Buffered I/O support for Zoned Namespace SSD

Youngjae Lee, Jeeyoon Jung, Dongkun Shin
Department of Electrical and Computer Engineering, Sungkyunkwan University
yjlee4154@gmail.com, {wjdwldbs1, dongkun}@skku.edu

Abstract

Recently, NVMe Zoned Namespace (ZNS) interface has been proposed to reduce the NAND flash management overheads of solid-state drive by matching the host interface to internal hardware characteristics. For the Zoned Namespace SSD, writes must be performed using direct I/O to ensure the sequential write order, which may degrade the performance of user applications in many cases. In this paper, we introduce buffered write support for ZNS SSD in Linux kernel I/O stack and implement LSM-tree-based key-value store for ZNS SSD to analyze the effect of buffered writes. Our evaluation shows that buffered writes can reduce the total amount of I/O and increase overall performance of applications utilizing ZNS SSD.

Keywords: ZNS, SSD, key-value store

1. Introduction

As flash memory technology has been developed and prices have decreased, solid state drives (SSDs) are replacing hard disk drives. SSD is widely adopted from consumer electronics such as smartphones and laptops to high-performance data center servers, due to their advantages such as low power consumption, high throughput and random IOPS. However, SSD periodically performs garbage collection (GC) internally to provide a block interface, which causes unpredictable performance and wears out the NAND lifespan [1].

The recently proposed NVMe zoned namespace (ZNS) interface can alleviate internal GC by exposing the write characteristics of NAND flash memory to the host [2]. However, as ZNS provides only sequential write interface, the existing file system cannot be used, and a dedicated file system must be used, or the user application must implement its own file system.

Filesystems that use out-of-place updates such as F2FS and Btrfs currently support zoned block devices [3, 4]. In addition, LSM-tree-based key-value store is a suitable workload to utilize ZNS SSD at the application-level by performing update using only sequential writes [5].

If ZNS SSD is used directly in application-level, user must use direct I/O to ensure write order [6]. Since data is written to the storage by bypassing the page cache, even recently written data should be read from the storage device, and thus degrade I/O performance in most cases. This can be solved by implementing a cache at the user-level, but user data is often not block aligned and implementing a user-level cache with good performance is complicated.

In this study, we modify the Linux kernel's I/O stack to support buffered write for ZNS SSD and implement the LSM-tree-based key-value store to evaluate the effectiveness of ZNS and buffered I/O.

2. Background

2.1. Zoned Namespace SSD

The NVMe zoned namespace can simplify the SSD internals and provide predictable performance by matching the interface to the characteristics of the NAND flash and eliminates internal GC operations. Unlike the existing Open-channel SSD that had to manage all hardware characteristics such as wear-leveling of NAND in the host, ZNS is more abstraction and exposes only sequential write characteristics to the host. Unlike the Open-channel SSD, ZNS exposes only write and erase characteristics of NAND flash to the host, and hardware characteristics such as wear-leveling or bad-block management are managed by the device, reducing the burden on the host.

The ZNS interface uses the logical block addressing as in the block interface, but LBA is divided into fixed size zones, and writes must be performed in a sequential order within each zone. To reuse a zone, user must explicitly reset the zone to erase the contents and make it writable state.

2.2. LSM-tree

Unlike the existing index data structures such as B-tree, LSM-tree is a write-optimized data structure by batching data and writing in a sequential write pattern. LSM-tree-based key-value store collects a certain amount of new key-value records in MemTable, a memory data structure and periodically flushes to disk in the form of SSTable.

Key-value records are sorted in ascending order within a single SSTable and organized in several levels. To search for an arbitrary record, all SSTables with a key range including the target key must be searched. LSM-tree periodically performs compaction in the background to maintain its structure and optimize read performance. The background compaction removes obsolete records and rearranges the records to create new SSTables.

3. Buffered I/O stack for ZNS

Linux buffered I/O uses the page cache to accelerate storage access, by caching recently accessed data in DRAM, thus data can be accessed without incurring storage I/O. When the user calls the write system call, the Linux kernel copies data to the page cache and marks it as dirty, and the dirty pages are written back later in the background.

Since writeback is not guaranteed to be performed sequentially for each zone, buffered write cannot be used for ZNS SSDs and direct I/O must be used [6]. However, if direct I/O is used, accessing recently written data should be performed from the storage, thus resulting in high latency and waste of the SSD bandwidth. To solve this problem, we have modified Linux kernel I/O stack as follows.

Page cache writeback. The writeback of the dirty pages is performed in sequential order for the LBA within each zone. Additionally, when the user write request updates the already written LBA, an error occurs to prevent overwrite problem.

When making a zone into the finish state, the dirty page writeback for the zone should precede the issue of zone finish command. Therefore, when user finishes zone via ioctl system call, zone finish command is sent to the device after all dirty pages in the corresponding zone are written back.

I/O scheduler. While page cache writeback is performed in sequential order, inversion may still occur in request insertion order if synchronous writes from users are performed at the same time. Currently, Linux deadline I/O scheduler uses per-zone locking for zoned block devices to serialize writes to ensure only one ongoing write I/O is exists for each zone [6]. However, the write command can be delivered to the device in the wrong order as it does not guarantee command ordering.

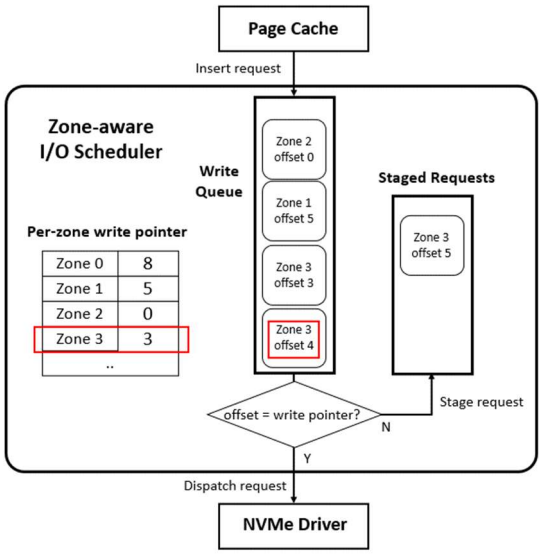


Figure 1. Zone-aware I/O scheduler

Therefore, we added a zone-aware scheduler to guarantee write order by managing write pointers in the scheduler. Figure 1 shows the overall architecture of I/O scheduler. If the target LBA of the dispatch candidate in the write queue does not match the offset of the write pointer table of the current zone, the request is put into a staged request queue. When the write request is completed, the write pointer is incremented to point to the next write offset, and when user resets zone, the write pointer is initialized to zero.

In the example of Figure 1, the first request of the write queue has an offset of 4, but the write pointer of the current zone 3 is 3, so the request is switched to the stage state. The next request will be dispatched since its offset matches to the writer pointer, and after the write pointer is advanced, the requests in the stage queue will be dispatched.

4. Key-value store

We have implemented LevelDB-ZNS, an LSM-tree-based key-value store for ZNS SSD, based on LevelDB [7]. LevelDB-ZNS opens a raw block device to perform I/O using read/write system calls and performs zone management using libzbd [8].

LevelDB-ZNS uses a simple user-level file system, which supports the minimum operations required for LevelDB such as create, append write, and delete. It also leverages the fact that SSTable and log files are created in almost fixed size, and thus allocates one zone for each file. Our ZNS SSD prototype provides conventional zones which allows random writes and LevelDB-ZNS uses them to store log files and file system metadata.

Figure 2 shows the layout of the LevelDB-ZNS file system. Zone 0 is used as a super block to store file system information and includes a metadata table

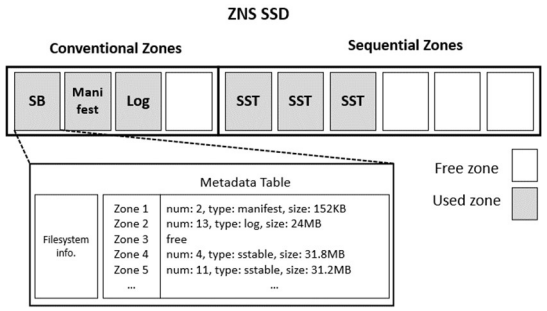


Figure 2. Layout of the LevelDB-ZNS filesystem

to store file size, number, and file type. Metadata is also maintained in-memory as a hash table so that metadata can be quickly referenced without I/Os. Whenever a file metadata is changed due to a file operation such as create, write, or close, the change is recorded in the metadata table.

For SSTable, a sequential zone is allocated since its data is written in units of a fixed-size buffer. For log and manifest files, which requires variable-sized append, the conventional zone is used.

When creating an SSTable, LevelDB uses fsync before closing the file to make the file persistent. In ZNS SSD, the zone must be finished to make zone inactive after write. Since our modified Linux kernel ensures the writeback of dirty pages when user finishes zone, we replaced fsync with zone finish.

When the SSTable is deleted, the allocated zone is reset then inserted into the free list for later reuse. We have also used fadvise to invalidate the page cache of closed file's zone to release memory resources.

5. Evaluation

5.1. Setup

We have implemented ZNS SSD prototype on the Cosmos+ OpenSSD[9] platform. The ZNS SSD prototype supports NVMe ZNS compliant commands and provides 32MB sized zones. Besides sequential write zones, a few conventional zones are also provided for metadata and log files. We also used block interface SSDs with page-mapping FTL for comparison.

For the host environment, a PC with an Intel core i7-4790 CPU and 16GB of DRAM is used. The operating system is Ubuntu 20.04 LTS, with the modified Linux kernel based on 5.10.

We used two versions of LevelDB-ZNS using direct and buffered writes. Both versions use buffered I/O for reads. For comparison, LevelDB's experiment using ext4 file system on block interface SSD was also performed. For comparison, the LevelDB was used using the ext4 file system on the block interface SSD. We evaluated the performance using db_bench and YCSB benchmarks [10].

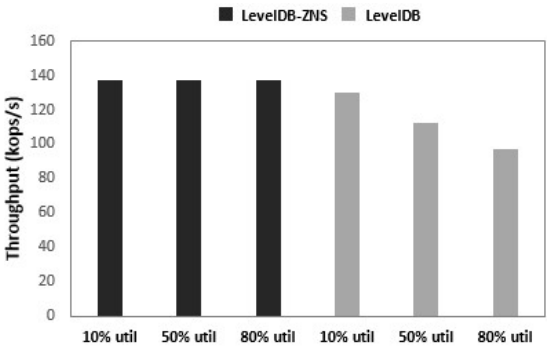


Figure 3. db_bench benchmark throughput varying NAND utilization

5.2. Comparison with block interface SSD

In order to evaluate the effect of ZNS, we compare the benchmark performance of block interface SSD and ZNS SSD. Figure 3 shows the results of db_bench fillrandom benchmark with 100M operations.

In cased of block SSD, performance is degraded as NAND utilization increases due to internal garbage collection. ZNS SSD shows constant performance regardless of NAND utilization by eliminating internal GC. In addition, even when NAND utilization is low, LevelDB-ZNS shows better performance than the LevelDB due to the removal of the file system overhead.

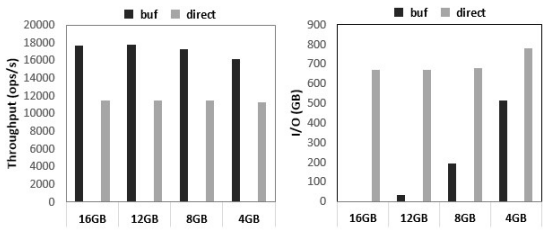
5.3. Comparison of Buffered and Direct I/O

For performance comparison, we use the load and workloads A to D of the YCSB benchmark. Both the number of KV records and the number of operations were set to 10M.

Figure 4 (a) shows the benchmark performance varying the memory limit in Load workload. In case of 16GB memory limit, the entire dataset is fit in memory. When buffered write is used, throughput increases as available memory increases. In the case of direct write, the performance did not increase much even if more memory was given.

Figure 4 (b) shows the total amount of read in YCSB load and workload A. In the case of buffered I/O, if DRAM is sufficient, recently written data is read from the page cache and thus save a lot of reads from the device. However, in the case of direct write, since all data including recently written data must be read from the device, the amount of read is amplified and SSD bandwidth is wasted.

Figure 5 shows the throughput of YCSB load and workload A to D with 8GB DRAM. The LevelDB with buffered write shows 1.5x to 2.38x higher performance for all workloads. In the case of load, only the SSTable for compaction input is read, so the performance difference is the relatively small. The performance difference is the smallest in workload C,



(a) YCSB Load – throughput (b) YCSB A – read I/O

Figure 4. YCSB benchmark performance

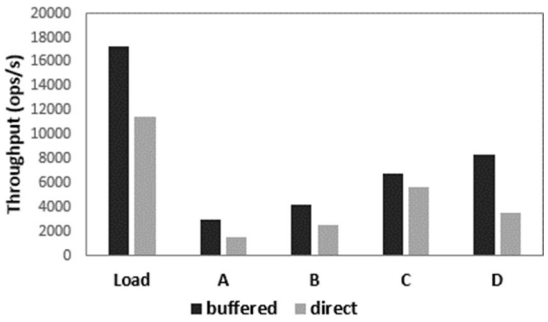


Figure 5. YCSB benchmark Throughput

which is a read-only workload. Workload D shows higher performance difference since page cache hit rate is high due to the workload characteristic of reading recently inserted records.

6. Conclusion

In this paper, we propose a buffered I/O stack for ZNS SSD and evaluate its effectiveness with our key-value store implementation for ZNS SSD. Our evaluation shows that using direct write causes a page cache miss for recently written data and thus degrades the performance of applications that depend on the page cache. We plan to design and optimize the user-level file system in key-value store for the ZNS SSD as a future work.

Acknowledgement

This work was supported by Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.IITP-2017-0-00914, Software Framework for Intelligent IoT Devices)

References

[1] Kim, Jaeho, et al. “Alleviating garbage collection interference through spatial separation in all flash arrays.” In 2019 USENIX Annual Technical Conference (USENIX ATC 19), 2019.

[2] Bjørling, Matias, et al. "ZNS: Avoiding the Block Interface Tax for Flash-based SSDs." Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC’21). 2021.

[3] Changman Lee, et al. “F2FS: A new file system for flash storage.” In 13th USENIX Conference on File and Storage Technologies (FAST’15), 2015.

[4] Ohad Rodeh, et al. “BTRFS: The linux B-tree filesystem.” ACM Transactions on Storage, 2013.

[5] Patrick O’Neil, et all. “The log-structured merge-tree (LSM-tree).” Acta Informatica. 1996.

[6] Zoned Namespaces (ZNS) SSDs. https://zonedstorage.io/introduction/zns/.

[7] Sanjay Ghemawat and Jeff Dean. LevelDB. http://code.google.com/p/leveldb, 2011.

[8] libzbd User Library. https://zonedstorage.io/projects/libzbd/

[9] Jaewook Kwak, et al. "Cosmos+ OpenSSD: Rapid prototype for flash storage systems." ACM Transactions on Storage (TOS) 16, 2020.

[10] Brian F Cooper, et al. “Benchmarking Cloud Serving Systems with YCSB.” In Proceedings of the 1st ACM Symposium on Cloud Computing, 2010.