QBLK: Towards Fully Exploiting the Parallelism of Open-Channel SSDs

Hongwei Qin, Dan Feng, Wei Tong*, Jingning Liu and Yutong Zhao
Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System
Engineering Research Center of data storage systems and Technology, Ministry of Education of China
School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
Email:{glqhw, dfeng, Tongwei, jnliu, ytzhao}@hust.edu.cn
*Corresponding author

***Abstract*—By exposing physical channels to host software, Open-Channel SSD shows great potential in future high performance storage systems. However, the existing scheme fails to achieve acceptable performance under heavy workloads. The main reasons reside not only in its single-buffer architecture, more importantly, but also in its line-based physical address management. Besides, the lock of address mapping table is also a performance burden under heavy workloads. We propose QBLK, an open source driver which tries to better exploit the parallelism of Open-Channel SSDs. Particularly, QBLK adopts four key techniques, namely (1) Multi-queue based buffering, (2) Per-channel based address management, (3) Lock-free address mapping, and (4) Fine-grained draining. Experimental results show that QBLK achieves up to 97.4% bandwidth improvement compared with the state-of-the-art PBLK scheme.**

***Index Terms*—NAND flash, Open-Channel SSD, Parallelism**

I. Introduction

As Open-Channel SSDs (OCSSDs) become more popular in data centers, various approaches has been adopted to enhance its performance. One of the most distinguished properties of Open-Channel SSDs is that they utilize the host CPU/memory capabilities to manage the Flash Translation Layer (FTL). Efforts have been taken to increase the performance of Open-Channel SSDs by cutting duplicated functions or providing predictable latencies. Although the idea of "Open-Channel" brings us lots of advantages, we are still facing some challenges where traditional SSDs do not meet. One of the challenges appears to be how to reduce conflicts among threads under heavy workloads.

Existing work lightNVM [1] provides us a systematic framework where host software can communicate with Open-Channel SSDs under the widely used NVMe protocol [2]. There are three ways for system designers to build an OCSSD storage system: (1)Implementing FTL inside a device driver and expose a block IO interface. This enables legacy applications or kernel modules to use OCSSDs without any changes. (2) Implementing FTL inside a file system and expose a POSIX interface. This enables system designers to cut some duplicated functions between file systems and FTL. (3) Implementing FTL functionalities inside user-space applications. This enables users to get predictable latencies and more application-specific control of the device. This paper mainly focus on how to achieve good performance under heavy workloads through the first IO path. However, many theories and schemes we propose in this paper can be used to the second and the third IO path of Open-Channel SSDs. We'd like to do it in the future.

We observe that current state-of-the-art Open-Channel SSD driver PBLK can not fully exploit the performance of Open-Channel SSDs. After digging into the detail implementation of PBLK, we find three in-efficient designs which affects performance.

First, PBLK uses one single ring buffer to cache or merge write requests. One kernel thread checks the ring buffer periodically to drain data into the physical device. This simplifies the design of translation map and write back functions. However, as the number of user threads grows, the ring buffer will become a performance burden. Second, the physical address management of PBLK is based on a logical structure called *line*. A *line* contains several blocks which can be accessed in parallel. As more parallel running units try to allocate free pages from a *line*, the allocation procedure can be a new bottleneck. Third, different entries inside the translation map don't need to be accessed exclusively. However, PBLK uses one spin-lock to protect the whole map. This may cause extra overheads under heavy workloads.

We propose QBLK, an open source [3] driver which tries to better exploit the parallelism of Open-Channel SSDs. Specifically, QBLK adopts four key techniques, namely (1) Multi-queue based buffering, (2) Per-channel based address management, (3) Lock-free address mapping, and (4)Fine-grained draining. Our experimental result shows that QBLK achieves up to 97.4% bandwidth improvement compared with the state-of-the-art PBLK scheme.

The implementation of QBLK is just the first step of our design. Its theories and schemes can be expanded to file system and user application designs, which would be our future work.

This paper is organized as follows. Section II introduces our design of QBLK. Section III evaluates the effectiveness

This work is supported by the Nature Science Foundation of China under Grant No. 61821003, No. 61772222, No. U1705261, No. 61832007, and the National Science and Technology Major Project No.2017ZX01032-101.

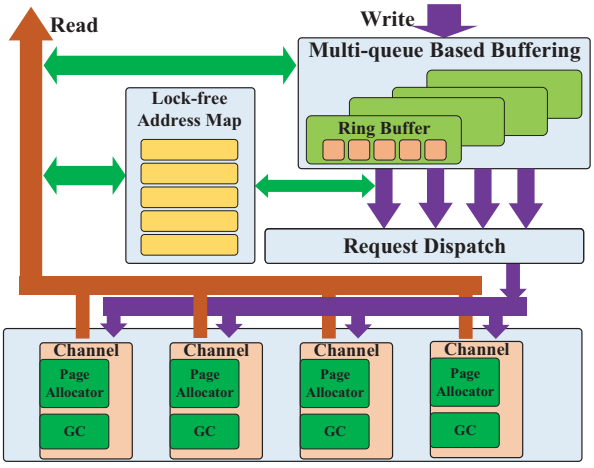


Fig. 1: QBLK architecture

of QBLK. Section IV provides the related works. Section V concludes.

II. QBLK DESIGN

In this section, we introduce our design of QBLK. We implement QBLK in the Linux 4.16 based on the lightNVM subsystem [1]. The source code is available on GitHub: https://github.com/WNLO-DSAL/QBLK. The architecture of QBLK is shown in Fig. 1. QBLK adopts four key techniques, namely (1) Multi-queue based buffering, (2) Per-channel based address management, (3) Lock-free address mapping, and (4) Fine-grained draining. Detailed information of the four techniques are described from Section II-A to Section II-D.

A. Multi-queue Based Buffering

One of the responsibilities for QBLK is to cache necessary amount of data before programming a flash page. Similar to PBLK, QBLK also adopts ring buffer based write caching. The difference is that QBLK uses multiple ring buffers instead of one. In this section, we first introduce our experimental observations. Then, we provide QBLK's implementations of multi-queue based buffering in detail.

1) Observations for Single Buffering: We use fio [4] to test the CPU consumption of PBLK which uses a single buffer to cache writes under different number of threads. Detailed experimental setup is described in Section III-A. As shown in Fig. 6(a), as the number of workload threads increases from 1 to 32, the percentage of CPU time consumed by the spin-lock of PBLK's ring buffer increases from 0.5% to 84.05% and the percentage of CPU's idle time decreases from 91.19% to 0%. As a result, PBLK's ring buffer architecture suffers badly from the competition among different threads. With more time consumed by the spin-lock, system overall performance will decrease.

2) Implementation of Multi-queue Based Buffering: QBLK adopts a multi-buffer architecture. The implementation of multi-buffering is based on the linux BLK-MQ [5] infrastructure. During the initialization procedure, QBLK allocates a ring buffer for each CPU and bind it with the per-CPU queue allocated by BLK-MQ. The calculation of write buffers' total size is identical to [1]. Since QBLK allocates more ring buffers than PBLK, the capacity of each ring buffer in QBLK is M times less than that of PBLK, where M is the number of host CPUs.

When write request comes, the writing thread chooses the ring buffer based on the CPU it currently running on. Different from traditional BLK-MQ based drivers, QBLK doesn't actually queue any request during the queuing procedure. Instead, it simply copies user data to the ring buffer and finishes that request. Draining procedure isn't in the critical path. QBLK allocates one kernel thread for each ring buffer to write back dirty pages inside the buffer. We'll introduce writeback strategies in detail in section II-B2 and II-D. The ring buffers are pure write buffers, in other words, they don't cache read requests. As a result, there is no competition for one ring buffer between multiple user threads on different CPUs.

When data is in a ring buffer and not outdated, the mapping table should points to it for further lookup operations. However, QBLK doesn't do in-place updates in ring buffers. In other words, QBLK allocates buffer entries even though it hits the buffer. Since we have multiple ring buffers, an extra field indicating ring buffer ID (RBID) is used in the translation map entry. We will discuss the implementation detail of translation map in section II-C.

The multi-queue based buffering reduces the competition among threads, thus achieving better performance scalability. However, the enhanced performance doesn't come for free. As we've mentioned before, if the total buffer size remains unchanged, the capacity of each per-CPU buffer is much smaller than that of an unique buffer. If intense IOs are issued by only few number of application threads, we cannot utilize buffers bound with CPUs which are not running such threads. Under this circumstance, QBLK delivers less performance. There are two ways to further optimize for this rare situation: (1) From applications' perspective, use multi-threads to do the IO intensive jobs. (2) From QBLK's perspective, further optimize the multi-buffer architecture. We would like to do it in the future.

B. Per-channel Based Address Management

1) The Competition Domain: The physical address management of PBLK is based on a logical structure called *line*. A *line* contains several blocks with identical block numbers inside their parallel units (Logical Unit Numbers or LUNs). At any given time, PBLK only activates one *line*. Pages are allocated linearly inside a *line*.

The line-based address management is good for PBLK because there is only one client for the physical address management subsystem. However, it doesn't apply to QBLK. With multiple kernel threads draining data from ring buffers, locks or semaphores are needed to ensure metadata consistency when allocating memories or switching the activated line. More number of kernel threads operate in parallel, more intense they competes.

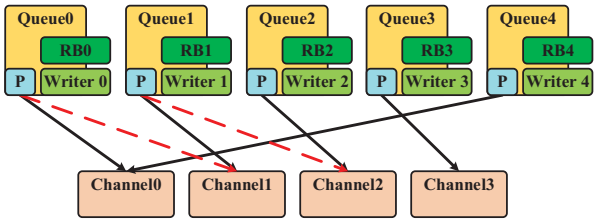
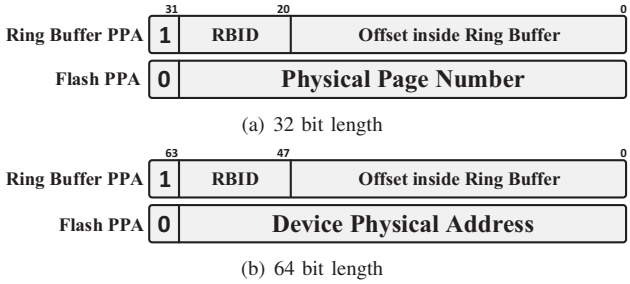


Fig. 2: Queue Mapping of QBLK



(a) 32 bit length

(b) 64 bit length

Fig. 3: Data Structure of QBLK's Map Entry

Such kind of competition comes from the mismatch of parallel units between applications and devices. In our case, draining threads and LUNs are reflections of parallel units of user applications and the SSD device, respectively. Draining threads competes with each other to allocate free pages.

Obviously, we can not statically assign a LUN to a draining thread because it will badly degrade the system parallelism when there are only few user threads running.

Essentially, PBLK's approach, which combines a contiguous free space to one entity, i.e. *line*, and uses it to satisfy the page allocations, can not mitigate the competition. With K user threads operates in parallel, the competition domain of the entity would be K. Likewise, other schemes that combine a contiguous free space to an entity may also suffers from this kind of competition.

Another intuitive approach is to organize free spaces by LUNs and to allocate free spaces from an idle LUN. However, this will make the metadata management too scattered and more space-consuming. Besides, maintaining and checking LUN's status under multiple threads are complex operations. System engineers should avoid complicated procedures to get rid of unnecessary overheads when designing system software for media with relatively low latency such as flash.

2) QBLK's Approach: As shown in Fig. 1, QBLK adopts a per-channel based address management. The page allocation and garbage collections within a channel is similar to PBLK's *line* based management. The difference is how to select the write back channel.

Fig. 2 shows an example of QBLK's channel mapping. As we've mentioned before, a ring buffer is bound to a request queue and a writeback kernel thread. The kernel thread periodically checks whether its corresponding buffer has gathered enough data to write back. Once the data is ready, the kernel thread immediately write them back. For each queue, there's an atomic pointer P points to its write back channel. P is initialized to $(i \bmod N)$ where i is the queue id and N is the number of channels. For each request, its draining thread uses P to decide which channel it writes back to. Every time a drain thread reads P, it moves the pointer pointing to the next channel. Since P is a per-queue pointer and there's only one draining thread for each queue, no lock is needed for P.

In the remainder of this section, we focus on how QBLK address the following two challenges during writeback:

(i) How to reduce the competition domain among draining threads?

(ii) How to preserve good performance under few application threads.

For challenge 1: When QBLK's running, there are roughly M/N draining threads competing the allocation lock of each channel where M is the number of queues. In other words, QBLK shrinks the competition domain of each channel to approximately M/N.

For challenge 2: QBLK's approach naturally preserve good performance under few application threads. For example, when there is only one user thread running, the corresponding draining thread will allocate free flash pages by iterating all channels of the device.

C. Lock-free Address Mapping

It's important to preserve the constraints of access atomicity to the translation map. PBLK takes a simple approach where it uses one spin lock for the whole map. This is in-efficient because different entries inside a map do not need to be accessed exclusively. Therefore, it's practical to loose the atomic granularity into one map entry. However, it's impractical to allocate one lock for each entry because the space overhead would be too much. Since 32/64bit atomic memory access primitives are widely supported by modern CPUs [6], we can leverage the atomic primitives to build a lock-free translation map.

As shown in Fig. 3, there are two types of map entry size in QBLK (32 or 64 bits). When allocating the translation map, every map entry are aligned on a 32/64-bit boundary (depends on the map entry size). Using which type of map entry depends on the total OCSSD capacity after over-provisioning. QBLK always tries to use smaller map entry size whenever it is possible. A tag in the most significant bit indicates whether this entry points to a buffer entry. If it does (Tag=1), an RBID (ring buffer ID) further indicates which ring buffer it points to.

With the help of *atomic_read()* and *atomic_set()* primitives, we can easily get/set data from/to a map entry in an atomic manner. However, there are two situations where these primitives don't work.

Situation 1) When application data is newly written to a ring buffer, we need to get the address of the corresponding old data, invalidate it, and set the address of ring buffer entry

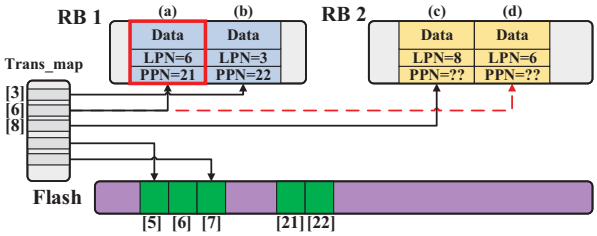


Fig. 4: Translation Map Example

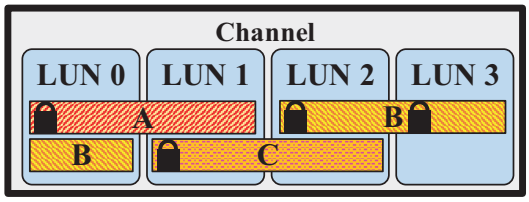


Fig. 5: A Potential Deadlock QBLK Must Avoid

into the map entry, in an atomic manner. Simply using the above-mentioned primitives may cause duplicated or missed invalidations.

Situation 2) Freeing space from an ring buffer, as shown in Fig. 4. Ring buffer entry *(a)* is chosen to be the victim entry. Its data has already been written into flash page 21. We need to check whether the logical page number (LPN) 6's map entry still points to ring buffer entry *(a)*. If it does, we should set the map entry's value to 21 so that further lookup can succeed. If it doesn't, we shouldn't change that map entry because the data in flash page 21 is already out-dated. Instead, we should invalidate flash page 21. **These operations to the translation map can't be interleaved by other instructions.** Otherwise, for example, just before we set the map entry to 21, LPN 6 is written by another thread on CPU 2 and mapped to buffer entry *(d)*. At this time, setting the map entry's value to 21 would be a terrible mistake.

QBLK utilizes two other primitives to overcome these challenges. They are not commonly used but fit quite well with our scheme. For situation 1, QBLK uses *atomic_xchg()* to atomically swap out the current value of the target map entry while setting a new one. The data which old entry value points to is later invalidated. For situation 2, QBLK uses *atomic_cmpxchg()* to compare the address of buffer entry and the target map entry. If the map entry still points to the buffer entry, it will be swapped with the physical address that the entry data has been written to.

D. Fine-grained Draining

When writing back pages from a ring buffer, it's usually better to submit more pages inside a request. By doing so, we can decrease the amount of total requests, diminishing software overhead. However, in PBLK, the number of pages for a write back request is fixed to *min_write_pages* which equals to the maximum number of pages in a LUN that can be accessed in parallel.

Simply increasing the maximum number of pages that a write back request can carry may lead to unexpected dead lock, as shown in Fig. 5. For each write back request, it must get all the corresponding LUN semaphores before submitting to device. Request A, B and C are writing back to the same channel. A, B and C respectively claims the semaphores of 0/1, 2/3/0 and 1/2. A dead lock will appear if A gets LUN0, B gets LUN2/3 and C gets LUN1. QBLK adopts a simple scheme to avoid this issue without importing extra locks/semaphores. When allocating pages, a draining thread uses a bitmap to mark all the LUNs it corresponds to and acquires LUN semaphores in an ascending order before the request submission. Thus, the draining thread of B will not get LUN2's semaphore before it holds LUN0's and no deadlock will appear.

QBLK achieves another improvement by adding the budget of per-LUN semaphores to a higher value than 1. If the initial value of the semaphore is set to 1, a LUN can be idle when it completes a request while the next one hasn't come. By increasing the initial value of per-LUN semaphore, QBLK enables more number of in-flight requests inside a LUN and eliminates the idle time. When implementing QBLK, we find that the marginal performance gain of adding the initial value of per-LUN semaphore decreases dramatically. In QBLK's implementation, we set this value to 8.

TABLE I: Test Configurations

Hardware Configurations		FIO Configurations	
CPU	32	**IO pattern**	Random Writes
Memory Capacity	32GiB	**DirectIO**	Yes
OCSSD Capacity	64GiB	**ioengine**	libaio
Channels	32	**iodepth**	2
LUNs per Channel	4	**iodepth_low**	2
Planes per LUN	2	**Duration**	30s
Flash Page Size	16KiB	**Request Block Size**	4KiB

III. Evaluation

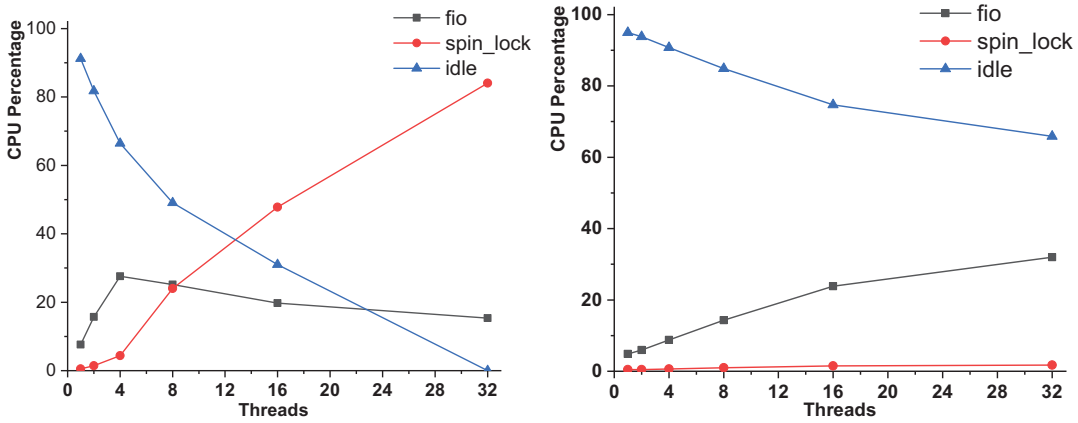
A. Evaluation Methodology

In this section, we evaluate the performance of QBLK. We try to answer the two following questions:

1) How much host capacity does QBLK consume compared with PBLK?
2) How is the performance of QBLK?

To conduct our test environment, we use FEMU [7], a QEMU/KVM based virtual machine manager. FEMU uses host's DRAM to emulate an Open-Channel SSD for the guest virtual machine. We use exactly the same latency parameters in [7] which are profiled from a real Open-Channel SSD. Hardware configuration details are listed in Table I.

We first use perf [8] along with flame graph [9] to evaluate the CPU consumption of host software. Then, we use fio [4] to evaluate the performance of QBLK. Fio configurations are listed in Table I.



(a) PBLK (b) QBLK

Fig. 6: CPU time consumption

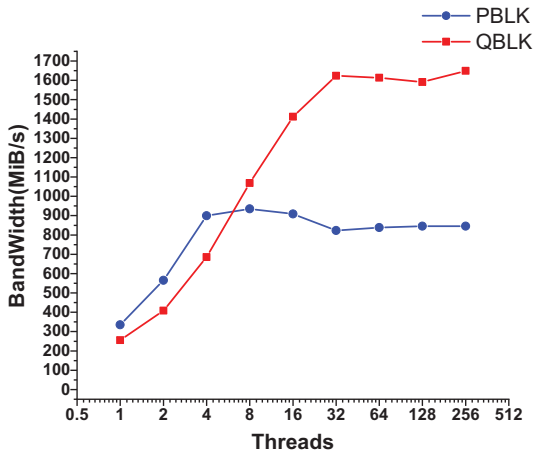


Fig. 7: Bandwidth Test

B. CPU consumption

The CPU consumption of PBLK and QBLK is respectively shown in Fig. 6(a) and 6(b). As the number of threads increases from 1 to 32, the CPU time consumed by QBLK's spin-locks nearly doesn't change (from 0.46% to 1.73%). This shows the effectiveness of how QBLK reduce the competition under heavy workloads. Compared with PBLK, QBLK can save CPU time dramatically under heavy workloads. For 32 threads where PBLK's CPU utilization is almost 100%, QBLK can still reserve 65.86% CPU idle time for application usage.

C. Bandwidth Test

Fig. 7 shows the bandwidth test for PBLK and QBLK. The number of test threads is increased from 1 to 256. We can draw two observations from the figure.

1) For PBLK, as the number of test threads grows from 1 to 8, total bandwidth increases from 335MiB/s to 935MiB/s, reaching it's best performance. As the number of test threads continues to grow, PBLK meets some performance fluctuations.
2) For QBLK, as the number of test threads grows from 1 to 32, total bandwidth continues to increase from 256MiB/s to 1624MiB/s. As the number of test threads further increases to 256, the total performance maintains at about 1600MiB/s.

QBLK achieves up to 97.4% bandwidth improvement with 32 test threads. QBLK's peak bandwidth is 76.4% higher than that of PBLK's.

QBLK enjoys better performance under heavy workloads. The reason is that it can successfully reduce the competition domains in two layers: (1)requests from user threads to ring buffers and (2)data from buffers to device channels. With fewer time consumed by competing different kinds of locks and requests' software overhead, the overall performance increases naturally. As the number of test threads adds to more than 32, no further performance improvement has been found. The reason is that our test device only has 32 CPUs. As the number of test threads reaches 32, it can already fulfill all the ring buffers that QBLK allocates, thus reaching its peak performance.

When the number of user threads is low, QBLK can't achieve better bandwidth compared with PBLK. We've analyzed the reason and possible solutions in section II-A2.

IV. Related Works

In order to better exploit the performance of NAND flash SSDs , various layers of internal parallelism has been explored [10] [11]. While traditional SSDs embed their FTLs in device sides to work as a generic block device [12], some researchers and vendors go for a different approach and try to manage FTL functions at host sides.

Linux flash file systems such as JFFS [13] and YAFFS [14]are build upon raw flash devices. They use log-structure techniques to handle the flash's out-of-place-update issue. UBIFS [15] goes for a further step by using UBI layer to take care of bad block management. UBIFS uses B+ tree to index filesystem elements to decrease the wandering tree overhead. Besides, UBIFS uses anchor erase blocks to prevent

superblock from being wear out quickly. Thus, UBIFS delivers better performance and stability than JFFS/YAFFS. However, these file systems are built upon a Memory Technology Device (MTD), which do not support asynchronous requests and lack the whole system performance.

For PCIe SSDs, SanDisk/Fusion-io developed a virtual flash storage layer to manage SSDs by utilizing host CPU/memory capabilities [16]. By simplifying the complexity of file systems, it can achieve better performance. ParaFS [17] is a file system used upon a customized PCIe raw flash device. The flash device shares its channel information with host software. ParaFS handles the FTL and uses a 2-D data allocation scheme to separate hot/cold data without satisfying SSD's parallelism. AMF [18] is an open-sourced NAND flash storage system. It utilizes the traditional Linux block layer to access the raw flash device by forbidding the upper layer to write one physical page twice without interleaving an erase command. Upon the block layer, they developed ALFS, a no in-place update version of F2FS [19], to use the flash device. ParaFS and AMF are both log-structured file systems. They both allocate free space in a segment based manner. Therefore, as we've discussed in section II-B1, they may suffer from writing back threads competing the lock of one segment under heavy workloads.

LightNVM [1] is a Linux kernel sub-system aiming at better exposing device channel informations to upper layers, including drivers, filesystems and user applications. Different from above-mentioned schemes, LightNVM make use of NVMe [2] to better deliver commands and use Open-Channel SSD to satisfy the requirements.

Jhin et al. proposed a multi-buffer strategy MT-FTL for OCSSDs [20]. QBLK's multi-queue based buffering is similar to MT-FTL. However, QBLK further exploit OCSSD's parallelism by digging into more FTL procedures such as translation map lookup and data writeback. Besides, QBLK uses real world OCSSD's latency to conduct test environment.

BLK-MQ [5] is a linux block layer infrastructure. It enables drivers to make use of multiple queues (MQ) instead of one single queue (SQ). By doing so, request queue based drivers are able to handle IO requests in a scale-out manner. QBLK utilizes BLK-MQ as its block level abstraction.

V. Conclusion

QBLK is an open source Open-Channel SSD driver. The design goal of QBLK is to fully exploit the parallelism of Open-Channel SSDs while minimizing the software overhead.

QBLK adopts several schemes to achieve this goal. First, QBLK uses multi-queue based buffering to minimize the queuing conflicts among applicaton threads. Second, QBLK adopts an per-channel based address management to diminish the free space allocation conflicts among ring buffer draining threads. Third, for the mapping table management, QBLK utilizes primitives that most modern CPU supports to avoid unnecessary conflicts when accessing different map entries. Last, QBLK enables finer draining granularity by eliminating the potential deadlock issue and hides software overhead by increasing access budgets of LUNs.

Experimental results show that QBLK achieves up to 97.4% bandwidth improvement compared with the state-of-the-art PBLK scheme.

References

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

[2] "Nvm express," https://nvmexpress.org/.

[3] H. Qin, "Qblk," https://github.com/WNLO-DSAL/QBLK.

[4] J. Axboe, "Flexible i/o tester," https://github.com/axboe/fio.

[5] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet, "Linux block io: introducing multi-queue ssd access on multi-core systems," in *Proceedings of the 6th international systems and storage conference*. ACM, 2013, p. 22.

[6] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3B: System programming Guide, Part*, vol. 2, 2011.

[7] H. Li, M. Hao, M. H. Tong, S. Sundararaman, M. Bjørling, and H. S. Gunawi, "The case of femu: cheap, accurate, scalable and extensible flash emulator," in *16th USENIX Conference on File and Storage Technologies (FAST)*, 2018, pp. 83–90.

[8] B. Gregg, "Linux perf tools," https://github.com/brendangregg/perf-tools.

[9] ——, "Flame graph," https://github.com/brendangregg/FlameGraph.

[10] 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 *2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2011, pp. 266–277.

[11] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and C. Ren, "Exploring and exploiting the multilevel parallelism inside ssds for improved performance and endurance," *IEEE Transactions on Computers*, vol. 62, no. 6, pp. 1141–1155, 2013.

[12] Y. Feng, D. Feng, C. Yu, W. Tong, and J. Liu, "Mapping granularity adaptive ftl based on flash page re-programming," in *Proceedings of the Conference on Design, Automation & Test in Europe*. European Design and Automation Association, 2017, pp. 374–379.

[13] D. Woodhouse, "Jffs: The journalling flash file system," in *Ottawa linux symposium*, vol. 2001, 2001.

[14] C. Manning, "How yaffs works," *Retrieved April*, vol. 6, p. 2011, 2010.

[15] A. B. Bityutskiy, "Jffs3 design issues," 2005.

[16] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn, "Dfs: A file system for virtualized flash storage," *ACM Transactions on Storage (TOS)*, vol. 6, no. 3, p. 14, 2010.

[17] J. Zhang, J. Shu, and Y. Lu, "Parafs: A log-structured file system to exploit the internal parallelism of flash devices." in *USENIX Annual Technical Conference*, 2016, pp. 87–100.

[18] S. Lee, M. Liu, S. W. Jun, S. Xu, J. Kim, and A. Arvind, "Application-managed flash." in *14th USENIX Conference on File and Storage Technologies (FAST)*, 2016, pp. 339–353.

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

[20] J. Jhin, H. Kim, and D. Shin, "Optimizing host-level flash translation layer with considering storage stack of host systems," in *Proceedings of the 12th International Conference on Ubiquitous Information Management and Communication*. ACM, 2018, p. 75.