DapDB: Cutting Latency in Key-Value Store with Dynamic Arrangement of internal Parallelism in SSD

1st Heerak Lim
Seoul National University
Seoul, Republic of Korea
rockylim@snu.ac.kr

2nd Hwajung Kim
Seoul National University
Seoul, Republic of Korea
fanciful02@gmail.com

3rd Heon Young Yeom
Seoul National University
Seoul, Republic of Korea
yeom@snu.ac.kr

4th Yongseok Son
Chung-Ang University
Seoul, Republic of Korea
sysganda@cau.ac.kr

***Abstract*—Modern data centers aim to take advantage of high parallelism in storage devices for I/O intensive applications such as storage servers, cache systems, and database systems. Database systems are the most typical applications that should provide a highly reliable service with high-performance. Therefore, many data centers running database systems are actively introducing next-generation high-performance storage devices such as Non Volatile Memory Express (NVMe) based Solid State Devices (SSDs). NVMe SSDs and its protocol are characterized by taking full advantage of the high degree of parallelism of the device which is expected to ensure enhanced performance of the applications. However, taking full advantage of the device's parallelism does not always guarantee high performance as well as predictable performance. In particular, heavily mixed read and write requests give rise to serious performance degradation on throughput and response time due to the interferences of requests in SSDs each other. To eliminate the interference in SSDs and improve performance, this paper presents DapDB, a low-latency Key-Value Store tailored for Open-Channel SSD with dynamic arrangement of internal parallelism in SSDs. We divided and isolated entire parallel units (LUNs) of the NVMe SSD into tree type and re-arrange them to three different types of LSM-tree base Key-value store. To implement DapDB based on RocksDB, we used Open-Channel SSD. We modified storage backend of RocksDB to run application-driven Flash management scheme using Open-Channel SSD. DapDB can fully control internal parallelism of SSD. It may take advantage of entire parallelism for every I/O request, It can use the entire parallelism of the SSD for every I/O request, or it can use a strategy to yield the degree of parallelism of the I/O request to reduce the interference between the various I/O requests. Our extensive experiments have shown that DapDB's Isolation-Arrangement scheme achieves both improved overall throughput and response time, i.e., on average 1.20× faster and 43% less than traditional Striping-Arrangement scheme respectively.**

***Index Terms*—Storage, NAND-flash, Open-Channel-SSD, FTL, NVMe, LSM-tree**

I. Introduction

The role of high-performance storage devices is becoming indispensable to I/O intensive application. In particular, in database systems, the performance of storage devices directly affect their quality of service. Therefore, many data centers running database systems are introducing next-generation storage devices represented by NVMe SSDs to improve performance. In addition, to effectively utilize these high-performance storage devices, modern database systems are considering flash-based storage devices. For example, most of NoSQL database or key-value store systems including RocksDB [7], Apache Cassandra [8], and HBase [9] use a Log-Structured Merge tree (LSM-tree) [6] data structure which works well on flash-based SSDs.

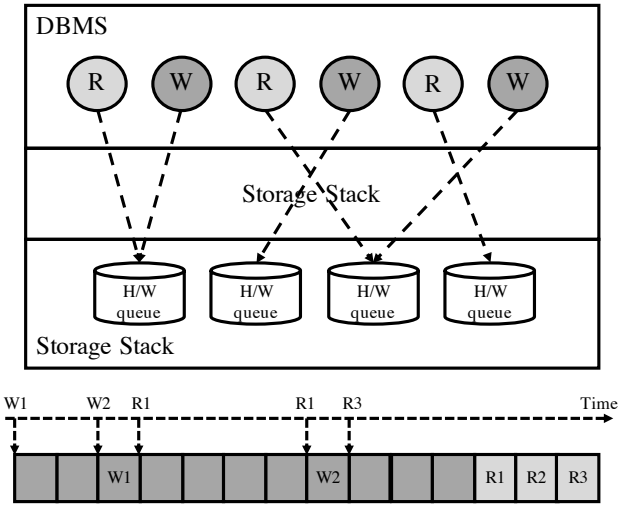
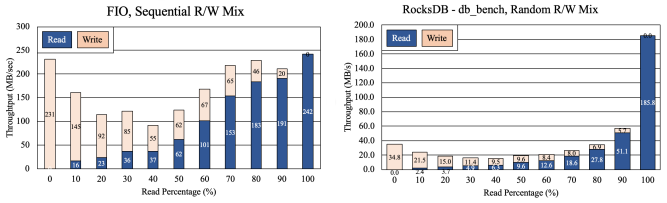


Fig. 1: Hardware Queue Contention in NVMe SSD. Read requests are delayed due to the write request which has long response time relatively

NAND-based NVMe SSDs provides higher read and write throughput compared to spinning drives or SATA-based SSDs. However, modern database systems do not fully utilize the capabilities of the NVMe SSD devices. The reason for this inefficiency is that many database systems are not designed with thorough consideration of the features of the NVMe SSD storage device.

On systems using NVMe-based SSD, single read requests can be handled very rapidly, but serious performance degradation will occur if write requests are requested at the same time. This is due to the difference in response time between read and write requests and delays come from internal FTL operations such as Garbage Collection (GC). Previous studies found that performance degradation could be up to 450% under the workloads with mixed reads and writes [17], [18]. Figure 1 shows how heavy write requests and light read requests mixed at a H/W queue in NVMe SSD. If read requests are placed in the same H/W as write requests, contention occurs and read requests are delayed due to the write requests that require



(a) FIO benchmark (b) RocksDB db_bench tool

Fig. 2: Read and write throughput comparison based on various read percentage in the workload.

long response time. Furthermore, unlike ordinary relational database systems, in the LSM-tree based key-value store, additional I/O requests are generated due to the compaction operation that creates a higher-level SST file by merging lower-level Sorted String Table (SST) files. For example, if clients send only insert requests, underlying flash-based storage devices must handle amplified number of write request as well as multiple read requests due to the compaction. As a result, this I/O amplification makes read and write requests mixed more frequently in LSM-tree based key-value store systems.

To quantitatively measure the performance degradation under the mixed workload of reads and writes of the device and database application on commercial NVMe SSDs [14], we used a Flexible I/O (FIO) benchmark tool [1] and RocksDB db_bench tool [13] respectively. Figure 2a represent the result shows serious deterioration in throughput performance depending on the degree of mixed read and write. In particular, when read percentage is 50, that is, in a workload which reads and writes are half occupied, the read performance is about 25.6% of read-only performance. Also, the write performance dropped to about a quarter of the write-only performance at the 50% read ratio. This graph shows a dramatic drop in performance depending on the degree of mixed reads and writes. Figure 2b illustrates that performance degradation due to I/O interference is much more serious in RocksdB. Even if the write request is mixed with only 10 percent and the read request accounts for 90 percent of the total workload, the read performance drops to a quarter of the read-only performance. The reason for this result came from write amplification due to the nature of the LSM-tree based key-value store that storage devices such as SSDs must accommodate requests larger than the client's write requests.

Modern NVMe SSDs have a set of parallel units (e.g., multiple channels) and allow host hardware and software to fully exploit the levels of parallelism. In the traditional FTL of legacy SSDs, when a SSD controller receives incoming write requests, it determines the data placement in such a way that it can access NAND flash in parallel as much as possible considering the internal geometry of the SSD. In a read-only or write-only workload, this scheme which maximizes the degree of parallelism represents best performance and many SSDs vendors present these fragmentary performance result (e.g., Sequential Read, Random Write, etc.) as their performance indicators. However, in real-world applications that need to serve intensive I/O at data centers rarely process this kind of lopsided I/O access patterns. In many real-world workloads, there are frequent mixes of read and write requests, and locality or hotness at the data.

In this paper, we propose DapDB, a LMS-tree based key-value store which directly manage internal parallelism of SSD taking its application context and I/O pattern into account. We design and implement application-driven flash management scheme that dynamically changes the arrangement of NAND-flash's internal parallel units on DapDB. **The main contributions of this work** are listed as follows:

oc_bench: Preliminary experimental evaluation. We designed and implemented FIO-like benchmark tool called *oc_bench* for Open-Channel SSD [12]. *oc_bench* is a tool for evaluating device performance, changing many parameters, such as number of threads or file size, as well as existing benchmark tools such as fio and iometer. In addition, *oc_bench* can determine physical address of data to be stored by utilizing the features of Open-Channel SSD. We evaluated the I/O performance of NVMe-based SSD, a Open-Channel SSD, when read requests are isolated from write requests physically on NAND flash. To isolated them, we configured each I/O thread access their own region without interference of other I/O threads by dedicated LUNs per I/O threads. The evaluation results demonstrate the overall performance is improved by up to 220% compared to the baseline that each thread use entire LUNs greedily.

I/O type isolation through application-driven flash management. In order to apply the read/write isolation scheme in *oc_bench* to DapDB, we modified the RocksDB storage backend so that the write requests of different I/O types (e.g., LOG file, Level 0 SST file, Compaction etc.) does not physically overlap within the NAND Flash each other. Unlike micro-benchmark tool(e.g., FIO or *oc_bench*), database applications must read data at the specific location where the data physically written. Thus, DapDB can not completely isolate read and write requests in the SSD physically. However, the unique characteristics of LSM-tree base database running as append-only and the write-only files (Write Ahead Log) can mitigate the mixing of read and write in runtime. We carefully evaluated DapDB under various read:write ratio using micro benchmark tool. The result represent that DapDB reduce average read response time by 43% compared to the baseline that works in a greedy manner.

Dynamic arrangement of NAND-flash parallelism In workload that read and write requests are frequently mixed, DapDB shows improves performance, but in other workloads such as read-only or write-only, existing scheme that make the most of parallelism performs better than our DapDB. To ensure that DapDB is flexible and work best for all workloads, we added dynamic LUNs arrangement scheme to DapDB. In the write-intensive workload that relatively requests are not mixed frequently, we made each I/O type of DapDB utilize entire LUNs to support high parallelism. To determine the arrangement of LUNs by analyzing the nature of the workload at runtime, we made simple count-based workload profiler

on DapDB's storage backend. We evaluated DapDB under a workload which the nature of the workload is changed during runtime. Evaluation result shows that DapDB responds flexibly to changing workloads and performed better than static LUNs arrangement scheme.

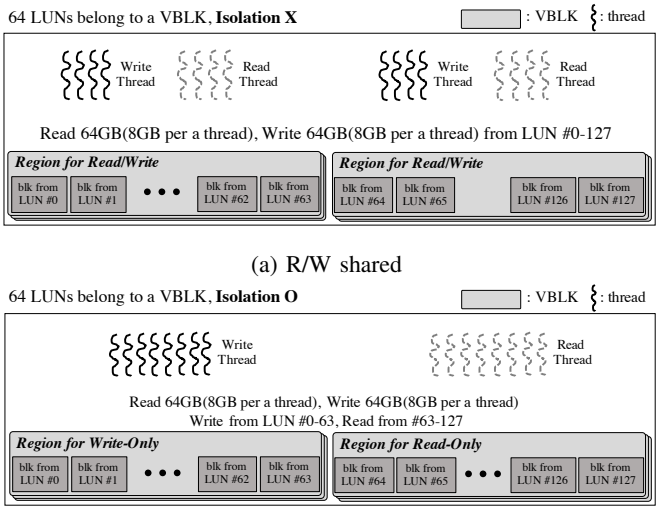
II. Background

A. Log-Structured Merge tree based Database

LSM-tree data structure is used in many modern key-value stores and storage systems to provide fast I/O services. It stores data in an append-only manner and thus has a fast write performance. In detail, the incoming key-value pairs from clients are written in sorted order in the in-memory write buffer called memtable. At the same time, Write Ahead Logging (WAL) files are written to persistent storage (HDD, SSD, etc.) for recovery in case of power faliure or crash. If data is written as much as the allocated size of memtable, background flush threads flush this memtable to the persistent storage device as Sorted String Table file (SST) of level 0 (L0). If a level 0 SST file is continuously generated in this manner, the capacity of level 0 is exceeded spoiling the tree shape of the LSM-tree data structure. Then, a background compaction mechanism is triggered to constrain the LSM tree shape. It read multiple L0 SST files to merge them to a next level SST file. This compaction mechanism works in the same way at level 0 as well as at other levels. While compaction, there may be multiple duplicate key-value pairs among the input SST files of compaction. Only one valid key remains in the compaction output SST file, and the remaining invalid key-value pairs are deleted. Through this compaction mechanism, LSM-tree-based database can store key-value pairs in append-only manner without in-place update. This LSM-tree algorithm is very similar to the internal implementation of FTL which merges physical blocks to make free blocks due to the nature of flash memory, which cannot be in-place update. In RocksDB, our target application, each different I/O type access to storage device by different threads. For example, *WAL files* are flushed while inserting a key-value pair into a memtable by the foreground main I/O threads, and the *L0 SST files* and the *L* (* > 0)SST files* are flushed by background flush threads and background compaction threads respectively.

B. Open-Channel SSDs

Recently, a new class of SSDs, Open-Channel SSD has been proposed as a method to manage the internal parallelism of SSDs. They expose the geometry inside of the SSD and share control responsibilities with the host in order to implement and maintain features that typical SSDs implement strictly in the device firmware. As a consequence, host could manage data placement, I/O scheduling, and GC, etc. So, it is possible to optimize the NAND-flash based storage device taking into account the application context or kernel context at host side. To facilitate this optimization and application design, Open Channel SSDs provide a user space I/O library for Open-Channel SSDs, a *liblightnvm* [11]. *liblightnvm* provides an interface that allows an application to do I/O using physical addresses of the SSD. Using *liblightnvm* C API, user space application can directly send a request to the device driver to read, write, or even to erase data with the specific physical address. Since NAND-flash works very sophisticatedly, there are complicated read and write limitations such as order-constraint for programming pages in a block, request-size-constraint considering plane mode. To resolve this difficulty, *liblightnvm* also provides a virtual block (vblk) interface, which works similar to lib-c *write*, *read*, and *pread*. A vblk consists of a set of physical blocks in the SSD and can be created to span all parallel units (LUNs) of SSD or a subset thereof. For example, with Open-Channel SSD equipped with 16 channels and 8 dies(chips) per channel, I/O thread can access up to 128 independent blocks at once using vblk which consist of 128 blocks from each of 128 LUNs.



(a) R/W shared

(b) R/W isolated

Fig. 3: oc_bench: Comparing architectures when read and write requests are physically isolated and processed in Open-Channel SSD using the *oc_benc tool*.

III. *oc_bench: Preliminary Experimental Evaluation*

We found the performance degradation due to the interference between the read requests and write requests through FIO experiments using commercial NVMe SSDs. In this chapter, we demonstrate how performance can be improved if the interference is eliminated by using Open-Channel SSDs We used the Open-Channel SSD and the *liblightnvm* library to implement a FIO-like benchmark tool called oc_bench. oc_bench is able to control the number threads and request size as it is in FIO to bench mark Open Channel SSDs under various test scenario.

Moreover, it is also able to control which LUNs to use to run the benchmark. That is, oc_bench can adjust the arrangement of parallel units (LUNs) by changing the mapping between the physical blocks and the virtual block (vblk) of the SSD. If I/O threads access to the SSD using different vblks each other and physical blocks of SSD that make up these vblks

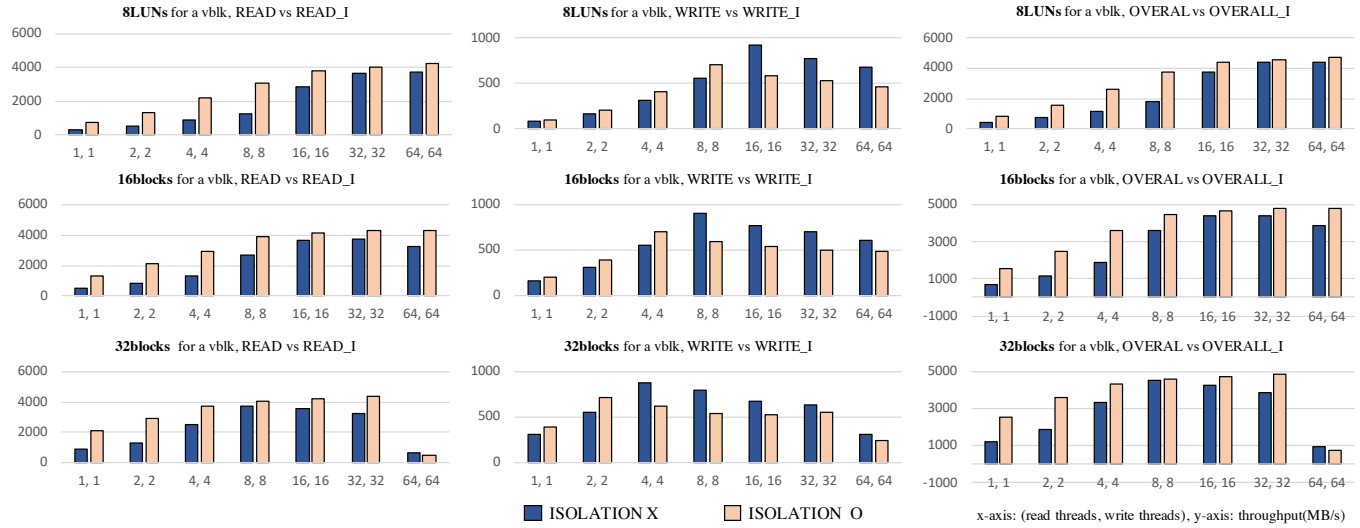


Fig. 4: oc_bench experiment - Comparison of performance improvement due to isolation of read, write, and overall performance depending on vblk size.

are located in all different LUNs, interference will not occur. Figure 3 show how the oc_bench tool works when a vblk is constructed by taking a block from each of 64 LUNs. Under the configuration that reads request and write requests are not isolated, contention occurs in LUNs. However, in the isolated design, contention does not occur because only one of reading or writing occurs in each region. We carefully experimented with various factors such as the number of threads and the size of the vblk.

The result of preliminary experimental evaluation is shown in Figure 4. In most configurations, the read performance was dramatically improved due to the R/W isolation and increased up to 253% when there were two read and write threads respectively. Except for a case with extreme contention (when vblk is consist of 32 physical blocks and 64 concurrent threads access overlapping vblks), read operations that do not use locks generally increase performance as the number of threads increases. In case of writes, the performance was also improved due to reduction of interference. However, As the number of I/O threads increases, the number of concurrent write threads accessing a vblk increases too, resulting in a dramatic drop in performance and thus decreasing the positive effect of isolation. The reason is that in the current Open-Channel SSD implementation, I/O threads use a coarse-grained locking mechanism resulting in lock contention when trying to write a vblk. Therefore, the smaller the number of LUNs configuring a vblk, each thread may access to the isolated LUN and show the result of scalable and improved performance. Despite the unoptimized implementation of concurrent write requests in some configuration, overall performance was improved in all configurations up to %220 because there was a significant performance improvement in reads.

IV. I/O TYPE ISOLATION THROUGH APPLICATION-DRIVEN FLASH MANAGEMENT

Through preliminary experiment results using oc_bench, we found that utilizing SSD device's entire parallelism does not always lead to the best performance. Depending on the workload, it may be better to compromise that read and write requests do not affect each other, rather than using full parallelism for each I/O thread. Hence we implemented DapDB based on RockDB, to apply this characteristics to real-world applications. Unlike benchmark tools, database applications have a lot of limitations in isolating read requests from write requests physically because database applications must read data from the physical location of the NAND-flash where the data written.

However, as shown in figure 5a and Figure5b, LSM-tree based database including RocksDB has different I/O scheme compared to ordinary Database Management Systems (DBMS) or storage systems. In RocksDB, three types of files are written to persistent storage device by key-value insertions and periodical compaction algorithm. First of all, write-only Write Ahead Log(WAL) files are stored by main I/O thread when key-value pairs are inserted in in-memory buffer, a memtable. The WAL file is only read during recovery, so it behaves as write-only in normal operation. Second, the L0 SST file is written by the background flush thread, and read by the main I/O thread and compaction thread. At last, SST files with a level greater than L0 are read and written by the compaction thread, and read by the main I/O thread. Because of these different types of files and the threads that read and write the files, it becomes possible to isolate reads and writes partially considering the behavior of LSM-tree data structure and algorithm. For the simplest, since WAL is write-only, DapDB can allocate a completely isolated LUNs for WAL files so that other threads are not interfered at all. Also, due

to the hotness of the data, high-level SST files are likely to have cold(rarely updated) characteristics. Consequently, they are easy to operate with read-only manner.

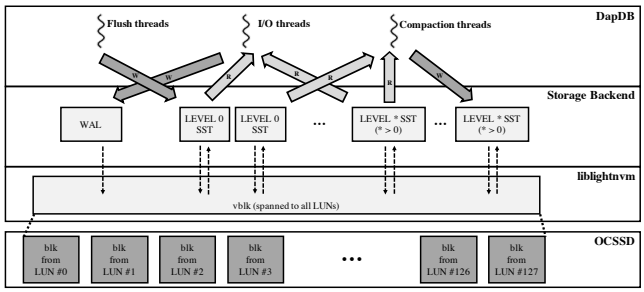
Figure 5a shows a baseline, a Striping-Arrangement scheme of DapDB, to compare with our isolated design. In this architecture, all different type of DapDB share huge virtual blocks (vblks) spanned to all parallel units (LUNs), as in the FTL of existing legacy SSD, to achieve high parallelism. On the other hand, Figure 5b represents the DapDB architecture with our optimization applied, a Isolation-Arrangement scheme. Isolation-Arrangement makes each different I/O type is written to its dedicated LUNs. Under the Isolation-Arrangement scheme, vblks are constructed using only LUNs which do not overlap each other depending on the type of vblk (*alpha, beta, theta*). As a result, all write requests of different I/O type can be isolated each other completely without interference. Although main I/O threads and compaction threads still can read SST files from all level theoretically, reads can be handled separately from writes as compared to the Striping-Arrangement because of the hotness of data and LSM-tree behavior.

A vblk is a set of physical blocks in Open-Channel SSD, each vblk has a predefined bandwidth. This bandwidth depends on how many LUNs the blocks that make up the vblk exist. For example, if a vblk is configured with physical blocks from all LUNs, this vblk has the maximum bandwidth. In our implementation, DapDB stores each WAL file, L0 SST file, and L* (* $>$ 0) SST file to alpha vblk, beta vblk and theta vblk, respectively. We divided the entire LUNs into 3:3:2 ratios and assigned each of the divided LUNs to each type of vblk to have them a independent writing target. To determine the ratios for the isolated bandwidth(i.e., the number of LUNs) assigned to the each vblk(*alpha* for WAL files, *beta* for L0 SST files and *theta* for L* (* $>$ 0) SST files), we used a heuristic method. We set the initial ratios, taking into account the total amount of data read and written to each files of each I/O type. Then, we adjusted the detail ratios through various experiments. These experimental procedures and details will not be discussed because they are beyond the scope of main points of the paper.

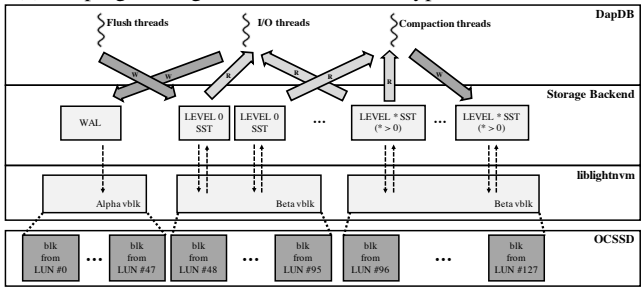
As a result, isolated LUNs divided by a 3:3:2 ratio have improved overall throughput performance over a wide range of workloads with read percentages ranging from 10 percent to 80 percent and dramatically reduced read latency under the workloads with 10%-90% read percentages.

V. Dynamic Arrangement of NAND-Flash Parallelism

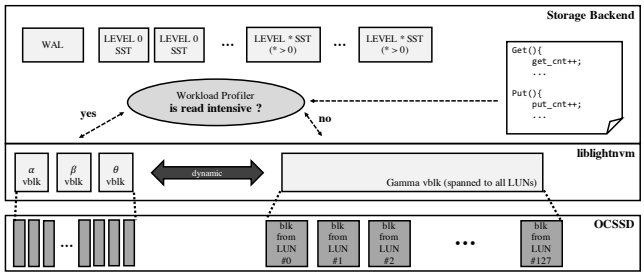
We divide a entire of parallelism (i.e., 128) into 3 portions and assigned each of them to the vblks depending on different needs of parallelism for corresponding their I/O types. As a result, DapDB with Isolation-Arrangement scheme reduce the delay in read requests due to the long turnaround time of the write request under the mixed reads and writes workload. However, statically isolated arrangement of LUNs can lead to performance degradation if the nature of the workload changes



(a) Striping-Arrangement: different I/O types share entire LUNs.



(b) Isolation-Arrangement: different I/O types use their own LUNs to write.



(c) Workload profiler - profiler determine the arrangement of LUNs

Fig. 5: RocksDB architecture using Open-Channel SSD and liblightnvm

to extreme write-intensive which read and write do not mix frequently because of the reduced degree of parallelism. To address this kind of problems, we put a simple workload profiler in the DapDB storage backend to monitor the pattern of the workload. It is a simple and traditional profiler based on the temporal locality of the workload pattern. The profiler counts 2 basic operations of key-value store, the *Put* and *Get*, for a specific period of interval to determine whether the workload is read intensive or write intensive. This uncomplicated profiler has almost no overhead because there are no additional calculations. Figure 5c shows how the profiler works, and DapDB decides which arrangement parallelism to use to store the data by this profiler. We evaluated performance of the Dynamic-Arrangement scheme of DapDB under rapidly changing workload of read: write ratio comparing with Striping-Arrangement scheme and Isolation-Arrangement scheme.

In summary, we implemented DapDB that features an application-driven flash management scheme considering the behavior and data structure of LSM-tree based key-value

store. In DapDB, the internal parallelism of the SSD is used according to the context of the application in a way that considers the interference rather than the traditional greedy scheme. Furthermore, it works in an optimal arrangement depending on the characteristics of the workload. The main benefits of DapDB include: 1) each type of write requests has their own region, thereby alleviating the delay of read comes from the expensive turnarround time of write requests; 2) The size of vblk, the unit of erase operation in liblightnvm, has been divided to smaller size. Therefore, the I/O blocking overhead due to erase operation is reduced; 3) It shows predictable performance due to reduced interference between I/O types.

VI. Evaluation

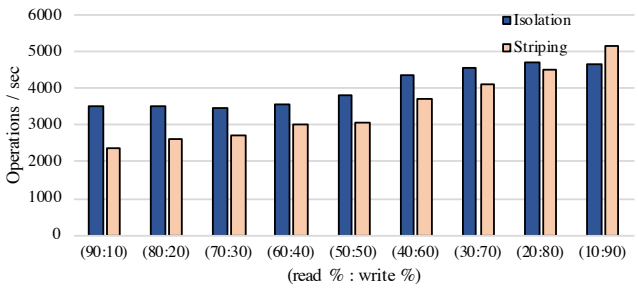
A. Experimental Setup

we conducted extensive experiment to evaluate DapDB by focusing on cutting latency as well as the overall throughput of LSM-tree based key-value store system. We used CNEX's Open-Channel SSD to implement application-driven flash management scheme. The CNEX LABs Westlake SDK is equipped with 2TB NAND MLC Flash with 16 channels and 8 parallel units (LUNs) per channel that make possible 128-concurrent I/O execution. For experimental evaluations, we used a 72-core Intel Xeon E7-8870 processor server machine equipped with 384 GB DRAM, PCI 3.0 interface connected with the Open-Channel SSD. Ubuntu 16.04 server and Linux kernel 4.15.0 version for Open-Channel SSD [15] supported DapDB. We implemented DapDB based on RocksDB's modified version using Open-Channel SSD and liblightnvm [16].

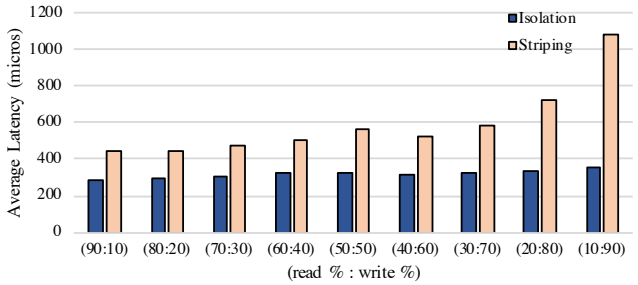
B. Performance Evaluation

In this section, we evaluate the throughput and latency performance of DapDB using db_bench micro benchmark released with RocksDB [13]. We evaluated random read and write performance by inserting and extracting 600,000 key-value items (i.e., 10GB) in a uniformly distributed random order. In all experiments, we started with 100,000 keys inserted in advance to prevent read miss. Because our target workload is a mixed workload of read and write requests, we evaluated the performance of DapDB under various workloads by changing the read-percentage parameter of db_bench from 10 percent to 90 percent.

Figure 6a plots the DapDB's overall throughput performance under 9 different workloads which have different percentage of read operation comparing Striping-Arrangement scheme and Isolation-Arrangement scheme of DapDB. Except for a workload with a read percentage of 10 percent, our work, Isolation-Arrangement scheme, shows improved overall throughput performance results. Also, It shows that as the ratio of read operation decreases, the degree of performance improvement of the Isolation-Arrangement scheme decreases. Especially, when the read operation occupy 10 percent of the workload, the greedy Striping-Arrangement scheme results in better throughput performance because the read requests and write requests are not mixed enough in the SSD. The average



(a) Overall Throughput (ops/s)



(b) Average Read Latency (micros)

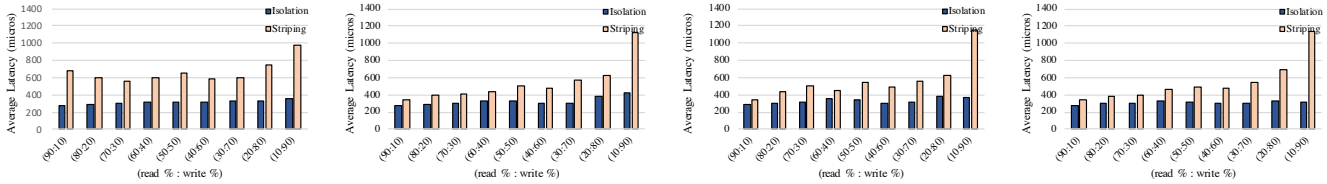
Fig. 6: Striping-Arrangement VS Isolation-Arrangement.

throughput improvement on all workloads is around 20% and the workloads with a read percentage of 90 achieved performance improvements of up to 47% over Striping-Arrangement scheme.

Similarly, we evaluated average read response time which is critical for Quality of Service(QoS). In the experiment, read operations request data in SST files from level 0 to level 3. Figure 6b represents the average response time results for all read operations. In the Striping-Arrangement scheme, the latency increases proportionally as the ratio of write requests to workload increases. On the other hand, the Isolation-Arrangement scheme always shows predictable performance regardless of the percentage of write requests in the workload. It also achieves 43%-reduced average latency performance result and 67% reduced latency under the most write-intensive workload. The reason for this is that the read delay can be improved the most under the write-intensive workload which has long-latency write requests.

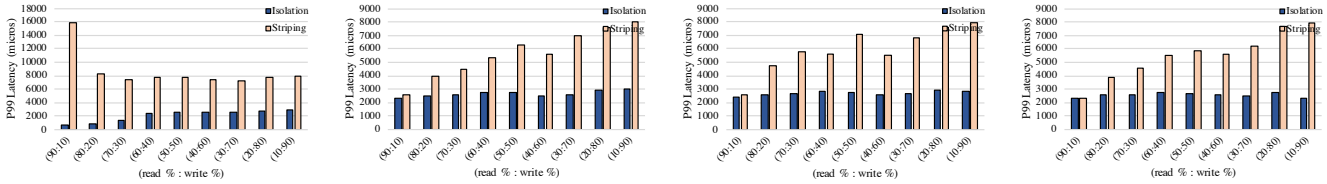
We measured the read latency of each level separately for detailed analysis of read latency. In this experiment, reading occurred in 4 levels of SST files from level 0 to level 3. For all workloads with a read percentage of 10 percent to 90 percent, the Isolation-Arrangement scheme reduce average read latency at all levels versus the Striping-Arrangement scheme and reduce tail latency by up to 96 percent. In all experiments, the Isolation-Arrangement scheme show predictable read latency.

In order to verify the behavior of the dynamic LUNs arrangement scheme, we evaluated DapDB under workload which changes its characteristics during runtime. The entire run-time of the experiment is divided into 50 intervals, and the



(a) Level-0 SST file (b) Level-1 SST file (c) Level-2 SST file (d) Level-3 SST file

Fig. 7: Average Read Latency of key-value pair from Level-0 to Level-3 SST files



(a) Level-0 SST file (b) Level-1 SST file (c) Level-2 SST file (d) Level-3 SST file

Fig. 8: P99th Tail Latency of key-value pair from Level-0 to Level-3

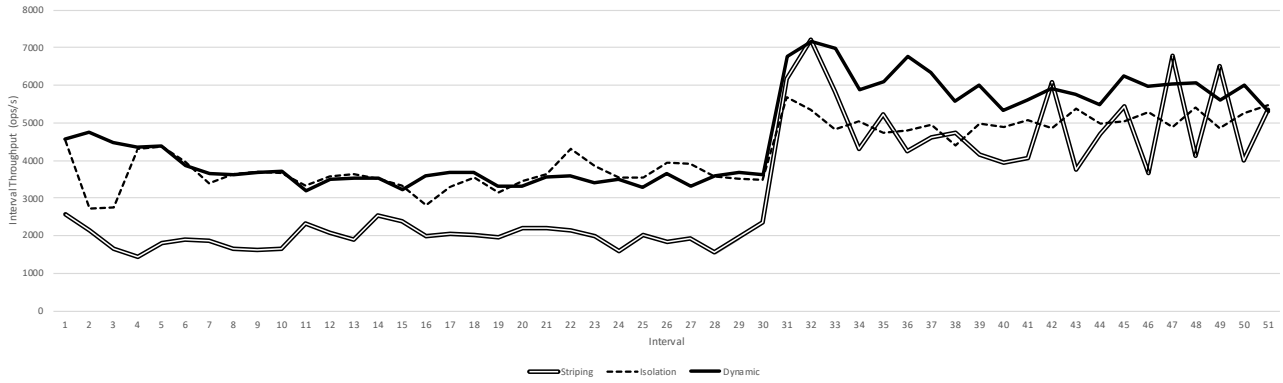


Fig. 9: Interval throughput comparison under changing workload - Striping VS Isolation VS Dynamic

interval throughput performance is plotted for each interval. During the first 30 intervals, the workload is read-intensive which has 10% of read-percentage, and the remaining 20 intervals, DapDB process write-intensive workloads with 90% of read operations. Experimental result shows that performance of the Isolation scheme represent same performance with the dynamic approach during the first 30 intervals. On the other hand, the Striping-Arrangement scheme shows relatively deteriorated performance results due to I/O interference. The static Isolation scheme under write-intensive workload after the 30th interval shows performance degradation due to degraded parallelism rather than performance improvement due to eliminated interference. However, the Dynamic-Arrangement scheme shows the best performance in all intervals because the workload profiler dynamically changes the arrangement of parallelism by monitoring the characteristics of the changing workload.

VII. Related Work

One of the solutions to solve the performance degradation of the database system in a mixed workload of reads and writes has been proposed using data replication [3]. In this paper, they proposed a storage scheme called Rails that the read performance is always predictable without interference, even if there are write requests by having a dedicated SSD for reading and writing and periodically synchronizing the two SSDs. Indeed, they achieved predictable read performance just as in a write-free environment. However, in their system, the same data must be duplicated in the two flash storage devices and held in different physical locations. That is, there is a disadvantage in that the capacity of the storage device can not be fully utilized.

Several studies have been proposed data management techniques for storage devices considering application context. In the multi-stream [4], the host gives stream information to place data having a similar access pattern through the same stream internally in the SSD, and the storage device utilizes it for

data storage. However, because the physical data placement and GC are still controlled by the SSD internal FTL, the I/O requests by the application may not work in concert with the internal operations of the SSD such as GC and wear-leveling. In order to solve these problems, an application-driven flash management scheme has been proposed, and cross-stack optimization has been attempted in consideration of the application context and NAND-flash characteristics [5] [19].

Our study is in line with these approaches [3], [4] in terms of physically separating read and write requests in SSD. In contrast, we enable adaptive SSD's parallelism management by fully take advantage of application context and I/O pattern.

VIII. Conclusion

In this paper, we present DapDB, an LSM-tree based key-value store tailored for Open-Channel SSDs. DapDB is designed to achieve both good performance of throughput and latency with two main technique: Isolation-Arrangement, Dynamic-Arrangement of parallelism. We implement DapDB on a real Open-Channel SSD using liblightnvm, a user-space I/O library. Experimental results show that DapDB improves overall performance under various workloads, i.e., 1.47× faster than existing greedy scheme that makes maximum use of SSD internal parallelism. Also, most importantly, DapDB achieves that dramatically reduced average and tail read latency up to 67% and 96% respectively depending on the workload. DapDB's performance fain mainly come from eliminated interference between I/O types and dynamic arrangement of parallelism based on workload characteristics. The open-source DapDB is available at *https://github.com/RockyLim92/rocksdb*.

References

[1] https://linux.die.net/man/1/fio

[2] https://github.com/facebook/rocksdb/wiki/Benchmarking-tools

[3] Skourtis, Dimitris, et al. "Flash on rails: Consistent flash performance through redundancy." 2014 USENIX Annual Technical Conference (USENIXATC 14). 2014.

[4] Kang, Jeong-Uk, et al. "The multi-streamed solid-state drive." 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14). 2014.

[5] González, Javier, et al. "Application-driven flash translation layers on open-channel SSDs." Proceedings of the 7th Non Volatile Memory Workshop (NVMW). 2016.

[6] O'Neil, Patrick, et al. "The log-structured merge-tree (LSM-tree)." Acta Informatica 33.4 (1996): 351-385.

[7] RocksdB, https://rocksdb.org

[8] Apache Cassandra, http://cassandra.apache.org

[9] Apache HBase, https://hbase.apache.org

[10] Open-Channel Solid State Drive Interface Specification, https://openchannelssd.readthedocs.io/en/latest/specification

[11] liblightnvm - User space I/O library for Open-Channel SSDs, http://lightnvm.io/liblightnvm

[12] https://github.com/RockyLim92/ocssd_bench

[13] https://github.com/facebook/rocksdb/wiki/Benchmarking-tools

[14] https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/960pro

[15] https://github.com/OpenChannelSSD/linux/tree/pblk.cnex

[16] https://github.com/RockyLim92/rocksdb

[17] Chen, Feng, Rubao Lee, and Xiaodong Zhang. "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing." 2011 IEEE 17th International Symposium on High Performance Computer Architecture. IEEE, 2011.

[18] Kang, Woon-Hak, et al. "Durable write cache in flash memory SSD for relational and NoSQL databases." Proceedings of the 2014 ACM SIGMOD international conference on Management of data. ACM, 2014.

[19] Lim, Heerak. "Application-Driven Flash Management: LSM-tree based Database Optimization through Read/Write Isolation." Proceedings of the Doctoral Symposium of the 19th International Middleware Conference. ACM, 2018.

[20] Matias Bjørling. "From Open-Channel SSDs to Zoned Namespaces" USENIX Association 2019.