# Lazy TRIM: Optimizing the Journaling Overhead caused by TRIM commands on Ext4 File System

Kitae Lee[†‡], Dong Hyun Kang[†], Daeho Jeong[‡], and Young Ik Eom[†]
[†]Sungkyunkwan University, South Korea [‡]Samsung Electronics, South Korea
{lkt8711, kkangsu,yieom}@skku.edu {kitae87.lee,daeho.jeong}@samsung.com

***Abstract*—Nowadays, NAND flash storage devices have become a standard for secondary storage in consumer electronics devices. Unfortunately, *TRIM* command for the flash storage causes journaling overhead on the ext4 file system. In this paper, to figure out the root cause of the overhead, we study the relationship between ordered mode journaling and the *TRIM* command. Then, we propose a novel scheme, called *lazy TRIM*, that adopts multi-threading in background to gracefully distribute the journaling overhead caused by the *TRIM* command. Our evaluation results clearly confirm that *lazy TRIM* improves the latency of the fsync system call by up to 99% compared with the conventional scheme.**

## I. Introduction

Recently, consumer electronic (CE) devices use NAND flash storage devices (*e.g.*, eMMCs, UFSs, SD Cards, SSDs) as their primary storage media [3], [6]. This is because the flash storage provides several advantages such as compact size, fast access time, and low battery consumption. Unfortunately, the flash storage devices do not allow in-place updates. In order to overcome this, most flash storage devices employ a special software, called flash translation layer (FTL), that maps an incoming logical block address (LBA) to physical block address (PBA). In other words, when an update request on the existing data occurs, FTL first writes the data into a new page (*i.e.*, new physical block address) and then replaces the corresponding old physical block address to the new one. As a result, the old block address is invalidated and the new one is available on the FTL mapping.

The *TRIM* command is designed to inform the flash storage of LBAs which belong to the deleted file in the file system. This command helps to mitigate the overhead of garbage collection (GC) inside the flash storage because FTL can exploit a set of LBAs within the *TRIM* commands to avoid unnecessary block and page copy operations during GC [1].

However, the *TRIM* command would cause an additional interface overhead on-the-fly. Unsurprisingly, in order to reduce the overhead, many research groups focused on the *TRIM* command. Some researchers proposed the judicious triming scheme that selectively issues the *TRIM* command to the flash storage [2], and other researchers focused on the timing when the *TRIM* is issued to the storage to avoid negative effects on the performance [4], [7].

In this paper, we also study the *TRIM* command. However, unlike previous work, we focus on the command in terms of journaling on ext4 file system. This is because we found out that the *TRIM* command significantly delays a commit time of the journaling when the *TRIM* command option is turned on. We also propose a novel scheme, called *lazy TRIM*, that efficiently distributes the journaling overhead caused by *TRIM* commands by processing the commands in parallel. Our scheme also helps to reduce the number of *TRIM* executions by canceling some *TRIM* commands to reuse the LBAs in them for next block allocation. Our experimental results show that our scheme improves the latency of fsync by up to 99% and the time necessary for issuing total *TRIM* commands by up to 60%, compared with the traditional scheme.

This research was supported by Basic Science Research Program (NRF-2017R1A2B3004660) and Next-Generation Information Computing Development Program (NRF-2015M3C4A7065696) through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT.

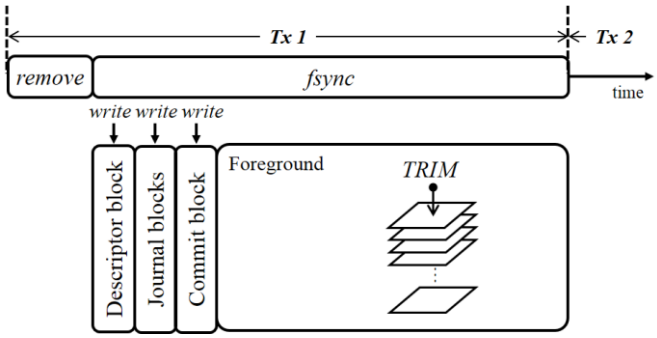


Fig. 1. Traditional ordered mode journaling on the ext4 filesystem

## II. Background

In this section, we study the relationship between the ordered mode journaling and the *TRIM* command. Figure 1 shows the procedure of file deletion and commit operation with an example when the *TRIM* option is turned on at mount time [9]. If a user unlinks a file on the file system (*i.e.*, delete a file) and then calls the synchronous system call (*e.g.*, `fsync`, `fdatasync`), the journal daemon inside the kernel triggers a *journal commit* operation as in the following steps. First, the daemon writes a journal descriptor block and journal blocks (metadata blocks) which are modified by removing the file into the journal area. Second, the daemon writes a commit block after the metadata blocks are persistently stored on the flash storage [8], [10]. Third, the daemon issues a set of *TRIM* commands that include LBAs for data blocks belonging to the deleted file. Finally, the commit procedure is finished after issuing the flush command to the underlying flash storage [5].

In the commit procedure, the third step is very interesting because the data blocks are reflected after the metadata block is issued to the flash storage despite of the ordered mode


978-1-5386-3025-9/18/$31.00 ©2018 IEEE

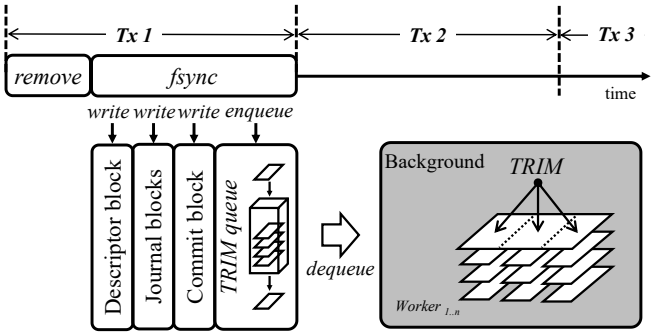


Fig. 2. Ordered mode journaling with *lazy TRIM* on the ext4 filesystem

journaling; in the case of an append operation, the data block is always reflected before issuing the metadata block. Such a *reverse order* operation is necessary to maintain the consistency of the file system because it must prevent the case that the metadata points to the data block that was invalidated by the *TRIM* command. Unfortunately, the third step negatively affects on the overall performance because the commit procedure should wait until all *TRIM* commands are transferred to the underlying flash storage. In addition, since the traditional journaling scheme issues the *TRIM* command in a linear way, the time for commit transaction would be delayed for long period of time. This observation motivates us to propose *lazy TRIM* scheme.

## III. Design and Implementation

In this paper, our goal is to optimize the journaling overhead that is caused by *TRIM* command on ext4 file system. In order to achieve our goal, we carefully designed *lazy TRIM* scheme for journaling on ext4 file system. The key idea of *lazy TRIM* is to replace the serialized *TRIM* commands to the parallelized one.

### A. Lazy TRIM

To follow the procedure of the journaling transaction, *lazy TRIM* is designed by extending ext4 file system and the journaling parts that are correlated with the *TRIM* command. To distribute the journaling overhead efficiently, the *lazy TRIM* employs a *worker thread* that uses the *TRIM queue* to communicate with the journal daemon, where it gives three advantages. First, the *worker thread* helps to issue the *TRIM* commands in a parallel and lazy manner. Second, the *lazy TRIM* uses system resources optimally by increasing or decreasing the number of *worker threads* automatically according to the number of *TRIM* commands to be issued. Third, since the *worker thread* runs in background, it can significantly reduce the latency of the synchronous system call.

Figure 2 shows how our scheme handles the *TRIM* command. When a synchronous call is triggered after a user deletes a file, we adopt *lazy TRIM* in the third step of the *journal commit* operation by going through the following steps. (1) *lazy TRIM* inserts all *TRIM* commands into the *TRIM queue* that is implemented as linked lists, instead of issuing the commands immediately. (2) it wakes up the *worker thread* that has been registered at boot time, and then returns immediately. This is possible because the queued *TRIM* commands will be handled by the *worker thread* in background. (3) the *worker thread* dequeues the *TRIM* commands from the *TRIM queue* and then issues them. Our scheme adjusts the number of threads according to the number of queued *TRIM* commands. In this way, *lazy TRIM* can efficiently reduce the journaling overhead incurred by issuing the *TRIM* commands during the commit operation.

TABLE I
Samsung Galaxy S8 Specification

| | |
|---|---|
| Android version | 7.0 Nougat |
| Kernel version | 4.4.13 |
| Processor | Exynos 8895 SoC,<br>Custom CPU MP4 2.3GHz CPU + ARM Cortex-A53 MP4 1.7GHz CPU |
| Storage | 64GB UFS 2.1 |
| Memory | 4GB LPDDR4 |

### B. Reuse TRIM

Using the *TRIM queue* gives us an additional optimization opportunity to reduce the number of queued *TRIM* commands by reusing the LBAs that are invalidated by the *TRIM* command. The reason behind such an optimization is that overwriting the existing LBAs provides the same effect as *TRIM* commands because the flash storages do not support in-place update feature. In this paper, we call this optimization as *reuse TRIM*. The procedure of the *reuse TRIM* is very simple and straightforward. When a user requests a block allocation to append a file, we dequeue one of the LBAs that belongs to the *TRIM* command by scanning the *TRIM queue*. Then, we return the dequeued LBA as the result of the block allocation without issuing the *TRIM* command to the storage. As a result, this feature meaningfully reduces the number of LBAs that belong to the queued *TRIM* commands on-the-fly.

## IV. Performance Evaluation

In order to verify the effectiveness of *lazy TRIM* along with *reuse TRIM*, we implemented a prototype of our scheme by modifying both ext4 file system and journaling daemon (*i.e.*, JBD2) on the real consumer devices, Samsung Galaxy S8. Table I shows the specification of the device.

To compare our scheme with the traditional approach, we developed a micro-benchmark. The benchmark creates a file and appends data of predefined size from 64KB to 64MB along with lseek operation alternately until the total size of the file becomes 1GB. Due to the lseek between write operations, the file is fragmented according to the append size (*i.e.*, the smaller the append size is, the larger the number of fragments in the file). The number of fragments is strongly related to the number of *TRIM* commands because one *TRIM* command includes only one fragment.

We ran the benchmark by varying the append size from 64KB to 64MB and removed the file to measure the latency of

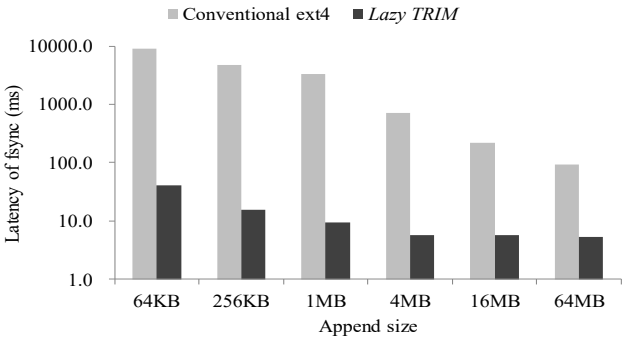


Fig. 3. Fsync latency comparison between the traditional and *lazy TRIM*

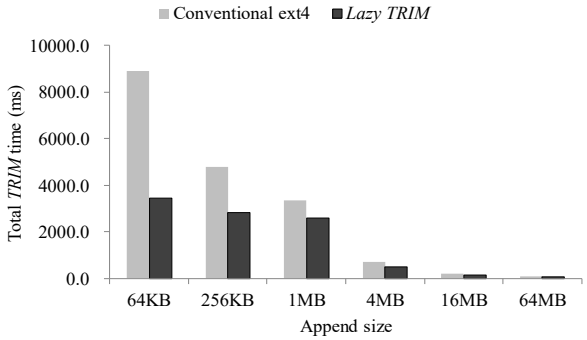


Fig. 4. The elapsed time required to issue all *TRIM* commands

TABLE II
WRITE LATENCY COMPARISON FOR NEW BLOCK ALLOCATION

| Traditional ext4 | *Lazy TRIM* | *Lazy reuse TRIM* |
|---|---|---|
| 2ms | 840ms | 2ms |

fsync. Figure 3 shows the evaluation results in terms of the latency of fsync. The X axis is the append size of each file, and the Y axis is the latency in log scale. As shown in Figure 3, the traditional approach reveals poor performance, compared with our scheme. In addition, it significantly increases the latency as the number of fragments is larger (*i.e.*, 64KB). Meanwhile, *lazy TRIM* outperforms the traditional approach in all cases. Especially, our scheme reduces the latency by up to 99% compared with the traditional one when the append size is 64KB. In addition, Figure 3 shows that the performance gap of our scheme among the append sizes is smaller than that of the traditional approach.

To deeply understand the evaluation results, we measured the time required to issue the *TRIM* commands. Figure 4 shows the measured time where, as we expected, the traditional approach need a long time to issue all the *TRIM* commands because it transfers the commands in a sequential way. In addition, since the commands are synchronously handled in foreground, this approach would exacerbate overall performance. On the other hand, *lazy TRIM* handles all *TRIM* commands in a short time. These results are possible because our scheme issues the *TRIM* commands in a parallel way in background. In the best case, our scheme is 60% faster than the traditional approach.

To prove the effectiveness of *reuse TRIM*, we measured the latency of a write operation that requires a new block. For the experiment, we first wrote data until the utilization of the mobile storage reaches 100%. Then, to measure the latency of a block allocation, we issued the write operation right after randomly removed a file on the file system. Table II shows the measured latency for the write operation.

As shown in Table II, *lazy TRIM* leads to delay the block allocation. The reason behind this result is that the *lazy TRIM* should postpone the new block allocation until the *worker thread* issues the queued *TRIM* commands to the flash storage. Meanwhile, the result of the *lazy TRIM* with *reuse TRIM* (*lazy reuse TRIM*) is similar to that of the traditional approach. This is because *lazy reuse TRIM* does not need to wait by reusing one of the LBAs that belong to the *TRIM* command on the *TRIM queue*. As a result, *lazy reuse TRIM* can support fast block allocation and parallelized *TRIM* commands without any overhead.

## V. CONCLUSION

In this paper, we propose a novel approach, *lazy TRIM*, that mitigates the journaling overhead that is caused by the *TRIM* command during the procedure of commit transaction. The key idea of *lazy TRIM* is to replace the serialized *TRIM* commands to the parallelized one that is running in background. Our experimental results show that *lazy TRIM* along with *reuse TRIM* significantly improves the performance in terms of latency and elapsed time. In this paper, we only evaluated our scheme with the synthetic workloads, and we have a plan to evaluate *lazy TRIM* with the real-world workloads in the future.

## REFERENCES

[1] J. Kim, H. Kim, S. Lee, and Y. Won, "FTL design for TRIM command," in *Proc. of the 5th International Workshop on Software Support for Portable Storage*, pp. 7-12, 2010.

[2] C. Hyun, J. Choi, D. Lee, and S. H. Noh, "To TRIM or not to TRIM : Judicious triming for solid state drives," *Poster presentation in the 23rd ACM Symposium on Operating Systems Principles*, 2011.

[3] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," in *Proc. of the 10th USENIX File and Storage Technologies, pp. 209-222*, 2012.

[4] B. Kim, D. H. Kang, C. Min, and Y. I. Eom, "Understanding implications of trim, discard, and background command for eMMC storage device," in *Proc. of the 3rd IEEE Global Conference on Consumer Electronics*, pp. 709-710, 2014.

[5] D. Jeong, Y. Lee, and J.-S. Kim, "Boosting quasi-asynchronous I/O for better responsiveness in mobile devices," in *Proc. of the 13th USENIX File and Storage Technologies*, pp. 191-202, 2015.

[6] S. J. Han, D. H. Kang, and Y. I. Eom, "Hybrid write buffer algorithm for improving performance and endurance of NAND flash storages," in *Proc. of the 34th IEEE International Conference on Consumer Electronics,* pp. 93-94, 2016.

[7] K. Kwon, D. H. Kang, J. Park, and Y. I. Eom, "An advanced TRIM command for extending lifetime of TLC NAND flash-based storage," in *Proc. of the 35th IEEE International Conference on Consumer Electronics,* pp. 443-444, 2017.

[8] D. P. Bovet and M. Cesati, *Understanding the linux kernel, 3-ed, O'Reilly*, pp. 772-774, 2008.

[9] Ext4 howto [Online]. Available: https://ext4.wiki.kernel.org/index.php/Ext4_Howto

[10] Ext4 layout [Online]. Available: https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout