[34] Y. Zhang, C. Dragga, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. ViewBox: Integrating Local File Systems with Cloud Storage Services. In *Proceedings of the USENIX conference on File and Storage Technologies*, FAST '14, pages 119–132, 2014.