RFLUSH: Rethink the Flush

Jeseong Yeon and Minseong Jeong, *Chungbuk National University;* Sungjin Lee, *DGIST;*
Eunji Lee, *Chungbuk National University, University of Wisconsin—Madison*

https://www.usenix.org/conference/fast18/presentation/yeon

This paper is included in the Proceedings of the
16th USENIX Conference on File and Storage Technologies.
February 12–15, 2018 • Oakland, CA, USA
ISBN 978-1-931971-42-3

RFLUSH: Rethink the Flush

Jeseong Yeon*, Minseong Jeong*, Sungjin Lee†, Eunji Lee*‡
*Chungbuk National University, †DGIST, ‡University of Wisconsin–Madison
{jsyeon, msjeong}@oslab.cbnu.ac.kr, sungjin.lee@dgist.ac.kr, eunji@cbnu.ac.kr

Abstract

A FLUSH command has been used for decades to enforce persistence and ordering of updates in a storage device. The command forces *all* the data in the volatile buffer of the storage device to non-volatile media to achieve persistency. This *lump-sum* approach to flushing has two performance consequences. First, it slows down non-volatile materialization of the writes that actually need to be made durable. Second, it deprives the writes that do not need to be made durable of an opportunity for absorbing future writes and coalescing.

We attempt to characterize the problems of this *semantic gap* of flushing in storage devices and propose RFLUSH that allows a fine-grained control over non-volatile materialization. The RFLUSH command delivers a range of logical block addresses (LBAs) that need to be flushed and thus enables the storage device to force only a subset of data in its buffer.

We implemented this fine-grained flush command in a storage device using an open-source flash development platform and modified the F2FS file system to make use of the command in processing fsync requests as a case study. Performance evaluation using the prototype shows that the inclusion of RFLUSH improves the throughput by up to 6.5x; reduces the write traffic by up to 43%; and eliminates the long tail in the response time.

1 Introduction

Historically, storage devices have made use of a volatile buffer for various purposes. For hard disk drives (HDDs), the volatile buffer has been used for absorbing writes and minimizing seeks, while solid state drives (SSDs) have used the buffer for improving their random write performance and masking the limited endurance of the underlying non-volatile media [6, 13, 15, 19, 38, 39, 44].

The adoption of a volatile buffer, however, can bring with it data loss and improper ordering of updates in a power outage. The FLUSH command has been introduced to resolve this issue; forcing *all* the pending writes to non-volatile media, ensuring persistence and proper serialization of updates.

Unfortunately, this *lump-sum* approach to enforcing persistency has undesired performance consequences [6, 13, 38, 39, 44]. To faithfully implement the flush semantics, the storage device must empty all the dirty pages in its volatile buffer, whereas a flush request is commonly issued with less stringent requirements. As an example, consider a concurrent execution of two applications: an on-line banking application that requires to persist each transaction immediately, and a big-data analytics application that writes a large amount of intermediate results, which is a common scenario in modern complicated and multi-tenant storage platforms. In this scenario, a flush request for a committed transaction by the banking application will end up with forcing a large amount of dirty data (most of it from the analytics application, and thus irrelevant) in the storage device, which slows down what is actually needed (forcing the dirty data from the banking application).

This paper attempts to cure the performance problem of the conventional flush mechanism outlined above by refactoring the storage device interface. The refactoring is to include a command called RFLUSH (Range Flush) which allows a fine-grained control over non-volatile materialization of dirty data in the buffer. The RFLUSH command transfers a range of logical block addresses (LBAs) that specifies data to be persisted with it, helping the storage device to optimize its non-volatile materialization. This command not only speeds up the non-volatile materialization of the target LBAs but also enhances buffering and coalescing of other dirty data in the buffer.

Our work is in line with a collection of recent studies. In the past, computer systems have been built upon a standard block device interface consisting of a small set of commands over its logical address space: read, write, and flush. This abstract view of a storage device allows a host system to readily access non-volatile media in an efficient manner. However, as emerging storage media such as flash memory and other non-volatile memories (NVMs) are more commonly used, the possibility of extending the conventional block device interface to leverage the full potential of the new storage media is being actively explored [1, 4, 7, 9, 23, 24, 27, 31, 32, 36, 46].

A TRIM command has been proposed to prevent useless data from being copied around and lowering the endurance of flash memory [36]. As another example, recent storage device interfaces support atomic writes, which can be efficiently supported in flash-based storage devices [7, 31]. Also, storage interface extensions such as those for delegating block allocation [1, 9, 27, 32, 46], multi-streamed SSDs [18, 29], host manageable storage devices [4, 23], and user programmable SSDs [35] have been studied to provide an extended functionality and/or achieve better performance in high-end storage systems.

The benefits of `RFLUSH` seem straightforward, but realizing it efficiently in a storage device and augmenting file systems and/or database systems to make an effective use of it are not without challenges. We implemented `RFLUSH` in a storage device using an open-source flash development platform [23] and modified a file system (F2FS) [22] to make use of the extended interface[1]. The modified F2FS uses the `RFLUSH` command in the handling of `fsync` (and its variants). In this way, user applications do not need to be modified since the interface (i.e., `fsync`) and its semantics are faithfully preserved.

The rest of this paper is organized as follows. We give our motivation for `RFLUSH` and briefly review the related technology trends (§2). We then present the `RFLUSH` command and describe its prototype implementation (§3). We present results from performance evaluation using the prototype (§4), and finally conclude (§5).

2 Motivation and Related Work

Flush Optimization using Non-volatile Memory: Many prior works have pointed out that the in-storage buffer flush is a critical contributor to performance variation and unexpected slowdown in storage devices [15, 19]. One approach to lessening the detrimental performance effects of flush is to use super capacitors for providing enough energy to force all the dirty data in the volatile buffer at the time of a power outage. SSD manufacturers incorporate super capacitors in their high-end SSD devices to make them tolerant on power outages, offering high performance and reliability at the same time [21]. As a similar approach, Xiangfeng presents a modern SSD architecture that uses non-volatile memory for a write buffer while maintaining a read cache as volatile [44]. However, both approaches intrinsically increase the manufacturing cost, resulting in lower competitiveness of the intended products. The two approaches are, however, complementary to `RFLUSH` in the sense that they allow the `RFLUSH` command to return immediately while giving priority for replacement to those dirty data that were the target of the command to make room in the buffer for future writes.

Flush Optimization in a Host: The problem of flushing mechanism also exists in a page cache between a host and a device, because the page cache adopts a flushing mechanism to ensure persistence and ordering of updates in its volatile buffer. As opposed to a storage intreface, POSIX file system interfaces provide fine-grained control over the flushing mechanism through `fsync` and `fdatasync` system calls, in addition to `sync`. However, the flushing activity is still costly in a larger size page cache, and thus there have been numerous studies to mitigate this problem. Likewise as on the storage side, specialized hardware such as battery-backed main memory has been considered to avoid flushing cost [5, 43]. As a software-based approach, Nightingale et al. present an externally synchronized file system called `xsyncfs` [30], which allows an application to avoid blocking during the long-latency synchronization. The `xsyncfs` allows a requesting application to immediately return from the synchronization request, but makes the updates visible when they become consistently durable, leading to improvement in responsiveness. Chidambaram et al. present a new crash-consistency protocol that decouples ordering and durability, thereby providing data consistency with high performance [6]. Instead of forcing a low-level disk promptly to flush its buffer, they allow a storage device to optimize a flushing mechanism within a time limit, while still satisfying the ordering constraints. While such optimization obtained through a trade-off between durability and performance is worthwhile to consider in storage interface extension, this paper, as an initial and fundamental approach, focuses on storage interfaces for enhancing performance without any compromising of durability.

SSD Trends: The demand to improve the flush interface is particularly high at this moment because the cost of a flush is amplified when it is combined with next-generation SSD technologies. As host interfaces such as the NVMe [10, 16] become fast, the performance bottleneck is being shifted from the host interface to the flash device. The flash memory latencies for reading, programming, and erasing are also steadily increasing although device density is improving. Therefore, the latest SSDs attempt to use an increasingly larger buffer (e.g., 512 MB to 2 GB) to compensate for flash memory's low performance and endurance [25, 33, 34, 37]. With this trend, it is obvious that cache flushing results in more serious performance degradations in the presence of a larger buffer.

Besides, there are SSDs that exploit a portion of the host memory as a dedicated in-storage buffer, which may seriously suffer from cache flushing. Such SSDs help to improve the performance while cutting off the cost by not using DRAM in the storage device [8, 33], but a tandem with a classical flush interface might incur GBs of data being flushed from the host to the storage device on a regular basis. Considering the high cost of data transfer between the host and the device, the existing flush mechanism would degrade the storage performance severely.

[1] https://github.com/jsyeon92/RFLUSH

Also, the page size of flash memory is getting bigger, which will affect the overall performance as an eager flushing forfeits the possibility of consolidation and re-alignment of pending writes, yielding a large number of underutilized pages [20].

High Demand on Isolation The need for improving the flush interface is also evident with respect to performance isolation. With the latest innovations of data centers, computation is rapidly being moved from standalone desktops to cloud systems. With this trend, performance isolation and accurate accounting across applications are more important than ever. Techniques for isolating storage performance on the host side have been researched extensively. IceFS isolates related data with a container-based grouping and eliminates shared physical resources or access dependencies among containers in a file system [13]. Differentiated Storage Services (DSS) [26] and IOFlow [41] propose to tag data across layers to determine which process issues a request at any given layer. Yang et al. present a split-level I/O scheduling framework that provides a set of hooks for acquiring knowledge needed for accurate accounting and fair scheduling [45].

However, not much research has been performed on the storage side to prevent interference among applications. Prior works on in-storage buffers mostly focus on the replacement policy [15, 19], and there is not much previous research on curing the inefficiency of the flush mechanism despite its huge impact on the performance and endurance of the storage device. We believe our analysis and proposal in this paper are highly timely and contribute to driving the storage interface to be in harmony with fast-advancing storage technologies.

3 Range Flush

The concept of `RFLUSH` is simple but there are many design issues to be addressed since it involves from the application down to the storage device. In §3.1, we discuss places where `RFLUSH` can be useful. Then, in §3.2, we explain how to identify data related to `RFLUSH`. Data associated with `RFLUSH` is not limited to user data but includes metadata. In §3.3, we discuss how to handle metadata for `RFLUSH`. We describe how to integrate `RFLUSH` into storage protocols in §3.4.

3.1 Where to Use `RFLUSH`

Since `RFLUSH` is more general than its counterpart `FLUSH` and allows finer-grained control over what to flush, there can be many use cases where it can be effective. In this paper, we focus on its use for optimizing the `fsync` and `fdatasync` system calls. (Hereafter we use `fsync` to denote both `fsync` and `fdatasync`.)

There are some obvious benefits in implementing `fsync` using `RFLUSH`. First, no application modifications are needed since the `fsync` semantics can be faithfully preserved. Second, information about the user data and metadata that are affected by the `fsync` is readily available. Third, there can be noticeable performance gains from isolating regions to flush by `fsync`.

Although we leave for future research the use of `RFLUSH` by the file system itself other than in the processing of the `fsync`, we can easily identify other potential use cases for `RFLUSH`. For example, many file systems use journaling for recovery purposes and they typically use write-ahead logging (WAL) [28] that requires logging be performed before the logged updates are written to their home locations. The `RFLUSH` command can be used to give priority to the non-volatile materialization of data in the log. The same write-ahead logging is used by almost all the database systems today and they can be equally benefited by the use of `RFLUSH`.

3.2 How to Identify the Associated Data

The next challenge in using `RFLUSH` lies in how to identify the associated data for a given `fsync` request. The file system needs to identify the set of pages that are associated with a file and thus has to be forced to persist. Among such pages, some are in the page cache in a dirty state. The file system can flush such pages to the storage device followed by an `RFLUSH` command targeting them. A problematic case is when some of the pages that need to be forced to persist have already been sent to storage, meaning that they can be either in a clean state or evicted from a page cache. Unfortunately, it is overly intricate to keep track of such data blocks, but if they are missing, the semantics of the `fsync` system call can be violated.

We address this challenge by specifying *whole* data blocks of a file. This approximation is made efficient by fundamental file-system design principles; most file systems allocate data blocks for a file as consecutively as possible so as to benefit from spatial locality [14]. This idea has been adopted to reduce the seek time for HDDs, but it holds true for SSDs as well since a high degree of spatial locality means better performance for SSDs because it allows for more efficient address translation and interleaving over multiple channels/chips in the SSD. With this policy, the data blocks of a file are likely to be encoded by only a few extents, which means only a small number of `RFLUSH` commands are needed.

However, this might not always be the case because there could be more fragmentations over time, in particular for larger files. To address this, our final design choice is to transfer the *inode number* of the target file, instead of a set of LBAs. This approach can faithfully preserve the `fsync` semantics, without excessive overhead needed to specify the range of data blocks to persist with RFLUSH. The implementation details of the inode-based RFLUSH protocol will be described in Section 3.4.

3.3 How to Handle Metadata

One thing that must not be overlooked is to flush file system metadata that has a dependency on the target file of the `fsync`; otherwise, there is a danger of data corruption or loss on a system crash. We explain using the F2FS file system as an example. The on-disk layout of F2FS has two areas; *metadata area* and *main area*. The metadata area keeps information for file system maintenance such as block allocation bitmaps and orphan inode lists [22]. In contrast, the main area is used to store normal data blocks and file metadata including inode and indirect blocks. Upon a write request, a set of blocks needs to be updated in an atomic manner to provide crash consistency [3]. Specifically, since F2FS is a log-structured file system, it allocates and updates a new data block out-of-place, requiring the updating of related metadata (i.e., inode) and indirect blocks to properly point to the new block. In turn, the block allocation bitmap and several tables that maintain information for space management should also be updated. This behavior leads to many small random writes to blocks containing the file system metadata; encoding of those writes as a set of ranges would be complicated. To get away with this complication, we decide to encode a full range of the metadata area, which is a superset of metadata to be updated, and send it along with the RFLUSH command.

This approximation seems to have a problem when `fsync` requests from multiple files are interfered with each other because their metadata shares a single LBA. Consider a case in which there are two different files *A* and *B*, whose inode structures are located in a single block. When the `fsync` requests occur for the files concurrently, forcing the entire metadata area by one `fsync` request might corrupt data integrity, violating ordering constraints between data and metadata of another file (e.g., file *B*'s metadata is persisted before file B's data block).

However, this is not the case because current file systems are carefully designed so as not to let this happen. For example, F2FS logs individual inode structure on an update, instead of an entire block, thereby preventing undesired interference that can be caused by interleaved `fsyncs`. Ext4 resolves this issue by forcing all dependant data prior to persisting the modified metadata block. Thus, in the above example, both data A and B are flushed to non-volatile storage before the metadata block when an `fsync` request occurs either for file A or B.

3.4 How to Integrate into a Storage Protocol

To make use of the RFLUSH primitive, the host interface should be extended. While this extension is difficult to be incorporated into mature storage interfaces such as SATA [12] or SAS [17], it is a viable option for emerging storage interfaces like NVMe [10, 16] to add proprietary extensions. Another possibility for incorporating extensions into the standard storage API is to use the open-channel SSD architecture [4, 23]. In this architecture, the host system implements many of the functionalities needed to manage flash memory (e.g., garbage collection). Also, by utilizing veiled information behind the storage device interface, this architecture enables the management of flash memory to meet the demands of the host system. We use the latter approach since the host-manageable architecture allows easy integration of the extensions for RFLUSH.

Our prototyping system implements the inode-based RFLUSH protocol through storage interface extension and F2FS file system modification. We add the range flush protocol to BlueDBM, which is an open-channel flash development platform from MIT [23], facilitating the construction of a host-manageable storage device. Specifically, we extend the host storage interface to support the RFLUSH primitive in which the inode number is encoded. Then, we augment the in-storage buffer handler in the FTL to locate the associated data blocks and flush them selectively upon an RFLUSH request. The buffer handler maintains the pending updates in a hash table using an inode number as a key. Note that this mechanism requires a write command that also includes an inode number such that the device controller determines which file the data block belongs to. However, the open-channel SSD half of which the FTL runs on the host side can easily determine this by referencing the kernel data structure with the transferred write request, which is used in our implementation.

On the host side, F2FS, the modified file system, communicates with BlueDBM through a block device interface and makes use of the RFLUSH primitive in implementing the `fsync` system call. When an `fsync` request arrives from the application, F2FS writes all dirty pages of the requested file and the associated metadata from a page cache to a storage device. Then, F2FS issues a pair of RFLUSH commands that include the inode numbers associated with the target file and the metadata area. The RFLUSH command is forwarded to the storage device controller through the underlying block I/O layer and device driver where a host side component of BlueDBM runs. BlueDBM completes the RFLUSH request by forcing writes associated with the given inode number.

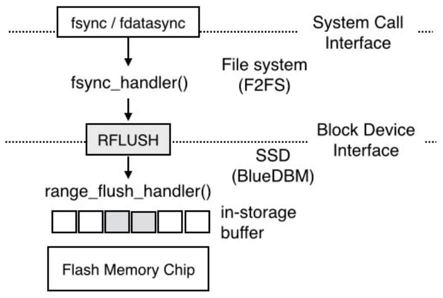

Figure 1: RFLUSH **protocol implementation.**

Configuration	Settings
Page / Block size	4KB / 64 Pages
Read / Write Latency	100us / 1300us
Block Erase Latency	1.5ms
Data Transfer Latency	100us (for 4KB)
Overprovisioning Ratio	3%
SSD capacity	37 GB
In-storage Buffer	256 MB / 1 GB

Table 1: **SSD platform setup.**

4 Performance Evaluation

We evaluate the proposed RFLUSH using a prototype implementation. The next section explains our evaluation methodology. In §4.2, we report results on the effectiveness of RFLUSH from experiments using both micro- and macro-benchmarks.

4.1 Methodology

We modified both the file system (F2FS) [22] and the storage device (BlueDBM) [23] to implement the RFLUSH protocol in Linux 4.7.2. Figure 1 shows the architecture of our experimental platform. When the user issues an fsync request through the system call interface, the sync_handler module inside the file system generates RFLUSH commands to BlueDBM. The range_flush_handler module within the FTL of BlueDBM handles the request by forcing the associated data from its volatile buffer to the non-volatile media.

Our experiments were performed on Intel Core i7 running at 3.3GHz with 64GB of DDR4 memory. The detailed configurations of BlueDBM are given in Table 1. To understand the performance consequence of the RFLUSH primitive, we first evaluate the prototype using a micro-benchmark based on FIO [11], which generates a synthetic workload that models a best-case scenario for RFLUSH. Then, we use a set of macro-benchmarks to examine the effectiveness of RFLUSH in a real environment. In our experiments, the storage device is accessed in a

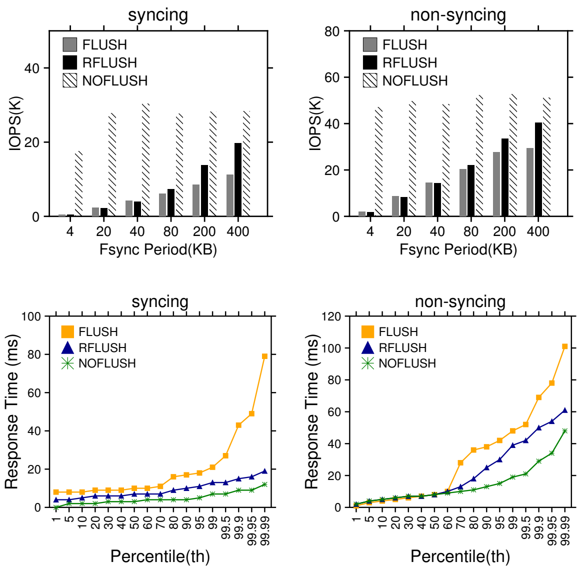

Figure 2: **IOPS and response time distributions of the micro-benchmark.** *Figures in the top row show IOPS for syncing and non-syncing threads with a 1GB storage buffer. The X-axis is the amount of data written between the invocations of fsync. The use of* RFLUSH *improves IOPS by up to 1.74x and 1.36x for the syncing and non-syncing threads, respectively, compared to using* FLUSH*. The two graphs in the bottom row give a percentile response time for both the syncing and non-syncing threads when fsync period is 400KB. The 99.99th percentile response time is reduced from 79.36us to 19.38us when* RFLUSH *is used instead of* FLUSH *in a syncting thread.*

Benchmark	Write #	Avg. Size	fsync Interval / #
Fileserver	1536K	1MB	None
TPC-C	2.2K	16KB	21373us / 13448
Linkbench	101K	16KB	10016us / 14097

Table 2: **Macro-benchmark characteristics.**

direct mode (unless otherwise specified) to observe the behavior of RFLUSH more clearly in a controlled environment. The performance is measured five times for each scenario and their median is reported.

4.2 Experimental Results

Micro-Benchmark: To assess the potential performance gain made possible by RFLUSH, we used a micro-benchmark based on FIO [11] that approximates a typical scenario where there is a mixture of asynchronous and synchronous writes. The micro-benchmark consists of both *syncing* and *non-syncing* threads. Both types of thread perform the same task except for their syncing behavior. Both write 2GB data randomly to a file with a 4KB granularity in a direct mode. The difference is a syncing thread issues an fsync request after writing a

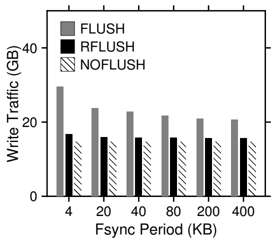

Figure 3: **Write traffic from the micro-benchmark.** *The write traffic is measured at the interface between the in-storage buffer and the flash memory when the buffer size is 1GB.* `RFLUSH` *reduces write traffic by 24% to 43% for the fsync periods we considered.*

given amount of data.

In the experiment, there were one syncing thread and 12 non-syncing threads, and we measured their performances for three possible configurations: `FLUSH`, `RFLUSH`, and `NOFLUSH`. The `FLUSH` configuration forces to flash memory all data in the volatile buffer of the storage device, while the `RFLUSH` configuration forces only the data in a given LBA range. In the `NOFLUSH` configuration, the storage device ignores all the sync requests. In all configurations, if the number of dirty pages in the buffer is above a threshold (90% here), a certain number of pages are written-back to flash memory by a background activity in the storage device. Figure 2 shows the performance of both the syncing and non-syncing threads in terms of IOPS and response time. In the figures of the top row, the X-axis is the amount of data written before the syncing thread issues an `fsync` request.

The results show that there is a large performance improvement for the syncing thread when `RFLUSH` is used instead of `FLUSH`. This performance improvement is mainly due to the fact that the flushing activities of the syncing thread are not interfered by the flushing of non-urgent writes from non-syncing threads when `RFLUSH` is used. For the same reason, `RFLUSH` also eliminates a long tail in the response time distribution for the syncing thread, which is critical to providing a consistent performance from a storage device.

The results also show that even the performance of non-syncing threads is improved. When `RFLUSH` is used, a prioritized flushing of data written by the syncing thread gives more time for the dirty data from non-syncing threads to reside in the buffer. The increased time in the buffer allows them to absorb more writes to the same LBA and also to be coalesced more with other writes, resulting in a better performance. As a result, `RFLUSH` reduces the write traffic significantly compared to `FLUSH` as Figure 3 illustrates. Its result even comes close to that of `NOFLUSH`. In this scenario, each three of the 12 non-syncing threads access the same file, while a

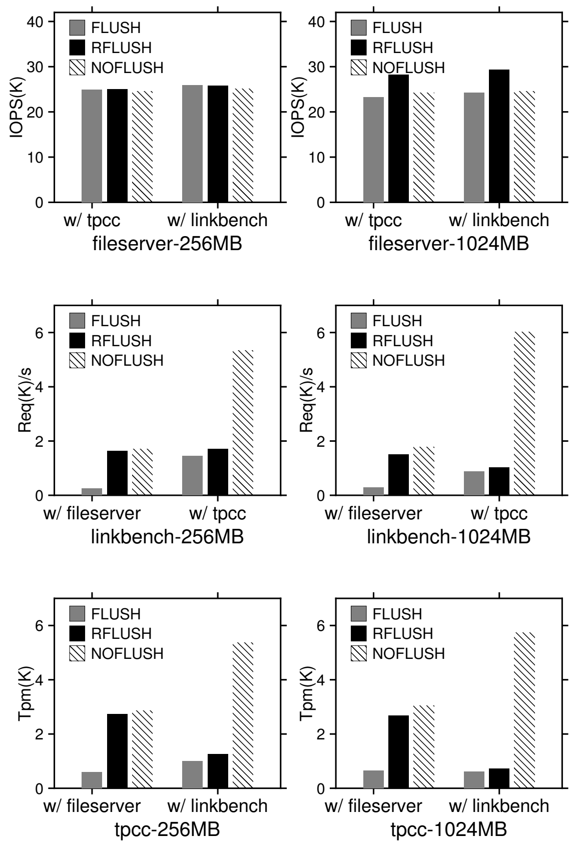

Figure 4: **Performance of mixed real workloads in a direct mode.** *These figures show IOPS for each pair of benchmarks when a storage buffer size is 256MB and 1024MB. TPC-C and Linkbench achieves 5.3x to 6.5x and 4.1x to 4.5x higher IOPS with* `RFLUSH` *when running together with Fileserver. Fileserver also delivers 20% higher IOPS with* `RFLUSH` *when executing with TPC-C and Linkbench. When TPC-C and Linkbench are mixed, their performances are improved by 1.4x to 1.6x and 1.17x to 1.3x, respectively.*

syncing thread accesses its own file. Thus, the writes of the non-syncing thread have a locality. F2FS basically updates the data in an out-of-place manner, but it allows overwrite once the data is copied for updates after the last checkpoint, unless the explicit `fsync` request occurs. Therefore, F2FS benefits from the enhanced buffering effect of the `RFLUSH` primitive in the writes of non-syncing threads.

A somewhat non-intuitive result is that when the `fsync` requests are issued too frequently, in some extreme cases `RFLUSH` even performs worse than `FLUSH` even though the former results in much less write traffic to the storage device. Careful analysis over the results reveals that if `fsyncs` are too frequent, the performance is dominated by `fsyncs` rather than the actual write traffic associated with them.

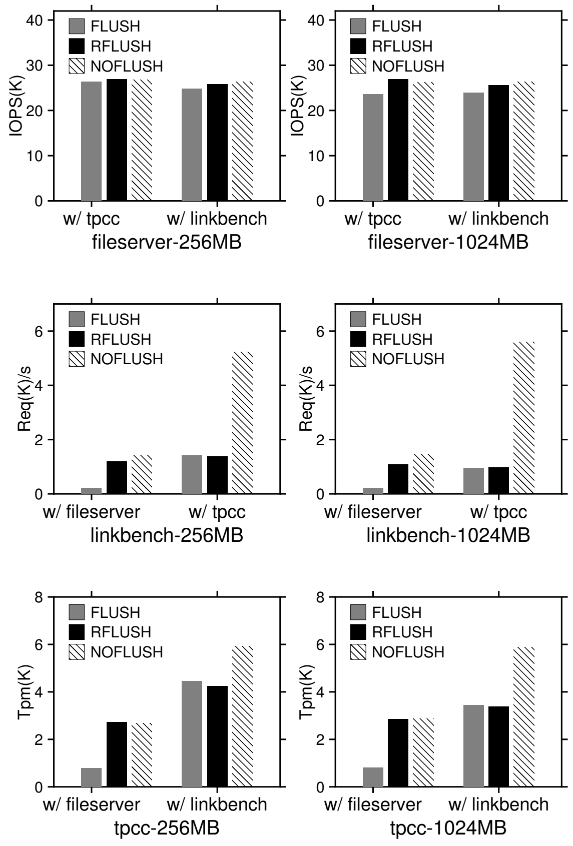

Figure 5: **Performance of mixed real workloads in a buffered mode.** *These figures report the performance with the page cache turned on. Although the absolute values are different, the results show the same general trends as in a direct mode (cf. Figure 4).*

Macro-Benchmarks: To assess the performance impact of RFLUSH in the real world, we selected three macro-benchmarks (Fileserver, Linkbench, and TPC-C) and measured their performances when a pair of them run concurrently. Fileserver generates a large number of asynchronous writes acting like a multi-streaming server [40]. Linkbench is a graph processing application based on the Facebook Social Graph, containing a few kilobytes of writes with frequent sync requests [2]. TPC-C is an on-line transaction processing benchmark which issues small-sized random writes with frequent synchronization [42]. Table 2 summarizes various statistics about the three macro-benchmarks.

Figure 4 shows the results in terms of IOPS for each pair of the three macro-benchmarks. The results show that the performance improvement by RFLUSH is most noticeable when asynchronous and synchronous workloads are mixed, as in the micro-benchmark we considered in the previous section. For example, TPC-C and Linkbench show 4.5x and 6.5x higher IOPS with RFLUSH, when they run together with Fileserver, which is consistent with the micro-benchmark results in the previous section.

The results also show that there are performance improvements even in the case where both benchmarks contain synchronous workloads. For example, when TPC-C and Linkbench are running together, RFLUSH improves performance by up to 1.4x and 1.29x in TPC-C and Linkbench, respectively. This result is due to time-multiplexed non-volatile materializations for `fsyncs` from the two benchmarks. One counter-intuitive observation is that an RFLUSH outperforms a NOFLUSH in a mixture of Fileserver and TPC-C/Linkbench with a 1024MB buffer. This improvement comes from that an RFLUSH replenishes free space more quickly by proactively writing back the buffered data on a synchronization request, which helps the efficent handling of the bulky writes generated from Fileserver.

We also performed the same experiments in a buffered mode (i.e., with the page cache turned on). Figure 5 reports the performance in the same format as in Figure 4. Although the absolute values are different, the results show the same general trends as in a direct mode shown in Figure 4. The performance gap between RFLUSH and FLUSH is reduced because of periodic flushing from the page cache but the difference is only marginal.

5 Conclusion

In this paper, we raised an issue about the negative performance impact of a lump-sum approach to persisting buffered data within a storage device and presented RFLUSH that allows a fine-grained persistence control. We implemented an RFLUSH prototype by modifying a file system (F2FS) in Linux 4.7.2 as well as a storage device based upon an open-source flash development platform. Performance evaluation using the prototype shows that RFLUSH increases overall I/O performance by up to 6.5x, and eliminates a long tail latency of synchronous writes.

6 Acknowledgments

We thank Ming Zhao (our shepherd) and the anonymous reviewers for their insightful comments. This work was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (No. 2017R1D1A1B03031494) and by the Ministry of Science, ICT & Future Planning (No. 2014R1A1A3053505 and No. NRF-2017R1E1A1A01077410).

References

[1] Anand, A., Sen, S., Krioukov, A., Popovici, F., Akella, A., Arpaci-Dusseau, A., Arpaci-Dusseau, R., and Banerjee, S. Avoiding file system micromanagement with range writes. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (2008), OSDI, USENIX Association, pp. 161–176.

[2] Armstrong, T. G., Ponnekanti, V., Borthakur, D., and Callaghan, M. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), ICMD, ACM, pp. 1185–1196.

[3] Arpaci-Dusseau, R. H., and Arpaci-Dusseau, A. C. *Three Easy Pieces*. Arpaci-Dusseau Books, 2015.

[4] Bjørling, M., González, J., and Bonnet, P. Lightnvm: The linux open-channel ssd subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies* (2017), FAST, pp. 359–374.

[5] Chen, P. M., Ng, W. T., Chandra, S., Aycock, C., Rajamani, G., and Lowell, D. The rio file cache: Surviving operating system crashes. *Acm Sigplan Notices 31*, 9 (1996), 74–83.

[6] Chidambaram, V., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Optimistic crash consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (2013), SOSP, ACM, pp. 228–243.

[7] Coburn, J., Bunker, T., Schwarz, M., Gupta, R., and Swanson, S. From aries to mars: Transaction support for next-generation, solid-state drives. In *Proceedings of the 24th ACM symposium on operating systems principles* (2013), SOSP, ACM, pp. 197–212.

[8] Dorgelo. Host memory buffer based ssd systems. Flash Memory Summi, 2015.

[9] Dubitzky, Z., Gold, I., Henis, E., Satran, J., and Sheinwald, D. Dsf: Data sharing facility. *Technical report* (2002).

[10] Eshghi, K., and Micheloni, R. Ssd architecture and pci express interface. In *Inside Solid State Drives (SSDs)*. Springer, 2013, pp. 19–45.

[11] FIO. Fio benchmark. https://github.com/axboe/fio.git, 2017.

[12] Grimsrud, K., and Smith, H. *Serial ATA Storage Architecture and Applications: Designing High-Performance, Cost-Effective I/O Solutions*. Intel press, 2003.

[13] Grupp, L. M., Davis, J. D., and Swanson, S. The harey tortoise: Managing heterogeneous write performance in ssds. In *USENIX Annual Technical Conference* (2013), ATC, pp. 79–90.

[14] He, J., Nguyen, D., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Reducing File System Tail Latencies with Chopper. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (Santa Clara, CA, February 2015), FAST.

[15] Huang, S.-M., and Chang, L.-P. Exploiting page correlations for write buffering in page-mapping multichannel ssds. *ACM Transactions on Embedded Computing Systems 15*, 1 (2016), 12.

[16] HUFFMAN, A. Nvm express: Going mainstream and whats next. Intel Developers Forum, 2014.

[17] Jackson, M. *SAS Storage Architecture*. MindShare Press, 2005.

[18] Kang, J.-U., Hyun, J., Maeng, H., and Cho, S. The multi-streamed solid-state drive. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage and File Systems* (Philadelphia, PA, 2014), HotStorage.

[19] Kim, H., and Ahn, S. Bplru: A buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (2008), FAST, pp. 1–14.

[20] Kim, M., Lee, J., Lee, S., Park, J., and Kim, J. Improving performance and lifetime of large-page nand storages using erase-free subpage programming. In *Proceedings of the 54th Annual Design Automation Conference 2017* (2017), DAC, ACM, p. 24.

[21] LAPEDUS, M. Sorting out next-gen memory. http://semiengineering.com/sorting-out-next-gen-memory/, 2016.

[22] Lee, C., Sim, D., Hwang, J. Y., and Cho, S. F2fs: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (2015), FAST, pp. 273–286.

[23] Lee, S., Liu, M., Jun, S. W., Xu, S., Kim, J., and Arvind, A. Application-managed flash. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies* (2017), FAST, pp. 339–353.

[24] Marks, K. An nvm express tutorial. Flash Memory Summit, 2013.

[25] Marvell. Conservative use of dram. http://www.anandtech.com/show/9942/marvell-implements-host-memory-buffer-for-dramless-88nv1140-ssd-controller, 2016.

[26] Mesnier, M., Chen, F., Luo, T., and Akers, J. B. Differentiated storage services. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011), SOSP, ACM, pp. 57–70.

[27] Min, C., Kang, W.-H., Kim, T., Lee, S.-W., and Eom, Y. I. Lightweight application-level crash consistency on transactional flash storage. In *USENIX Annual Technical Conference* (2015), ATC, pp. 221–234.

[28] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems 17*, 1 (Mar. 1992), 94–162.

[29] Nam, E. H., Kim, B. S. J., Eom, H., and Min, S. L. Ozone (o3): An out-of-order flash memory controller architecture. *IEEE Transactions on Computers 60*, 5 (2011), 653–666.

[30] Nightingale, E. B., Veeraraghavan, K., Chen, P. M., and Flinn, J. Rethink the sync. *ACM Transactions on Computer Systems (TOCS) 26*, 3 (2008), 6.

[31] Ouyang, X., Nellans, D., Wipfel, R., Flynn, D., and Panda, D. K. Beyond block i/o: Rethinking traditional storage primitives. In *The 17th IEEE International Symposium on High Performance Computer Architecture* (2011), HPCA, IEEE, pp. 301–311.

[32] Prabhakaran, V., Rodeheffer, T. L., and Zhou, L. Transactional flash. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (2008), OSDI, pp. 147–160.

[33] Samsung. Samsung ssd 850 pro. http://www.samsung.com, 2016.

[34] Sandisk. Sandisk extreme pro ssd. http://www.anandtech.com/show/8170/sandisk-extreme-pro-240gb-480gb-960gb-review, 2016.

[35] Seshadri, S., Gahagan, M., Bhaskaran, M. S., Bunker, T., De, A., Jin, Y., Liu, Y., and Swanson, S. Willow: A user-programmable ssd. In *Proceedings of the 11th USENIX conference on Operating systems design and implementation* (2014), OSDI, pp. 67–80.

[36] SHU, F., AND OBR, N. Data set management commands proposal for ata8-acs2, revision 1 ed. *Microsoft Corporation, One Microsoft Way, Redmond, WA* (2012), 98052–6399.

[37] SKHYNIX. Sk hynix se3010 enterprise ssd review. http://www.tomsitpro.com/articles/sk-hynix-se3010-enterprise-ssd-review,2-977-2.html, 2016.

[38] SOLWORTH, J. A., AND ORJI, C. U. Write-only disk caches. *ACM SIGMOD Record 19*, 2 (1990), 123–132.

[39] STEIGERWALD, M. Imposing order. *Linux Magazine, May* (2007).

[40] TARASOV, V., ZADOK, E., AND SHEPLER, S. Filebench: A flexible framework for file system benchmarking. *USENIX; login 41* (2016).

[41] THERESKA, E., BALLANI, H., O'SHEA, G., KARAGIANNIS, T., ROWSTRON, A., TALPEY, T., BLACK, R., AND ZHU, T. Ioflow: a software-defined storage architecture. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (2013), SOSP, ACM, pp. 182–196.

[42] TPCC. Tpcc-mysql benchmark. https://github.com/Percona-Lab/tpcc-mysql, 2017.

[43] WANG, A.-I., REIHER, P. L., POPEK, G. J., AND KUENNING, G. H. Conquest: Better performance through a disk/persistent-ram hybrid file system. In *USENIX Annual Technical Conference* (2002), ATC, pp. 15–28.

[44] XIANGFENG, L. IO Pattern based Optimization in SSD. Flash Memory Summit, 2016.

[45] YANG, S., HARTER, T., AGRAWAL, N., KOWSALYA, S. S., KRISHNAMURTHY, A., AL-KISWANY, S., KAUSHIK, R. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Split-level i/o scheduling. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles* (2015), SOSP, ACM, pp. 474–489.

[46] ZHANG, Y., ARULRAJ, L. P., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. De-indirection for flash-based ssds with nameless writes. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (2012), FAST, pp. 1–16.