Untie a Knot in a Last Stage Buffer

Jeseong Yeon*, Minseong Jeong*, Sungjin Lee†, Eunji Lee*‡
*Chungbuk National University, †Inha University, ‡University of Wisconsin-madison
https://oslab.cbnu.ac.kr, sunjin.lee@inha.ac.kr, eunji@cbnu.ac.kr

Historically, storage medium has used a small amount of RAM as a disk buffer to mask its poor random performance and limited endurance. Because a volatile buffer, however, can bring with it data loss and improper ordering of updates in a crash, a *flush* command has been introduced as a storage interface that forces device to immediately commit any pending writes to storage. This mechanism sacrifices performance by clearing an entire buffer upon a flush, whereas it is commonly issued with less stringent requirements; however the overhead is affordable because the buffer size is limited.

However, the cost of flushing is increasing significantly as latest SSDs are attempting to deploy a larger buffer to compensate for their continuously decreasing latency and endurance. For example, 1TB SSDs are employing 512MB to 2GB RAM as a disk buffer, while some manufacturers are exploring ways of using host memory as a storage buffer, instead of incorporating a large RAM within storage [1]. In either case, the conventional flushing mechanism yields a more painful impact, considerably forfeiting the possibility of buffering or coalescing I/Os in the buffer. To better understand this, we run a simple experiment where four threads write 1MB data on a file at 4KB granularity, respectively, one of them issuing an `fsync` call every 100KB writes. From Fig. 1, we can see that the flushing limits performance significantly as a buffer size increases; it also causes long tail latency in response time due to the varying amount of writes across the runs.

This paper addresses this challenge with a new storage primitive called *range flush*, which transfers additional information on which data are associated with a flush command. This primitive enables underlying devices to identify and flush the *associated data only*, eliminating avoidable writes and relaxing the constraints in I/O processing. The benefit of range flush seems straightforward, but realizing it effective in an I/O system, from host to storage, is not without challenge. We demonstrate the effectiveness of range flush by

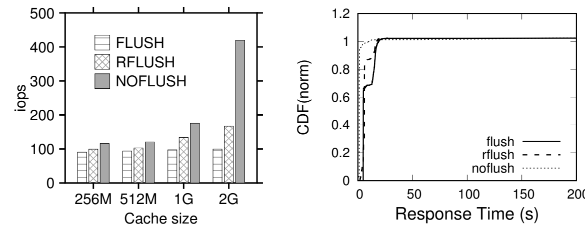


Figure 1: **IOPS and Response Time.**

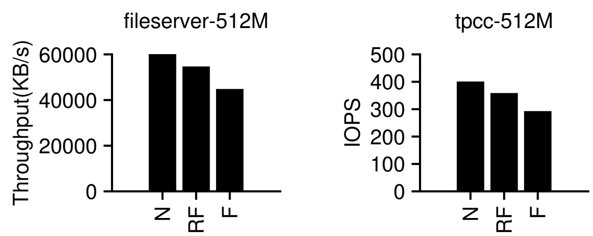


Figure 2: **Fileserver and TPC-C Performance.**

implementing its protocol in F2FS and Linux storage stacks. Specifically, we implement an `fsync` handling module in a file system to make use of an `RFLUSH` (range flush) command instead of `FLUSH`. The `FLUSH` command delivers inefficiency because it flushes the entire buffer while the semantic of `fsync` persists specific file data only. To avoid this, `RFLUSH` transfers a file inode number along with the command, such that underlying storage can obtain the LBAs of the affected data by referencing the inode data structure. Note that we ensure that the storage flushes the associated metadata in a tandem, otherwise the system will end up with corruption in a crash. The storage protocol of `RFLUSH` is implemented in an open-channel SSD development platform [1]. We measure the performance by running two heterogeneous workloads concurrently: fileserver (large asynchronous writes) and TPC-C (small synchronous writes). Fig. 2 shows `RFLUSH` achieves 1.23x and 1.22x better performances than `FLUSH` the two workloads when a buffer size is 512MB.

Acknowledgments

This work was supported by the Ministry of Science, ICT & Future Planning (No. NRF-2014R1A1A3053505).

References

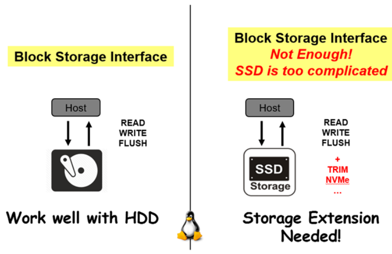
[1] M.-C. Chen, "Host Memory Buffer (HMB) based SSD System", *Flash Memory Summit*, 2015.

[1] https://github.com/chamdoo/bluedbm

Untie A Knot in a Last Stage Buffer

Jeseong Yeon[1], Minseong Jeong[1], Sungjin Lee[2], Eunji Lee[1,3]
[1] Chungbuk National University, [2] Inha University, [3] University of Wisconsin-madison

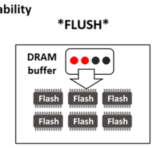
Storage Interfaces are Evolving!



What's the matter with FLUSH?

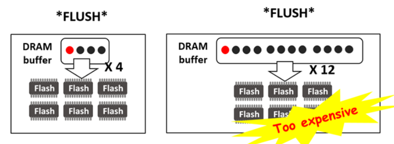
- Volatile buffer can bring about data loss and improper ordering in a crash
- FLUSH : enforce any pending writes to be persisted immediately
- (+) Control over ordering and durability
- (-) Decrease performance
  - Unnecessary writes
  - Less possibility of coalescing and scheduling

Affordable in a small buffer!



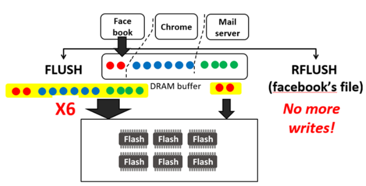
On-disk buffer is getting bigger!

- SSD are attempting to deploy more larger buffer to compensate for decreasing latency
- 16 to 32MB → 512MB to 2GB
- DRAM-less SSD / Host Memory Buffer



Range Flush : Untie a Knot in Last Stage

- RFLUSH: Transfer information on "which data are associated with the command" to storage

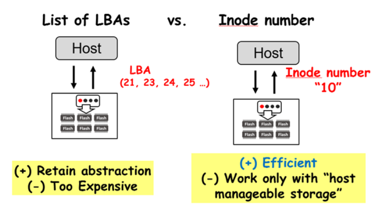


Challenges for Getting Real (1/3)

- Q1: Which data should be specified with respect to an RFLUSH primitive
- Two sources of FLUSH
  - From user applications via fsync()
  - From file system
- RFLUSH matches well with user's request!
  - Require persistency of specific file data in general
- File system uses FLUSH when needing to orchestrate ordering and durability of system-wide writes
  - E.g., commit or checkpoint transactions

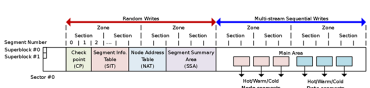
Challenges for Getting Real (2/3)

- Q2: How to communicate between host and storage ?



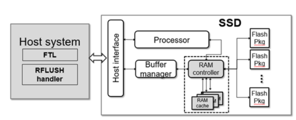
Challenges for Getting Real (3/3)

- Q3: How to handle dependent metadata?
- Ensure that associate metadata are made durable together to avoid corruption in a crash
- Case study with "F2FS"
  - Storage flushes "metadata zone" on a RFLUSH command

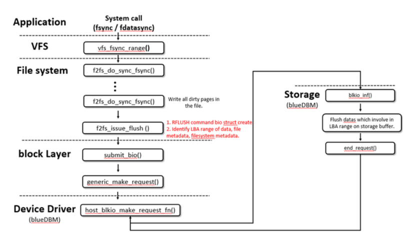


Implementation in Linux

- Linux 4.7.2
- F2FS file system
- BlueDBM : open-channel SSD developed by MIT

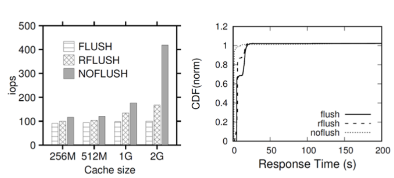


RFLUSH Implementation Overview

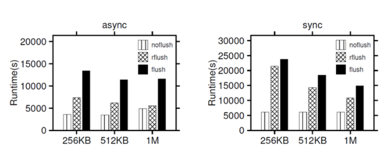


Performance Evaluation

- FIO Benchmark

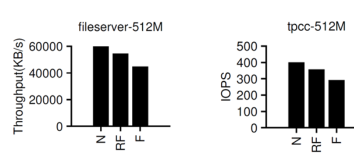


RFLUSH improves iops by 32% compared to FLUSH and reduces long tail latency



RFLUSH improves provides performance isolation for asynchronous writes

- Fileserver and TPC-C



RFLUSH delivers 1.22x and 1.23 better performance than FLUSH